



# **SQL Server 2008 Security Overview for Database Administrators**

White Paper

Published: January 2007

Updated: July 2008

**Summary:** This paper covers some of the most important security features in SQL Server 2008. It tells you how, as an administrator, you can install SQL Server securely and keep it that way even as applications and users make use of the data stored within.

For the latest information, see [Microsoft SQL Server 2008](#).

---

## Contents

Introduction .....	1
Secure Configuration .....	2
Windows Update .....	2
Surface Area Configuration .....	2
Authentication .....	3
Password Policy Enforcement .....	4
Endpoint Authentication.....	5
Authorization .....	7
Granular Permissions .....	8
Metadata Security .....	11
SQL Server Agent Proxies .....	12
Execution Context .....	16
User/Schema Separation .....	17
Encryption and Key Management .....	20
Data Encryption.....	20
Auditing in SQL Server 2008 .....	25
Conclusion .....	29

---

## Introduction

Security is becoming increasingly important as more networks are connected together. Your organization's assets must be protected, particularly its databases, which contain your company's valuable information. Security is one of the critical features of a database engine, protecting the enterprise against myriad threats. The security features of Microsoft® SQL Server® 2008 are designed to make it more secure and to make security more approachable and understandable to those who are responsible for data protection.

During the past few years, the concept of what a secure, computer-based system must be has been developing. Microsoft has been in the forefront of this development, and SQL Server is one of the first server products that fully implements that understanding. It enables the important principle of *least privilege* so you do not have to grant users more permissions than are necessary for them to do their jobs. It provides in-depth tools for defense so that you can implement measures to frustrate even the most skillful attackers.

Much has been written and discussed about the Microsoft Trustworthy Computing initiative that guides all software development at the company. For more information, see [Trustworthy Computing](#).

The four essential components of this initiative are:

- **Secure by design.** Software requires a secure design as a foundation for repelling attackers and protecting data.
- **Secure by default.** System administrators should not have to work to make a fresh installation secure; it should be that way by default.
- **Secure in deployment.** Software should help to keep itself updated with the latest security patches and assist in maintenance.
- **Communications.** Communicate best practices and evolving threat information so that administrators can proactively protect their systems.

These guiding principles are evident throughout SQL Server 2008, which provides all the tools you need to secure your databases.

This paper explores the most important security features for system and database administrators. It starts with a look at how SQL Server 2008 is straightforward to install and configure securely. It explores authentication and authorization features that control access to the server and determine what a user can do once authenticated. It finishes with a look at the database security features an administrator must understand in order to provide a secure environment for databases and the applications that access those databases.

---

## Secure Configuration

Nothing much has changed in the external security requirements of a server running SQL Server 2008. You need to physically secure the server and back up data regularly, put it behind one or more firewalls if it is connected to a network, avoid installing SQL Server on a computer with other server applications, and enable only the minimum network protocols required. Install SQL Server on a Microsoft Windows Server® 2003 or Microsoft Windows Server 2008 computer so that it has full advantage of operating system-level security protections.

The SQL Server 2008 installation program does all the usual installation tasks, and has a System Configuration Checker that notifies you of any deficiencies that might cause problems. Installing SQL Server 2008 does not enable all features by default. Instead, it installs the core essentials and widely used features. Other features that might not be needed in a production environment are turned off by default. You can use the supported tools to turn on just the features you need.

This is all part of the Trustworthy Computing *secure by default* mandate. Features that are not required by a basic database server are left uninstalled, resulting in a reduced surface area. Since by default not all features are enabled across all systems, a heterogeneity is introduced in terms of the install image of a system. Because this limits the number of systems that have features that are vulnerable to a potential attack, it helps defend against large-scale attacks or worms.

## Windows Update

New threats and vulnerabilities can be discovered after you deploy SQL Server in your enterprise. Windows Update is designed to ensure timely download and application of patches that significantly reduce specific security issues. You can use Windows Update to apply SQL Server 2008 patches automatically and reduce threats caused by known software vulnerabilities. In most enterprise environments, you should use the Windows Server Update Service to manage the distribution of patches and updates throughout the organization.

## Surface Area Configuration

SQL Server 2008 comes packed with numerous features, many of which are installed in a disabled state. For example, CLR integration, database mirroring, debugging, Service Broker, and mail functions are installed but are not running and not available until you explicitly turn them on or configure them. This design is consistent with the *reduction in surface area* paradigm of the "secure by default" philosophy of SQL Server, and leads to a reduced attack surface. If a feature is not available or enabled, an attacker cannot make use of it.

---

The tradeoff is that it can be time consuming to hunt down all of the Transact-SQL statements for turning on features. Even when you discover that the **sp\_configure system** stored procedure does much of what you need, you still must write non-intuitive code like the following:

```
sp_configure 'show advanced options', 1
reconfigure with override
sp_configure 'clr enabled', 1
```

There are far too many configuration options to take the time to write this kind of code— especially when you have multiple instances of SQL Server deployed throughout the organization. SQL Server 2008 includes a policy-based management technology. Policy-Based Management provides a number of configuration *facets*, each of which defines a set of related configuration settings or properties. You can use these facets to create *conditions* that specify the desired settings for the configuration options, and enforce these conditions as *policies* to SQL Server instances across the enterprise.

One the of the facets included in SQL Server 2008 is the **Surface Area** facet, and you can use this facet to define a policy that controls the status of various SQL Server 2008 features. By creating a policy that defines the desired surface area settings for your servers, you can easily enforce a minimal surface area on all SQL Server instances in your organization, and reduce the possibility of malicious attack.

## Authentication

Microsoft developed SQL Server 2000 at a time when data and servers required protection but did not have to withstand the relentless onslaught of attacks seen on the Internet today. The basic authentication question remains the same: *Who are you and how can you prove it?* SQL Server 2008 provides robust authentication features that provide better support at the security outskirts of the server for letting the good guys in and keeping the bad guys out.

SQL Server Authentication provides authentication for non-Windows®-based clients or for applications using a simple connection string containing user IDs and passwords. While this logon is easy to use and popular with application developers, it is not as secure as Windows authentication and is not the recommended authentication mechanism

SQL Server 2008 improves on the SQL Server Authentication option. First, it supports encryption of the channel by default through the use of SQL-generated certificates. Administrators do not have to acquire and install a valid SSL certificate to make sure that the channel over which the SQL credentials flow is secure. SQL Server 2008 automatically generates these certificates, and encrypts the channel automatically by default when transmitting login packets. This occurs if the client is at the SQL Server 2005 level or above.

**Note** The native certificate generated by SQL Server protects against passive man-in-the-middle attacks where the attacker is sniffing the network. To secure your systems more effectively against active man-in-the-middle attacks, deploy and use certificates that the clients trust as well.

SQL Server 2008 further enhances SQL Server Authentication because, by default, the database engine now uses Windows Group Policy for password complexity, password expiration, and account lockout on SQL logins when used in combination with a Windows Server 2003 or later. This means that you can enforce Windows password policy on your SQL Server accounts.

## Password Policy Enforcement

With SQL Server 2008, password policy enforcement is built into the server. Using the **NetValidatePasswordPolicy()** API, which is part of the NetAPI32 library on Windows Server 2003, SQL Server validates a password during authentication and during password set and reset, in accordance with Windows policies for password strength, expiration, and account lockout. The following table lists the settings that comprise the policy.

Windows Server 2003 Password Policy Components

Category	Name	Notes
Password Policy	Enforce password history	Prevents users from reusing old passwords, such as alternating between two passwords.
	Minimum password length	
	Password must meet complexity requirements	See text below.
	Store passwords using reversible encryption	Allows retrieving the password from Windows. You should never enable this, unless application requirements outweigh the need for secure passwords. (This policy does not apply to SQL Server.)
Password Expiration	Maximum password age	
	Minimum password age	
Account Lockout Policy	Account lockout duration	Duration of the account lockout in minutes. Windows enables this when the lockout threshold is > 0.
	Account lockout threshold	Maximum number of unsuccessful login attempts.
	Reset account lockout counter after	Time in minutes after which Windows resets the counter of unsuccessful attempts. Windows enables this when the lockout threshold is > 0.

---

If you are not running Windows Server 2003 or above, SQL Server still enforces password strength by using simple checks, preventing passwords that are:

- Null or empty
- The same as the name of computer or login
- Any of "password", "admin", "administrator", "sa", "sysadmin"

The same complexity standard is applied to all passwords you create and use in SQL Server, including passwords for the **sa** login, application roles, database master keys for encryption, and symmetric encryption keys.

SQL Server always checks the password policy by default, but you can suspend enforcement for individual logins with either the CREATE LOGIN or ALTER LOGIN statements as in the following code:

```
CREATE LOGIN bob WITH PASSWORD = 'S%V7V1v3c9Es8',  
CHECK_EXPIRATION = OFF, CHECK_POLICY = OFF
```

CHECK\_EXPIRATION uses the minimum and maximum password age part of the Windows Server 2003 policy, and CHECK\_POLICY uses the other policy settings.

Administrative settings allow turning on and off password policy checks, turning on and off password expiration checks, and forcing a password change the first time a user logs on. The MUST\_CHANGE option in CREATE LOGIN forces the user to change the password the next time they log on. On the client side, it allows a password change at logon. All of the new client-side data access technologies will support this, including OLE DB and ADO.NET, as well as client tools such as Management Studio.

If the user unsuccessfully attempts to log on too many times and exceeds the attempts allowed in the password policy, SQL Server locks the account, based on the settings in the Windows policy. An administrator can unlock the account with the ALTER LOGIN statement:

```
ALTER LOGIN alice WITH PASSWORD = '3x1Tq#PO^YIAz' UNLOCK
```

## Endpoint Authentication

SQL Server 2008 supports both the traditional, binary Tabular Data Stream for client access to data as well as native XML Web service access using HTTP. The primary benefit of allowing access via HTTP is that any client software and development tools that understand Web service protocols can access data stored in SQL Server. This means SQL Server 2008 can provide standalone Web service methods as well as be a complete endpoint in a Service Oriented Architecture (SOA).

---

Using SQL Server 2008 as a Web service host requires two general steps, each with plenty of possible variations:

- Defining stored procedures and user-defined functions that provide the Web service methods
- Defining an HTTP endpoint that receives method calls via HTTP and routes them to the appropriate procedure.

This paper focuses on the security issues involved. For details on configuring and using HTTP endpoints, see `CREATE ENDPOINT` (Transact-SQL) in SQL Server Books Online.

Because XML Web services in SQL Server uses HTTP and, by default, port 80, most firewalls allow the traffic to pass. However, an unprotected endpoint is a potential vector for attacks and you must secure it, so SQL Server has strong authentication and authorization. By default, SQL Server does not have any endpoints and you have to have a high level of permissions to create, alter, and enable HTTP endpoints.

SQL Server 2008 provides five different authentication types, similar to those used by IIS for Web site authentication.

- **Basic authentication.** Basic authentication is part of the HTTP 1.1 protocol, which transmits the login credentials in clear text that is base-64 encoded. The credential must map to a Windows login, which SQL Server then uses to authorize access to database resources. If you use Basic authentication, you cannot set the `PORTS` argument to `CLEAR` but must instead set it to `SSL` and use a digital certificate with SSL to encrypt the communication with the client software.
- **Digest authentication.** Digest authentication is also part of the HTTP 1.1 protocol. It hashes the credentials with MD5 before sending to the server so that credentials are not sent across the wire, even in encrypted form. The credentials must map to a valid Windows domain account; you cannot use local user accounts.
- **NTLM authentication.** NTLM uses the challenge-response protocol originally introduced in Microsoft Windows NT® and supported in all client and server versions of Windows since. It provides secure authentication when both client and server are Windows systems, and requires a valid domain account.
- **Kerberos authentication.** Kerberos authentication is available with Windows 2000 and later, based on an industry-standard protocol available on many operation systems. It allows for mutual authentication in which both the client and server are reasonably assured of the other's identity and provides a highly secure form of authentication. To use Kerberos on Windows Server 2003, you must register the Kerberos Service Principal Name (SPN) with `Http.sys` by using the `SetSPN.exe` utility that is part of the Windows Support Tools.
- **Integrated authentication.** Integrated authentication provides the best of NTLM and Kerberos authentication. The server uses whichever of the two authentication types the client requests, allowing the authentication the client supports while making the service available to older versions of Windows. You can configure `Http.sys` in Windows 2003 to negotiate with the client which protocol it should use.

---

The authentication method used for an endpoint is set with the **AUTHENTICATION** attribute of the **CREATE** or **ALTER ENDPOINT** statement. For example, the following code creates an endpoint that uses Kerberos for authentication:

```
CREATE ENDPOINT myEndpoint
STATE=STARTED
    AS HTTP (PATH = '/MyHttpEndpoint',
    AUTHENTICATION = (KERBEROS),
    PORTS = (CLEAR),
    SITE = 'MySqlServer')
FOR SOAP (WSDL = DEFAULT,
    DATABASE = 'myDB',
    NAMESPACE = 'http://example.com/MySqlServer/myDB/WebService')
```

SQL Server 2008 supports endpoints that listen to HTTP as well as a user-defined port on TCP. You can also format requests by using a variety of formats: SOAP, Transact-SQL, a format specific to Service Broker, and another used for database mirroring. When using SOAP you can take advantage of WS-Security headers to authenticate SQL Server logins.

Microsoft implemented Web Service endpoint authentication to support a wide variety of protocols and specifications, of which this paper touches on just a few. You will need to explicitly enable your authentication option and ensure that clients are able to provide the type of credentials required. After SQL Server authenticates the client, you can authorize the resources that the login is authorized to access, described in the next section.

## Authorization

After authentication, it is time to think about what an authenticated login can do. In this area, SQL Server 2008 and SQL Server 2005 are more flexible than earlier versions. Permissions are now far more granular so that you can grant the specific permissions required rather than granting membership in a fixed role that probably carries with it more permissions than are necessary. You now have far more entities, (securables) to which you can assign permissions that are more granular.

In addition to enhanced protection for user data, structural information and metadata about a particular securable is now available only to principals that have permission to access the securable.

Furthermore, it is possible to create custom permission sets using a mechanism that allows one to define the security context under which stored procedures can run.

---

In addition, SQL Server Agent uses a flexible proxy scheme to enable job steps to run and access required resources. All these features make SQL Server more complex but far more secure.

## Granular Permissions

One of the many ways that SQL Server 2008 and SQL Server 2005 are far more secure than earlier versions is the improved granularity of permissions. Previously, an administrator had to grant a user membership in a fixed server role or fixed database role to perform specific operations, but more often than not, those roles had permissions that were far too broad for simple tasks. The principle of *least privilege* requires that a user have only the minimum permissions to do a job, so assigning users to a broad role for narrow purposes violates this principle.

The set of fixed server and database roles is largely unchanged since SQL Server 2000, so you can still take advantage of those predefined bundles of permissions when users or applications require all or most of the defined permissions. Probably the biggest change is the addition of a **public** server role. However, the principle of least privilege mandates that you not use a role that is not a perfect fit for what the principal needs to do a job. Although it requires more work to discover and assign the permissions required for a principal, it can result in a far more secure database environment.

## Principals and Securables

In SQL Server 2008 a *principal* is any individual, group, or process that can request access to a protected resource and be granted permission to access it. As in previous versions of SQL Server, you can define a principal in Windows or you can base it on a SQL Server login with no corresponding Windows principal. The following list shows the hierarchy of SQL Server 2008 principals, excluding the fixed server and database roles, and how you can map logins and database users to security objects. The scope of the influence of the principal depends on the scope of its definition, so that a Windows-level principal is more encompassing than a SQL Server-level principal, which is more encompassing than a database-level principal. Every database user automatically belongs to the fixed **public** role.

### Windows-level principals

- Windows Domain login
- Windows Local login
- Windows group

---

## SQL Server-level principals

- SQL Server login
- SQL Server login mapped to a Windows login
- SQL Server login mapped to a certificate
- SQL Server login mapped to an asymmetric key

## Database-level principals

- Database user
- Database user mapped to SQL Server login
- Database user mapped to a Windows login
- Database user mapped to a certificate
- Database user mapped to an asymmetric key
- Database role
- Application role
- Public role

The other part of authorization is the objects that you can secure through the granting or denying of permissions. Figure 1 lists the hierarchy of securable objects in SQL Server 2008. At the server level, you can secure network endpoints to control the communication channels into and out of the server, as well as databases, bindings, and roles and logins. At the database and schema level, virtually every object you can create is securable, including those that reside within a schema.

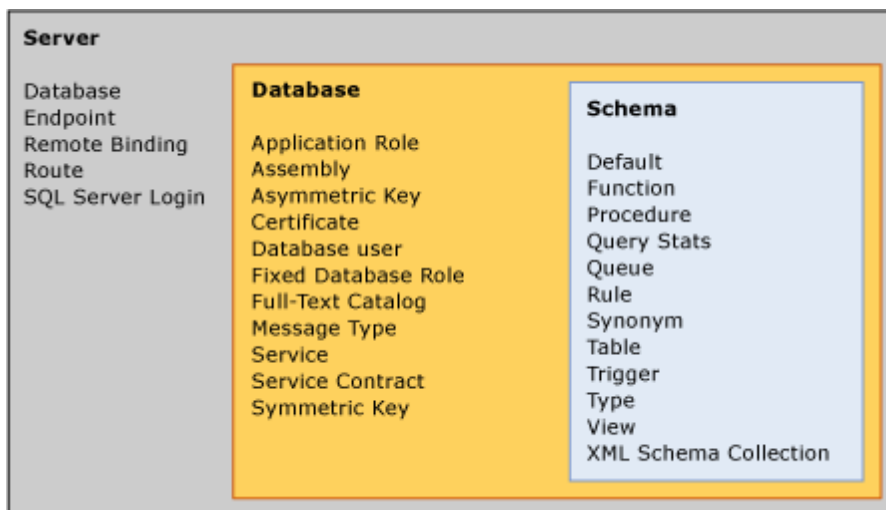


Figure 1: Securable object hierarchy in SQL Server 2008

---

## Roles and Permissions

For a sense of the number of permissions available in SQL Server you can invoke the **fn\_builtin\_permissions** system function:

```
SELECT * FROM sys.fn_builtin_permissions(default)
```

Following are the new permission types in SQL Server 2005:

- **CONTROL**. Confers owner-like permissions that effectively grant all defined permissions to the object and all objects in its scope, including the ability to grant other grantees any permission. CONTROL SERVER grants the equivalent of **sysadmin** privileges.
- **ALTER**. Confers permission to alter any of the properties of the securable objects except to change ownership. Inherently confers permissions to ALTER, CREATE, or DROP securable objects within the same scope. For example, granting ALTER permissions on a database includes permission to change its tables.
- **ALTER ANY <securable object>**. Confers permission to change any securable object of the type specified. For example, granting ALTER ANY ASSEMBLY allows changing any .NET assembly in the database, while at the server level granting ALTER ANY LOGIN lets the user change any login on that server.
- **IMPERSONATE ON <login or user>**. Confers permission to impersonate the specified user or login. As you will see later in this white paper, this permission is necessary to switch execution contexts for stored procedures. You also need this permission when doing impersonating in a batch.
- **TAKE OWNERSHIP**. Confers the permission to the grantee to take ownership of the securable, using the ALTER AUTHORIZATION statement.

SQL Server 2008 still uses the familiar GRANT, DENY, and REVOKE scheme for assigning or refusing permissions on a securable object to a principal. The GRANT statement now covers all of the new permission options, such as the scope of the grant and whether the principal can grant the permission to other principals. SQL Server does not allow cross-database permissions. To grant such permissions, create a duplicate user in each database and separately assign each database's user the permission.

Like earlier versions of SQL Server, activating an application role suspends other permissions for the duration that the role is active. However, in SQL Server 2008 and SQL Server 2005, you can unset an application role. Another difference between SQL Server 2000 and later versions is that when activating an application role, the role also suspends any server privilege, including **public**. For example, if you grant VIEW ANY DEFINITION to **public**, the application role will not honor it. This is most noticeable when accessing server-level metadata under an application role context.

**Note** The new, preferred alternative to application roles is to use execution context in code modules. For more information, see [Execution Context](#) in this paper.

Granting a particular permission can convey the rights of other permissions by implication. The ALTER permission on a schema, for example, "covers" more granular and lower-level permissions that are "implied." Figure 2 displays the implied permissions for ALTER SCHEMA. See "Covering/Implied Permissions (Database Engine)" in SQL Server Books Online for the Transact-SQL code for an `ImplyingPermissions` user-defined function that assembles the hierarchy list from the `sys.fn_built_in_permissions` catalog view and identifies the depth of each permission in the hierarchy. After adding `ImplyingPermissions` to the master database, I executed the following statement to produce Figure 2, passing in the object and permission type:

```
SELECT * FROM master.dbo.ImplyingPermissions('schema', 'alter')
      ORDER BY height, rank
```

This is a great way to explore the permissions hierarchy in SQL Server 2008.

	permname	class	height	rank
1	ALTER	SCHEMA	0	0
2	CONTROL	SCHEMA	0	1
3	ALTER ANY SCHEMA	DATABASE	1	1
4	ALTER	DATABASE	1	2
5	CONTROL	DATABASE	1	3
6	ALTER ANY DATABASE	SERVER	2	3
7	CONTROL SERVER	SERVER	2	4

Figure 2: Hierarchy of implied permissions of ALTER SCHEMA

When you consider the number and types of principals available, the number of securable objects in the server and a typical database, and the sheer number of available permissions and the covered and implied permissions, it quickly becomes clear just how granular permissions can be in SQL Server 2008. Creating a database now requires a much more detailed analysis of its security needs and careful control of permissions on all objects. Nevertheless, this analysis is well worth it and using the capabilities in SQL Server 2008 results in more secure databases.

## Metadata Security

One benefit of the granular permission scheme is that SQL Server protects metadata as well as data. Prior to SQL Server 2005, a user with any access to a database could see the metadata of all objects within the database, regardless of whether the user could access the data within it or execute a stored procedure.

SQL Server 2008 examines the permissions a principal has within the database and reveals the metadata of an object only if the principal is the owner or has some permission on the object. There is also a `VIEW`

---

DEFINITION permission that can grant permission to view metadata information even without other permissions in the object.

This protection extends to error messages returned from operations to access or update an object to which the user has no access. Rather than acknowledging that there is indeed a table named Address, and giving an attacker confirmation that he or she is on track, SQL Server returns an error message with alternate possibilities. For example, if a user with no permissions on any objects in the database attempts to drop the Address table, SQL Server displays the following error message:

```
Msg 3701, Level 14, State 20, Line 1
Cannot drop the table 'Address', because it does not exist or you do
not have permission.
```

This way, an attacker gets no confirmation that an Address table actually exists. However, someone debugging this problem still only has a limited number of possibilities to explore.

## SQL Server Agent Proxies

One of the best examples of the authorization model in SQL Server 2008 is SQL Server Agent. You can define various credentials often associated with Windows logins, linked to users with the necessary permissions to perform one or more SQL Server Agent steps. A SQL Server Agent proxy then links a credential with a job step to provide the necessary permissions.

This provides a granular means of following the principle of least privilege: granting a job step the permissions it needs and no more. You can create as many proxies as you wish, associating each of them with one or more SQL Server Agent subsystems. This is in stark contrast to the all-powerful proxy account in SQL Server 2000, which let the user create job steps in any of the SQL Server Agent subsystems.

**Note** When you upgrade a server from SQL Server 2000, a single proxy account is created and all subsystems are assigned to that single proxy account so that existing jobs will continue to run. After upgrading, create credentials and proxy accounts to implement a more secure, granular set of proxies to protect server resources.

Figure 3 shows the Object Explorer in Management Studio with a list of subsystems available in SQL Server Agent. Each subsystem can have one or more proxies associated with it that grant the appropriate permissions for a job step. The one exception is that Transact-SQL subsystems execute with the permissions of the module owner as they did in SQL Server 2000.

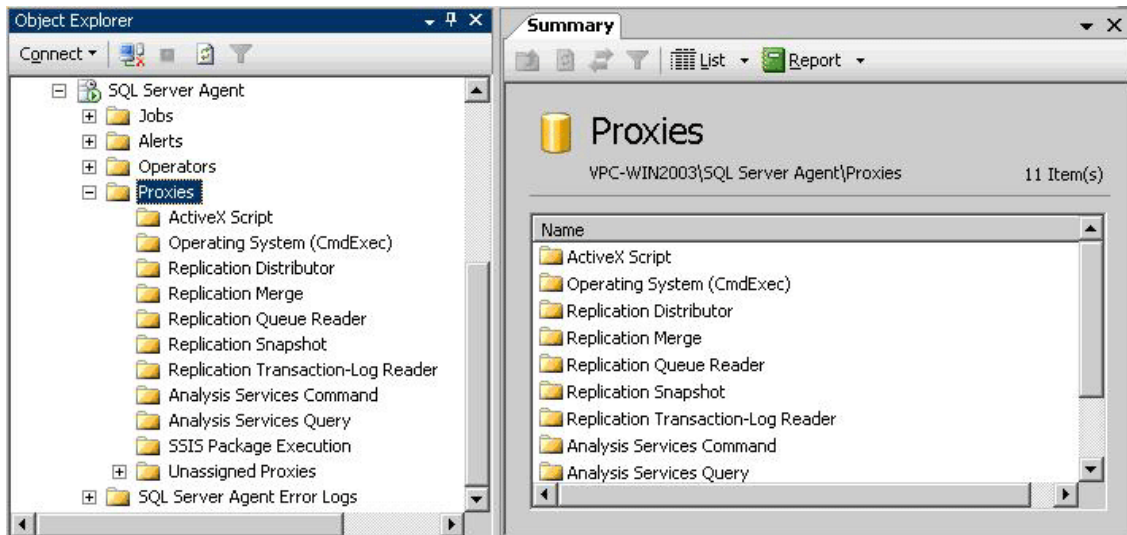


Figure 3: SQL Server Agent subsystems you can associate with proxies

Upon a fresh installation of SQL Server, only the System Administrator role has permissions to maintain SQL Server Agent jobs, and the management pane in the Management Studio Object Explorer is only available to **sysadmins**. SQL Server 2008 makes available a few other roles you can use to grant various levels of permissions. You can assign users to the **SQLAgentUser**, **SQLAgentReaderRole**, or **SQLAgentOperator** roles, each of which grants increasing levels of permission to create, manage, and run jobs, or the **MaintenanceUser** role, which has all the permissions of **SQLAgentUser** plus the ability to create maintenance plans.

Members of the **sysadmin** role, of course, can do anything they want in any subsystem. To grant any other user rights to use subsystems requires the creation of at least one proxy account, which can grant rights to one or more subsystems. Figure 4 shows how a proxy account, MyProxy, is assigned to multiple principals—here a user and a role. The proxy account uses a credential, which links it to an account, usually a domain account, with permissions in the operating system necessary to perform whatever tasks are required by the subsystem. Each proxy can have one or more subsystems associated with it that grant the principal the ability to run those subsystems.

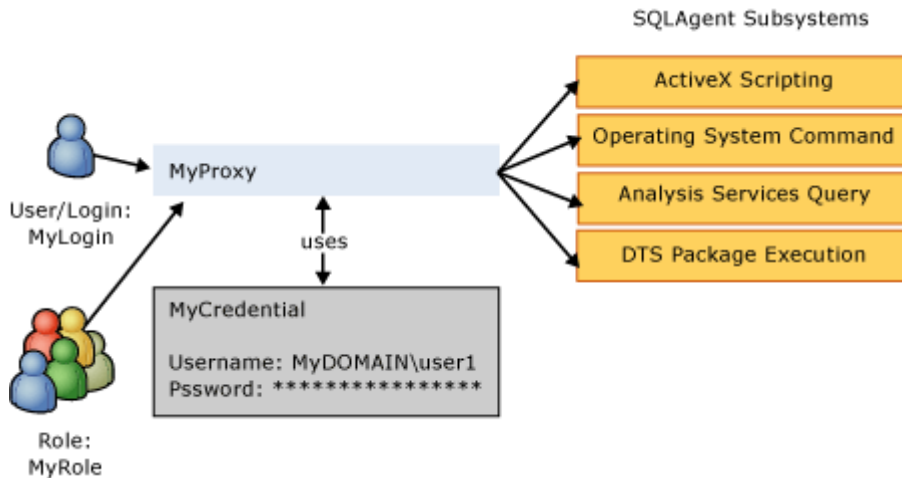


Figure 4: SQL Server Agent proxy account for various subsystems

The following code shows the Transact-SQL code necessary to implement the scheme shown in the figure. It starts by creating a credential, a database object that provides the link to the operating system account with rights to perform the desired actions in the subsystems. Then it adds a proxy account, MyProxy, which is really just a friendly name for the credential. Next, it assigns the proxy to two principals, here a SQL Server login and a custom role. Finally it associates the proxy with each of the four SQL Server Agent subsystems.

```

CREATE CREDENTIAL MyCredential WITH IDENTITY = 'MyDOMAIN\user1'
GO
msdb..sp_add_proxy @proxy_name = 'MyProxy',
    @credential_name = 'MyCredential'
GO
msdb..sp_grant_login_to_proxy @login_name = 'MyLogin',
    @proxy_name = 'MyProxy'
GO
msdb..sp_grant_login_to_proxy @login_name = 'MyRole',
    @proxy_name = 'MyProxy'
GO

sp_grant_proxy_to_subsystem @proxy_name = 'MyProxy',
    @subsystem_name = 'ActiveScripting'
GO
sp_grant_proxy_to_subsystem @proxy_name = 'MyProxy',
    @subsystem_name = 'CmdExec'
GO

```

```

sp_grant_proxy_to_subsystem @proxy_name = 'MyProxy',
    @subsystem_name = 'ANALYSISQUERY'

GO

sp_grant_proxy_to_subsystem @proxy_name = 'MyProxy',
    @subsystem_name = 'DTS'

GO

```

SQL Server Management Studio provides full support for creating credentials and proxies as shown in Figure 5. This creates the same proxy as the previous code.

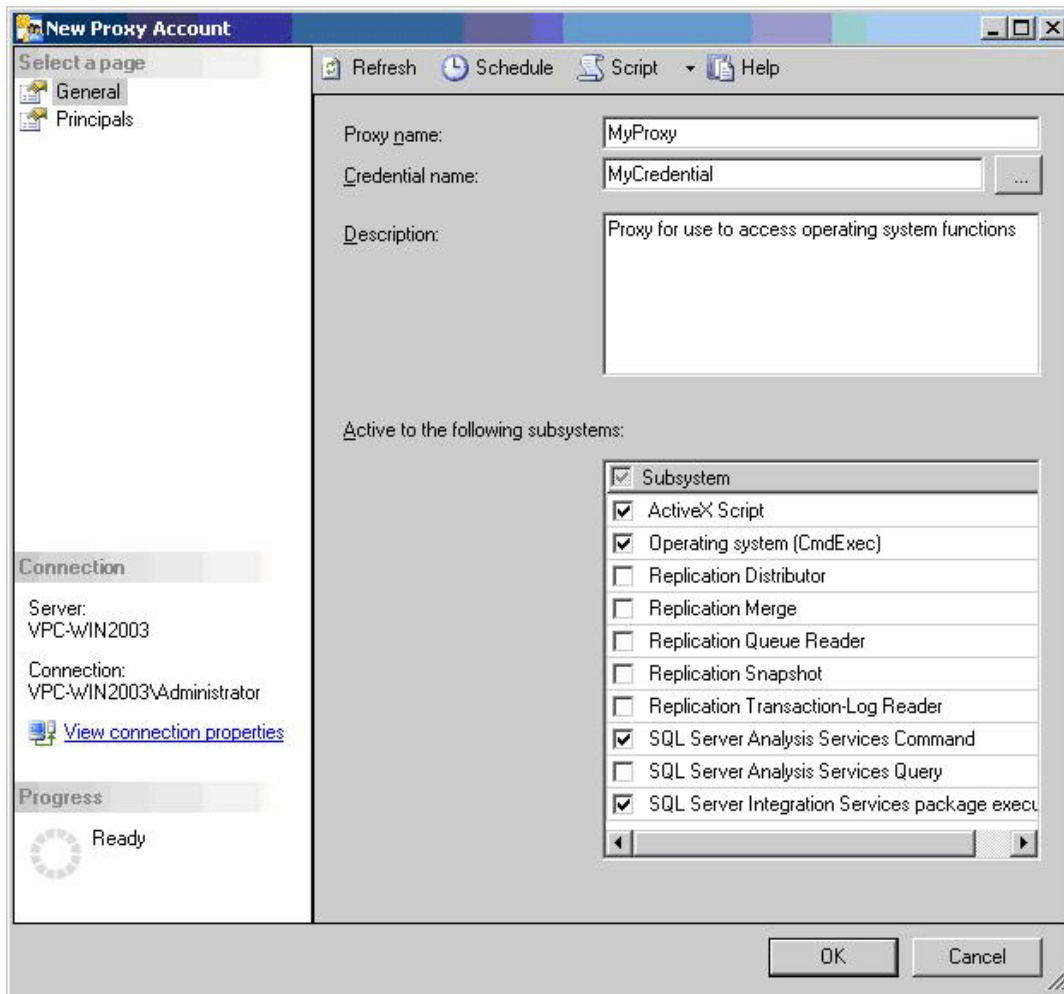


Figure 5: A new SQL Server Agent proxy in SQL Server Management Studio

A proxy is not a way to circumvent security in the operating system. If the credential used with a proxy doesn't have the permission in Windows, such as to write to a directory across the network, the proxy won't have it either. You can also use a proxy to grant limited execution rights to **xp\_cmdshell**, since it is a favorite tool used by attackers to extend their reach into the network once they compromise a SQL Server computer. The proxy provides this protection

---

because even if the principal has unlimited rights on the network, such as a domain administrator, any commands executed through the proxy have only the limited rights of the credential account.

## Execution Context

SQL Server has long supported the concept of ownership chaining as a way of ensuring that administrators and application developers have a way to check permissions upfront on the entry points to the database rather than being required to provision permissions on all objects accessed. As long as the user calling the module (stored procedure or function) or view had execute permissions on the module, or select permissions on the view, and the owner of the module, or view, was the owner of the objects accessed (an ownership chain), no permissions were checked on the underlying objects, and the caller received the data requested.

If the ownership chain was broken because the owner of the code did not own the referenced object, SQL Server checked the permissions against the caller's security context. If the caller had permission to access the object, SQL Server returned the data. If he or she did not, SQL Server raised an error.

Ownership chaining has some limitations; it applies only to data manipulation operations and not to dynamic SQL. Moreover, if you access objects across ownership boundaries, ownership chaining is not possible. Hence, this upfront permissions checking behavior only works for certain cases.

SQL Server 2008 includes the ability to mark modules with an execution context, such that the statements within the module can execute as a particular user as opposed to the calling user. This way, while the calling user still needs permissions to execute the module, SQL Server checks the permissions for statements within the module against the execution context of the module. You can use this behavior to overcome some of the shortcomings of ownership chaining because it applies to all statements within the module. Administrators wanting to perform upfront permission checking can use the execution context to do that.

Now when you define user-defined functions (except inline table-valued), stored procedures, and triggers you can use the EXECUTE AS clause to specify which user's permissions SQL Server uses to validate access to objects and data referenced by the procedure:

```
CREATE PROCEDURE GetData(@Table varchar(40))
    WITH EXECUTE AS 'User1'
```

---

SQL Server 2008 provides four EXECUTE AS options.

- **EXECUTE AS CALLER** specifies that the code executes in the security context of the caller of the module; no impersonation occurs. The caller must have access permissions on all of the objects referenced. However, SQL Server only checks permissions for broken ownership chains, so if the owner of the code also owns the underlying objects, only the execute permission of the module is checked. This is the default execution context for backward compatibility.
- **EXECUTE AS 'user\_name'** specifies that the code executes in the security context of the specified user. This is a great option if you do not want to rely on ownership chaining. Instead, you create a user with the necessary permissions to run the code and create custom permission sets.
- **EXECUTE AS SELF** is a shortcut notation for specifying the security context of the user who is creating or altering the module. SQL Server internally saves the actual user name associated with the module rather than "SELF."
- **EXECUTE AS OWNER** specifies that the security context is that of the current owner of the module at the time of module execution. If the module has no owner the context of the containing schema's owner is used. This is a great option when you want to be able to change the module's owner without changing the module itself.

Any time the user context changes using the EXECUTE AS option, the creator or alterer of the module must have IMPERSONATE permissions for the specified user. You cannot drop the specified user from the database until you have changed the execution context of all modules to other users.

## User/Schema Separation

SQL Server 2000 had no concept of a schema, which the ANSI SQL-99 specification defines as a collection of database objects owned by a single principal that forms a single namespace of objects. A schema is a container for database objects such as tables, views, stored procedures, functions, types, and triggers. It functions much as a namespace functions in the .NET Framework and XML, a way to group objects so that a database can reuse object names, such as allowing both dbo.Customer and Fred.Customer to exist in a single database, and to group objects under different owners.

**Note** You will need to use catalog views such as **sys.database\_sys.principals**, **sys.schemas**, **sys.objects**, and so forth. This is because the old **sysobjects** system table did not support schemas, and so was incapable of supporting U/S separation. Besides, the old catalog views are deprecated, so they will be dropped in a future version of SQL Server.

The top portion of Figure 6 shows how schemas worked in SQL Server 2000. When an administrator created a user named Alice in a database, SQL Server automatically created a schema named Alice that hid behind Alice the user. If Alice logged on to a server running SQL Server without database ownership and created Table1, the actual name of the table was Alice.Table1. The same held for other objects Alice created, such as Alice.StoredProcedure1 and

Alice.View1. If Alice is a database owner or a **sysadmin**, the objects she creates would be part of the **dbo** schema instead. Although we used to say that **dbo** owned the objects, it amounts to the same thing.

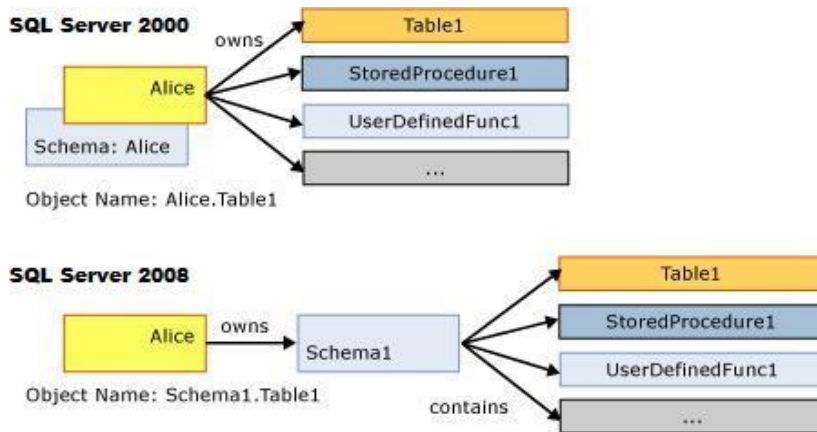


Figure 6: User/schema/objects in SQL Server 2000 and 2008

The problem with the unification of users and schemas in SQL Server 2000 arises when you need to change the ownership of objects, such as when Alice leaves the company and Lucinda takes over Alice's job. A system administrator would have to change ownership of all of the objects owned by Alice to Lucinda. More of a problem is that you would have to change any Transact-SQL or client application code that referred to Alice.Table1 to Lucinda.Table1 after Lucinda took ownership of the table. Depending on the number of objects Alice owns and how many applications had the name embedded in them; this could be a major undertaking. Microsoft has long recommended that the built-in **dbo** user owns all database objects to get around these problems. It was far easier to change a database's ownership than to change many objects and client applications.

**Note** Do not be confused by the SQL Server 2000 CREATE SCHEMA statement, which was just an easy way to create tables and views owned by a particular user and to grant permissions. You could use the statement to name a schema's owner but not to name the schema. SQL Server still irrevocably linked the owner to the schema with all the problems of changing ownership.

SQL Server 2008 cleans this up and implements the SQL-99 schema by separating the user from the schema as shown in the bottom part of Figure 6. When you create a new user Alice using the new CREATE USER DDL, SQL Server no longer automatically creates a schema with the same name. Instead, you must explicitly create a schema and assign ownership of it to a user. Because all of the database objects shown are now contained in the Schema1 schema, which Alice initially owns, it becomes simple to change ownership of all the schema's objects by simply changing the ownership of the schema to Lucinda. Each user can also have a default schema assigned to it, so that SQL Server assumes any objects referenced by name without the

---

schema reference to be in the default schema. In the bottom part of Figure , if Alice has Schema1 as her default schema, she can refer to the table as either Schema1.Table1 or simply as Table1. User Carol, who perhaps does not have a default schema associated with her user name, would have to refer to the table as Schema1.Table1. Any user without a default schema defined has **dbo** as the default.

Fully qualified object names in SQL Server 2008 have a four-part structure, similar to those in earlier versions of SQL Server:

```
server.database.schema.object
```

As in earlier versions, you can omit the server name if the object is on the same server as that where the code is running. You can omit the database name if the connection has the same database open, and you can omit the schema name if it is either the default schema for the current user or is owned by dbo, since that is the schema of last resort as SQL Server tries to disambiguate an object name.

Use the CREATE USER statement, instead of **sp\_adduser**, to create new users. This system stored procedure is still around for backward compatibility and has been changed a bit to conform to the new separation of users from schemas. **sp\_adduser** creates a schema with the same name as the new user name or the application role and assigns the schema as the default schema for the user, mimicking SQL Server 2000 behavior but providing a separate schema.

**Note** When using the ALTER AUTHORIZATION statement, it is possible to arrive in a state where YOU own a table in MY schema (or vice versa). This has some serious implications. For example, who owns the trigger on that table, you or me? The bottom line is that it can now be very tricky to discover the true owner of a schema-scoped object or type. There are two ways to get around this:

- Use OBJECTPROPERTY(id, 'OwnerId') to discover the true owner of an object.
- Use TYPEPROPERTY(type, 'OwnerId') to discover the true owner of a type.

SQL Server 2008 can help save keystrokes with synonyms. You can create a synonym for any object using the two, three, or four-part full object name. SQL Server uses the synonym to access the defined object. In the following code, the History synonym represents the specified schema.table in the **AdventureWorks** database. The SELECT statement returns the contents of the EmployeeDepartmentHistory table.

```
USE AdventureWorks
GO
CREATE SYNONYM History FOR HumanResources.EmployeeDepartmentHistory
SELECT * FROM History
```

---

**Note** The administrator or owner must grant permission on the synonym if someone else is to use it. GRANT SELECT on a synonym to a view or table or table-valued function. GRANT EXECUTE on a synonym to a procedure or scalar function, etc.

You could also define the History synonym for the complete, four-part name as in the following code:

```
CREATE SYNONYM History
    FOR
MyServer.AdventureWorks.HumanResources.EmployeeDepartmentHistory
```

Using the full, four-part name like this allows the use of the synonym from another database context, assuming the current user has permissions to use the synonym and read the table:

```
USE pubs
SELECT * FROM AdventureWorks..History
```

Note too that if you do not provide a schema name as part of the new synonym name, it will be part of the default schema.

## Encryption and Key Management

Security at the server level is probably the greatest concern for system administrators, but the database is where all the action is in a production environment. For the most part, a database administrator can let the database developer worry about the details in the database, as long as the developer works within the constraints of the environment. SQL Server 2008 provides plenty of features for securing the database.

### Data Encryption

SQL Server 2000 and earlier versions did not have built-in support for encrypting the data stored in a database. Why would you need to encrypt data that is stored in a well-secured database on a secure server nestled safely behind state-of-the-art firewalls? Because of an important, age-old security principal called *defense in depth*. Defense in depth means layering defenses so that even if attackers successfully pierce your outermost defenses they still must get through layer after layer of defense to get to the prize. In a database, it means that if an attacker gets through the firewall and through Windows security on the server to the database, he or she still has to do some brute force hacking to decrypt your data. In addition, in these days of legislated data and privacy protection, data must have strong protection.

SQL Server 2008 has rich support for various types of data encryption using symmetric and asymmetric keys, and digital certificates. Best of all, it takes

care of managing the keys for you, since key management is by far the hardest part of encryption. Keeping secrets secret is never easy.

As an administrator, you will probably need to manage at least the upper level of keys in the hierarchy shown in Figure 7. Database administrators must understand the service master key at the server level and the database master key at the database level. Each key protects its child keys, which in turn protect their child keys, down through the tree. The one exception is where a password protects a symmetric key or certificate, which is how SQL Server lets users manage their own keys and take responsibility for keeping the key secret.

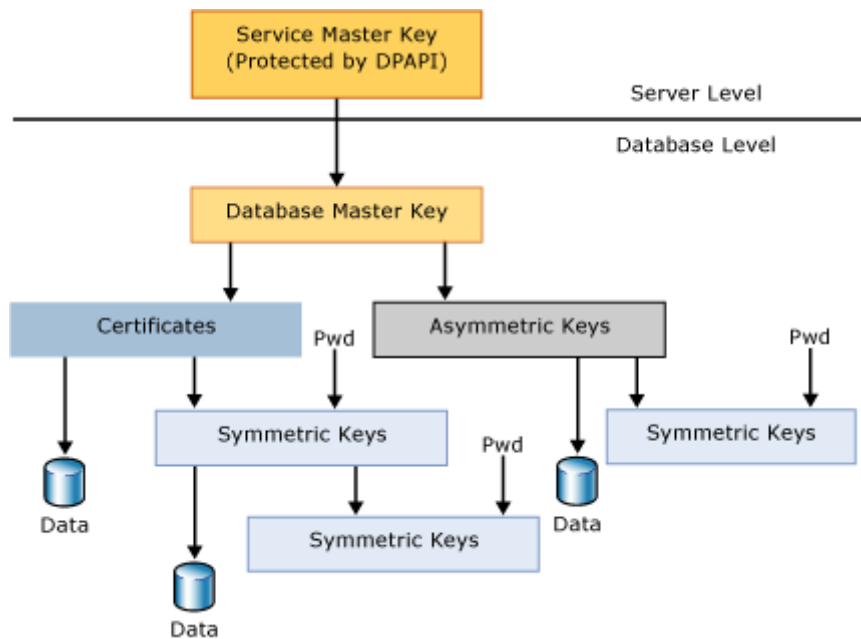


Figure 7: Encryption key hierarchy in SQL Server 2008

**Note** Microsoft recommends against using certificates or asymmetric keys for encrypting data directly. Asymmetric key encryption is many times slower and the amount of data that you can protect using this mechanism is limited, depending on the key modulus. You can protect certificates and asymmetric keys using a password instead of by the database master key.

The service master key is the one key that rules them all, all the keys and certificates in SQL Server. It is a symmetric key that SQL Server creates automatically during installation. It is obviously a critical secret because if it is compromised an attacker can eventually decipher every key in the server that is managed by SQL Server. The Data Protection API (DPAPI) in Windows protects the service master key.

SQL Server manages the service master key for you, although you can perform maintenance tasks on it to dump it to a file, regenerate it, and restore it from a file. However, most of the time you will not need or want to make any

---

of these changes to the key. It is advisable for administrators to back up their service master keys in case of key corruption.

Within the scope of a database, the database master key is the root encryption object for all keys, certificates, and data in the database. Each database can have a single master key; you will get an error if you try to create a second key. You must create a database master key before using it by using the CREATE MASTER KEY Transact-SQL statement with a user-supplied password:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'EOhnDGS6!7JKv'
```

SQL Server encrypts the key with a triple DES key derived from the password as well as the service master key. The first copy is stored in the database while the second is stored in the master database. Having the database master key protected by the database master key makes it possible for SQL Server to decrypt the database master key automatically when required. The end application or user does not need to open the master key explicitly using the password and is a major benefit of having the keys protected in the hierarchy.

Detaching a database with an existing master key and moving it to another server can be an issue. The problem is that the new server's database master key is different from that of the old server. As a result, the server cannot automatically decrypt the database master key. This can be circumvented by opening the database master key with the password with which it is encrypted and using the ALTER MASTER KEY statement to encrypt it by the new database master key. Otherwise, you always have to open the database master key explicitly before use.

Once the database master key exists, developers can use it to create any of three types of keys, depending on the type of encryption required:

- Asymmetric keys, used for public key cryptography with a public and private key pair
- Symmetric keys, used for shared secrets where the same key both encrypts and decrypts data
- Certificates, essentially wrappers for a public key

With all the encryption options and its deep integration into the server and database, encryption is now a viable way to add a final layer of defense to your data. Nevertheless, use the tool judiciously because encryption adds a lot of processing overhead to your server.

## Transparent Data Encryption

In SQL Server 2005, you can encrypt data in the database by writing custom Transact-SQL that uses the cryptographic capabilities of the database engine. SQL Server 2008 improves upon this situation by introducing transparent data encryption.

---

Transparent data encryption performs all of the cryptographic operations at the database level, which removes any need for application developers to create custom code to encrypt and decrypt data. Data is encrypted as it is written to disk, and decrypted as it is read from disk. By using SQL Server to manage encryption and decryption transparently, you can secure business data in the database without requiring any changes to existing applications, as shown in Figure 8.

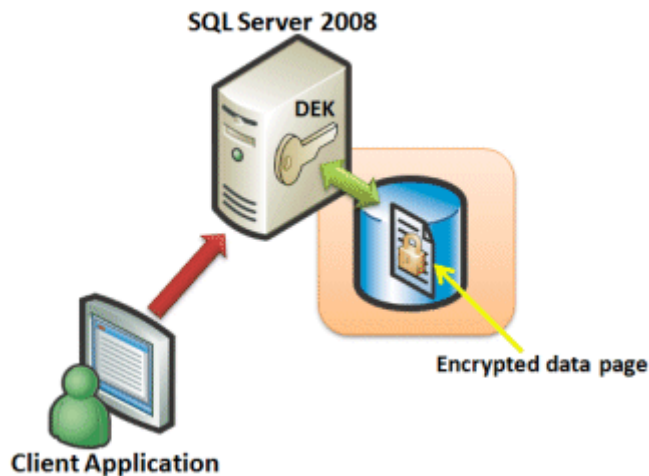


Figure 8: Transparent data encryption

A Database Encryption Key (DEK) is used to perform the encryption and decryption, and this DEK is stored in the database boot record for availability during recovery scenarios. You can use a service master key or Hardware Security Module (HSM) to protect the DEK. HSMs are usually USB devices or smartcards and are therefore less likely to be stolen or lost.

### Extensible Key Management

With the growing demand for regulatory compliance and the overall concern for data privacy, more organizations are using encryption as a way to provide a defense-in-depth solution. As organizations increasingly use encryption and keys to secure their data, key management becomes more complex. Some high security databases use thousands of keys and you must employ a system to store, retire, and regenerate these keys. Furthermore, you should store these keys separately from the data to improve security.

SQL Server 2008 exposes encryption functionality for use by third party vendors. These solutions work seamlessly with SQL Server 2005 and SQL Server 2008 databases and provide enterprise-wide dedicated key management. This moves the key management workload from SQL Server to a dedicated key management system.

Extensible key management in SQL Server 2008 also supports the use of HSMs to provide the physical separation of keys from data.

---

## Code Module Signing

One of the nice benefits of having encryption within SQL Server is that it provides the ability to sign code modules digitally (stored procedures, functions, triggers, and event notifications) with certificates. This provides much more granular control over access to database tables and other objects. Like encrypting data, you sign the code with the private key contained within the certificate. The result is that the tables used in the signed code module are accessible only through the code and not allowed outside of the code module. In other words, access to the tables is only available using the certificates that have been used to sign the module.

The effect can be the same with a stored procedure. For example, if it has an unbroken ownership chain, you carefully control which users get EXECUTE permission on the procedure, and you deny direct access to the underlying tables. But this doesn't help in situations such as when the procedure has a broken ownership chain or executes dynamic SQL, requiring that the user executing the procedure have permissions to the underlying tables. Another way to achieve the same effect is to use EXECUTE AS, but this changes the security context under which the procedure executes. This may not be desirable, for example, if you need to record in the table the user who actually caused the procedure to run (short of requiring a user name as a parameter to the procedure).

Signing code modules has the additional benefit of protecting against unauthorized changes to the code module. Like other documents that are digitally signed, the certificate is invalidated when the code changes. The code doesn't execute under the context of the certificate, so any objects that have their access provisioned to the certificate will not be accessible.

To do this, you create a certificate, associate it with a new user, and sign the procedure with the certificate. Grant this user whatever permissions are necessary to execute the stored procedure. In essence, you have added this user to the security context of the stored procedure as a secondary identity. Then grant execute permissions to whatever users or roles need to execute the procedure.

---

The following code shows these steps. Assume that you want to sign the mySchema.GetSecretStuff procedure, and that all of the referenced objects already exist in the database:

```
CREATE CERTIFICATE certCodeSigning
                                ENCRYPTION BY
PASSWORD = 'cJI%V4!axnJXfLC'
                                WITH SUBJECT =
'Code signing certificate'
GO
-- Sign the stored procedure
ADD SIGNATURE TO mySchema.GetSecretStuff BY CERTIFICATE
certCodeSigning
                                WITH PASSWORD =
'cJI%V4!axnJXfLC'
GO
-- Map a user to the certificate
CREATE USER certUser FOR CERTIFICATE certCodeSigning
GO
--Assign SELECT permissions to new certUser
GRANT SELECT ON SocialSecurity TO certUser
GO
-- Grant execute permission to the user who will run the code
GRANT EXECUTE ON mySchema.GetSecretStuff TO ProcedureUser
GO
```

Now only users explicitly granted EXECUTE permission on the stored procedure are able to access the table's data.

### **Auditing in SQL Server 2008**

An important part of any security solution is the ability to audit actions for accountability and regulatory compliance reasons. SQL Server 2008 includes a number of features that make it possible to audit activity.

---

## All Action Audit

SQL Server 2008 includes auditing support through the **Audit** object, which enables administrators to capture activity in the database server and store it in a log. With SQL Server 2008, you can store audit information in the following destinations:

- File
- Windows Application Log
- Windows Security Log

To write to the Windows Security Log, the SQL Server service must be configured to run as Local System, Local Service, Network Service, or a domain account that has the **SeAuditPrivilege** privilege and that is not an interactive user.

To create an **Audit** object, you must use the CREATE SERVER AUDIT statement. This statement defines an **Audit** object, and associates it with a destination. The specific options used to configure an **Audit** object depend on the audit destination. For example, the following Transact-SQL code creates two **Audit** objects; one to log activity to a file, and the other to log activity to the Windows Application log.

```
CREATE SERVER AUDIT HIPAA_File_Audit
    TO FILE ( FILEPATH='\\SQLPROD_1\Audit\' );

CREATE SERVER AUDIT HIPAA_AppLog_Audit
    TO APPLICATION_LOG
    WITH ( QUEUE_DELAY = 500, ON_FAILURE = SHUTDOWN);
```

Note that when logging to a file destination, the filename is not specified in the CREATE SERVER AUDIT statement. Audit file names take the form *AuditName\_AuditGUID\_nn\_TS.sqlaudit* where *AuditName* is the name of the Audit object, *AuditGUID* is a unique identifier associated with the **Audit** object, *nn* is a partition number used to partition file sets, and *TS* is a timestamp value. For example, the **HIPAA\_FILE\_Audit Audit** object created by the previous code sample could generate a log file with a name similar to the following:

```
HIPAA_File_Audit_{95A481F8-DEF3-40ad-B3C6-126B68257223}_00_29384.sqlaudit
```

You can use the QUEUE\_DELAY audit option to implement asynchronous auditing for performance reasons, and the ON\_FAILURE option determines the action to be taken if the audit information cannot be written to the destination. In the previously shown **HIPAA\_AppLog\_Audit** example, the

---

ON\_FAILURE option is configured to shut down the SQL Server instance if the log cannot be written to; in this case, the user who executes the CREATE SERVER AUDIT statement must have SHUTDOWN permission.

After you create an **Audit** object, you can add events with it by using the CREATE SERVER AUDIT SPECIFICATION and CREATE DATABASE AUDIT SPECIFICATION statements. The CREATE SERVER AUDIT SPECIFICATION adds server-level action groups (that is, pre-defined sets of related actions that can occur at the server level) to an **Audit**. For example, the following code adds the **FAILED\_LOGIN\_GROUP** action group (which records failed login attempts) to the **HIPAA\_File\_Audit** Audit.

```
CREATE SERVER AUDIT SPECIFICATION Failed_Login_Spec
FOR SERVER AUDIT HIPAA_File_Audit
    ADD (FAILED_LOGIN_GROUP);
```

The CREATE DATABASE AUDIT SPECIFICATION statement adds database-level action groups and individual database events to an **Audit**. Adding individual actions enables you to filter the actions that are logged based on the objects and users involved in the action. For example, the following code sample adds the DATABASE\_OBJECT\_CHANGE\_GROUP action group (which records any CREATE, ALTER, or DROP operations in the database) and any INSERT, UPDATE, or DELETE statement performed on objects in the **Sales** schema by the **SalesUser** or **SalesAdmin** users to the **HIPAA\_AppLog\_Audit** Audit.

```
CREATE DATABASE AUDIT SPECIFICATION Sales_Audit_Spec
FOR SERVER AUDIT HIPAA_AppLog_Audit
    ADD (DATABASE_OBJECT_CHANGE_GROUP),
    ADD (INSERT, UPDATE, DELETE
        ON Schema::Sales
        BY SalesUser, SalesAdmin);
```

The **Audit** object provides a manageable auditing framework that makes it easy to define the events that should be logged and the locations where the log should be stored. This addition to SQL Server helps you to implement a comprehensive auditing solution to secure your database and meet regulatory compliance requirements.

---

## DDL Triggers

DDL triggers were introduced in SQL Server 2005. Unlike DML triggers that execute Transact-SQL code when *data* in a table changes, a DDL trigger fires when the *structure* of the table changes. This is a great way to track and audit structural changes to a database schema.

The syntax for these triggers is similar to that of DML triggers. DDL triggers are AFTER triggers that fire in response to DDL language events; they do not fire in response to system-stored procedures that perform DDL-like operations. They are fully transactional, and so you can ROLLBACK a DDL change. You can run either Transact-SQL or CLR code in a DDL trigger. DDL triggers also support the EXECUTE AS clause similar to other modules.

SQL Server provides the information about the trigger event as untyped XML. It is available through a new, XML-emitting built-in function called EVENTDATA(). You can use XQuery expressions to parse the EVENTDATA() XML in order to discover event attributes like schema name, target object name, user name, as well as the entire Transact-SQL DDL statement that caused the trigger to fire in the first place. For examples, see EVENTDATA (Transact-SQL) in SQL Server Books Online.

Database-level DDL triggers fire on DDL language events at the database level and below. Examples are CREATE\_TABLE, ALTER\_USER, and so on. Server-level DDL triggers fire on DDL language events at the server level, for example CREATE\_DATABASE, ALTER\_LOGIN, etc. As an administrative convenience, you can use event groups like DDL\_TABLE\_EVENTS as shorthand to refer to all CREATE\_TABLE, ALTER\_TABLE, and DROP\_TABLE events. The various DDL event groups and event types, and their associated XML EVENTDATA(), are documented in SQL Server Books Online.

Unlike DML trigger names, which are schema-scoped, DDL trigger names are database scoped or server-scoped.

Use this new catalog view to discover trigger metadata for DML triggers and database-level DDL triggers:

```
SELECT * FROM sys.triggers ;  
GO
```

If the **parent\_class\_desc** column has a value of 'DATABASE,' it is a DDL trigger and the name is scoped by the database itself. The body of a Transact-SQL trigger is found in the **sys.sql\_modules** catalog view, and you can JOIN it to **sys.triggers** on the **object\_id** column. The metadata about a CLR trigger is found in the **sys.assembly\_modules** catalog view, and again, you can JOIN to **sys.triggers** on the **object\_id** column.

---

Use this catalog view to discover metadata for server-scoped DDL triggers:

```
SELECT * FROM sys.server_triggers ;  
GO
```

The body of a Transact-SQL server-level trigger is found in the **sys.server\_sql\_modules** catalog view, and you can JOIN it to **sys.server\_triggers** on the **object\_id** column. The metadata about a CLR server-level trigger is found in the **sys.server\_assembly\_modules** catalog view, and again, you can JOIN to **sys.server\_triggers** on the **object\_id** column.

You can use DDL triggers to capture and audit DDL activity in a database. Create an audit table with an untyped XML column. Create an EXECUTE AS SELF DDL trigger for the DDL events or event groups you are interested in. The body of the DDL trigger can simply INSERT the EVENTDATA() XML into the audit table.

Another interesting use of DDL triggers is to fire on the CREATE\_USER event, and then add code to automate permissions management. For example, you want all database users to get a GRANT EXECUTE on procedures P1, P2, and P3. The DDL trigger can extract the user name from the EVENTDATA() XML, dynamically formulate a statement like 'GRANT EXECUTE ON P1 TO someuser', and then EXEC() it.

## Conclusion

SQL Server 2008 provides rich security features to protect data and network resources. It is much easier to install securely, since all but the most essential features are either not installed by default or disabled if they are installed. SQL Server provides plenty of tools to configure the server, particularly for SQL Server Surface Area Configuration. Its authentication features are stronger because SQL Server more closely integrates with Windows authentication and protects against weak or ancient passwords. Granting and controlling what a user can do when authenticated is far more flexible with granular permissions, SQL Server Agent proxies, and execution context. Even metadata is more secure, since the system metadata views return information only about objects that the user has permission to use in some way. At the database level, encryption provides a final layer of defense while the separation of users and schemas makes managing users easier.

### For more information:

Microsoft SQL Server 2008

<http://www.microsoft.com/sqlserver/2008/en/us/default.aspx>

## Please give us your feedback:

Did this paper help you? 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 giving it a high rating because it has good examples, excellent screenshots, clear writing, or another reason?
- Are you giving it a low rating because it has poor examples, fuzzy screenshots, unclear writing?

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

[Send feedback.](#)

---

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 OR IMPLIED, 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.

© 2008 Microsoft Corporation. All rights reserved.

Microsoft, PowerShell, SharePoint, SQL Server, Visual Basic, Visual C#, Visual Studio, Windows, Windows Server, and the Server Identity Logo are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.