.NET Framework 4

**Overview of SQL Server Security (ADO.NET)**

A defense-in-depth strategy, with overlapping layers of security, is the best way to counter security threats. SQL Server provides a security architecture that is designed to allow database administrators and developers to create secure database applications and counter threats. Each version of SQL Server has improved on previous versions of SQL Server with the introduction of new features and functionality. However, security does not ship in the box. Each application is unique in its security requirements. Developers need to understand which combination of features and functionality are most appropriate to counter known threats, and to anticipate threats that may arise in the future.

SQL Server 2005 introduced many improvements to the SQL Server 2000 security framework, but the basic security architecture remains unchanged. A SQL Server instance contains a hierarchical collection of entities, starting with the server. Each server contains multiple databases, and each database contains a collection of securable objects. Every SQL Server securable has associated *permissions* that can be granted to a *principal*, which is an individual, group or process granted access to SQL Server. The SQL Server security framework manages access to securable entities through *authentication* and *authorization*.

* Authentication is the process of logging on to SQL Server by which a principal requests access by submitting credentials that the server evaluates. Authentication establishes the identity of the user or process being authenticated.
* Authorization is the process of determining which securable resources a principal can access, and which operations are allowed for those resources.

The topics in this section cover SQL Server security fundamentals, providing links to the complete documentation in the relevant version of SQL Server Books Online.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifIn This Section

[Authentication in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669066.aspx)

Describes logins and authentication in SQL Server and provides links to additional resources.

[Server and Database Roles in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669065.aspx)

Describes fixed server and database roles, custom database roles, and built-in accounts and provides links to additional resources.

[Ownership and User-Schema Separation in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669061.aspx)

Describes object ownership and user-schema separation and provides links to additional resources.

[Authorization and Permissions in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669084.aspx)

Describes granting permissions using the principle of least privilege and provides links to additional resources.

[Data Encryption in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669072.aspx)

Describes data encryption options in SQL Server and provides links to additional resources.

[CLR Integration Security in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669064.aspx)

Provides links to CLR integration security resources.

.NET Framework 4

**Authentication in SQL Server (ADO.NET)**

SQL Server supports two authentication modes, Windows authentication mode and mixed mode.

* Windows authentication is the default, and is often referred to as integrated security because this SQL Server security model is tightly integrated with Windows. Specific Windows user and group accounts are trusted to log in to SQL Server. Windows users who have already been authenticated do not have to present additional credentials.
* Mixed mode supports authentication both by Windows and by SQL Server. User name and password pairs are maintained within SQL Server.

|  |
| --- |
| **Description: Security noteSecurity Note** |
| We recommend using Windows authentication wherever possible. Windows authentication uses a series of encrypted messages to authenticate users in SQL Server. When SQL Server logins are used, SQL Server login names and passwords are passed across the network, which makes them less secure. |

With Windows authentication, users are already logged onto Windows and do not have to log on separately to SQL Server. The following **SqlConnection.ConnectionString** specifies Windows authentication without requiring the a user name or password.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl02_code');" \o "Copy Code)

"Server=MSSQL1;Database=AdventureWorks;Integrated Security=true;

|  |
| --- |
| **Description: NoteNote** |
| Logins are distinct from database users. You must map logins or Windows groups to database users or roles in a separate operation. You then grant permissions to users or roles to access database objects. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifAuthentication Scenarios

Windows authentication is usually the best choice in the following situations:

* There is a domain controller.
* The application and the database are on the same computer.
* You are using an instance of SQL Server Express.

SQL Server logins are often used in the following situations:

* If you have a workgroup.
* Users connect from different, non-trusted domains.
* Internet applications, such as ASP.NET.

|  |
| --- |
| **Description: NoteNote** |
| Specifying Windows authentication does not disable SQL Server logins. Use the ALTER LOGIN DISABLE Transact-SQL statement to disable highly-privileged SQL Server logins. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifLogin Types

SQL Server 2000 supports three types of logins:

* A local Windows user account or trusted domain account. SQL Server relies on Windows to authenticate the Windows user accounts.
* Windows group. Granting access to a Windows group grants access to all Windows user logins that are members of the group.
* SQL Server login. SQL Server stores both the username and a hash of the password in the master database, by using internal authentication methods to verify login attempts.

|  |
| --- |
| **Description: NoteNote** |
| SQL Server 2005 added logins created from certificates or asymmetric keys that are used only for code signing. They cannot be used to connect to SQL Server. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifMixed Mode Authentication

If you must use mixed mode authentication, you must create SQL Server logins, which are stored in SQL Server. You then have to supply the SQL Server user name and password at run time.

|  |
| --- |
| **Description: Security noteSecurity Note** |
| SQL Server installs with a SQL Server login named **sa** (an abbreviation of "system administrator"). Assign a strong password to the **sa** login and do not use the **sa** login in your application. The **sa** login maps to the **sysadmin** fixed server role, which has irrevocable administrative credentials on the whole server. There are no limits to the potential damage if an attacker gains access as a system administrator. All members of the Windows **BUILTIN\Administrators** group (the local administrator's group) are members of the **sysadmin** role by default, but can be removed from that role. |

SQL Server 2005 introduces Windows password policy mechanisms for SQL Server logins when it is running on Windows Server 2003 or later versions. Password complexity policies are designed to deter brute force attacks by increasing the number of possible passwords. SQL Server 2005 can apply the same complexity and expiration policies used in Windows Server 2003 to passwords used inside SQL Server.

|  |
| --- |
| **Description: Security noteSecurity Note** |
| Concatenating connection strings from user input can leave you vulnerable to a connection string injection attack. Use the [SqlConnectionStringBuilder](http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnectionstringbuilder.aspx) to create syntactically valid connection strings at run time. For more information, see [Connection String Builders (ADO.NET)](http://msdn.microsoft.com/en-us/library/ms254947.aspx). |

.NET Framework 4

**Server and Database Roles in SQL Server (ADO.NET)**

All versions of SQL Server use role-based security, which allows you to assign permissions to a role, or group of users, instead of to individual users. Fixed server and fixed database roles have a fixed set of permissions assigned to them.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFixed Server Roles

Fixed server roles have a fixed set of permissions and server-wide scope. They are intended for use in administering SQL Server and the permissions assigned to them cannot be changed. Logins can be assigned to fixed server roles without having a user account in a database.

|  |
| --- |
| **Description: Security noteSecurity Note** |
| The **sysadmin** fixed server role encompasses all other roles and has unlimited scope. Do not add principals to this role unless they are highly trusted. **sysadmin** role members have irrevocable administrative privileges on all server databases and resources. |

Be selective when you add users to fixed server roles. For example, the **bulkadmin** role allows users to insert the contents of any local file into a table, which could jeopardize data integrity. See SQL Server 2005 Books Online for the complete list of fixed server roles and permissions. The fixed server roles for SQL Server 2000 have the same names and permission sets as appropriate for SQL Server 2000.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFixed Database Roles

Fixed database roles have a pre-defined set of permissions that are designed to allow you to easily manage groups of permissions. Members of the **db\_owner** role can perform all configuration and maintenance activities on the database.

For more information about SQL Server predefined roles, see the following resources.

|  |  |
| --- | --- |
| **Resource** | **Description** |
| [Server-Level Roles](http://go.microsoft.com/fwlink/?LinkId=98372) and [Permissions of Fixed Server Roles](http://go.microsoft.com/fwlink/?LinkId=98373) in SQL Server 2005 Books Online | Describes fixed server roles and the permissions associated with them in SQL Server 2005. |
| [Database-Level Roles](http://go.microsoft.com/fwlink/?LinkId=98374) and [Permissions of Fixed Database Roles](http://go.microsoft.com/fwlink/?LinkId=98589) in SQL Server 2005 Books Online | Describes fixed database roles and the permissions associated with them |
| [Adding a Member to a Predefined Role](http://go.microsoft.com/fwlink/?LinkId=98375) in SQL Server 2000 Books Online | Describes the permissions associated with fixed server and fixed database roles and demonstrates how to add members to the roles. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifDatabase Roles and Users

Logins must be mapped to database user accounts in order to work with database objects. Database users can then be added to database roles, inheriting any permission sets associated with those roles. Since SQL Server 2005, all permissions can be granted.

You must also consider the **public** role, the **dbo** user account, and the **guest** account when you design security for your application.

**The public Role**

The **public** role is contained in every database, which includes system databases. It cannot be dropped and you cannot add or remove users from it. Permissions granted to the **public** role are inherited by all other users and roles because they belong to the **public** role by default. Grant **public** only the permissions you want all users to have.

**The dbo User Account**

The **dbo**, or database owner, is a user account that has implied permissions to perform all activities in the database. Members of the **sysadmin** fixed server role are automatically mapped to **dbo**.

|  |
| --- |
| **Description: NoteNote** |
| Beginning with SQL Server 2005, **dbo** is also the name of a schema, as discussed in [Ownership and User-Schema Separation in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669061.aspx). |

The **dbo** user account is frequently confused with the **db\_owner** fixed database role. The scope of **db\_owner** is a database; the scope of **sysadmin** is the whole server. Membership in the **db\_owner** role does not confer **dbo** user privileges.

**The guest User Account**

After a user has been authenticated and allowed to log in to an instance of SQL Server, a separate user account must exist in each database the user has to access. Requiring a user account in each database prevents users from connecting to an instance of SQL Server and accessing all the databases on a server. The existence of a **guest** user account in the database circumvents this requirement by allowing a login without a database user account to access a database.

The **guest** account is a built-in account in all versions of SQL Server. By default, it is disabled in new databases. If it is enabled, you can disable it by revoking its CONNECT permission by executing the Transact-SQL REVOKE CONNECT FROM GUEST statement. In SQL Server 2000 you can disable it by executing the Transact-SQL sp\_dropuser or sp\_revokedbaccess system stored procedures.

|  |
| --- |
| **Description: Security noteSecurity Note** |
| Avoid using the **guest** account; all logins without their own database permissions obtain the database permissions granted to this account. If you must use the **guest** account, grant it minimum permissions. |

.NET Framework 4

**Ownership and User-Schema Separation in SQL Server (ADO.NET)**

A core concept of SQL Server security is that owners of objects have irrevocable permissions to administer them. You cannot remove privileges from an object owner, and you cannot drop users from a database if they own objects in it. In SQL 2000, if a user creates an object without specifying the owner, the user becomes the owner of the object. An object owner has irrevocable permissions to administer the object, that caused problems if multiple users own objects. Having all database objects owned by a single owner simplifies managing permissions in SQL Server 2000 by taking advantage of ownership chaining, as discussed in [Authorization and Permissions in SQL Server (ADO.NET)](http://msdn.microsoft.com/en-us/library/bb669084.aspx).

|  |
| --- |
| **Description: NoteNote** |
| Members of the sysadmin fixed server role and the db\_owner fixed database role also have irrevocable ownership permissions on all objects in a database. |

Versions of SQL Server prior to SQL Server 2005 use the following four-part naming syntax for referring to objects.

Server.Database.ObjectOwner.DatabaseObject

If database users Bob and Sue each create Table1, you need to use the fully-qualified name. If a system administrator or database owner creates an object, it is owned by the dbo user account, not by an individual user.

SELECT col1 FROM Bob.Table1

SELECT col1 FROM Sue.Table1

SELECT col1 FROM dbo.Table1

|  |
| --- |
| **Description: NoteNote** |
| It is not necessary to use the ObjectOwner.DatabaseObject syntax when referring to an object owned by dbo, although it is more efficient to do so. If the owner name is not supplied, SQL Server 2000 checks first to see if the object exists under the name of the current user, then it checks to see if the object is owned by dbo. Supplying the two-part name saves an extra step. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifUser-Schema Separation

SQL Server 2005 introduced user-schema separation, which allows for more flexibility in managing database object permissions. A schema is a named container for database objects, which allows you to group objects into separate namespaces. For example, the AdventureWorks sample database contains schemas for Production, Sales, and HumanResources.

The four-part naming syntax for referring to objects specifies the schema name.

Server.Database.DatabaseSchema.DatabaseObject

### Schema Owners and Permissions

Schemas can be owned by any database principal, and a single principal can own multiple schemas. You can apply security rules to a schema, which are inherited by all objects in the schema. Once you set up access permissions for a schema, those permissions are automatically applied as new objects are added to the schema. Users can be assigned a default schema, and multiple database users can share the same schema.

By default, when developers create objects in a schema, the objects are owned by the security principal that owns the schema, not the developer. Object ownership can be transferred with ALTER AUTHORIZATION Transact-SQL statement. A schema can also contain objects that are owned by different users and have more granular permissions than those assigned to the schema, although this is not recommended because it adds complexity to managing permissions. Objects can be moved between schemas, and schema ownership can be transferred between principals. Database users can be dropped without affecting schemas.

### Built-In Schemas

SQL Server ships with ten pre-defined schemas that have the same names as the built-in database users and roles. These exist mainly for backward compatibility. You can drop the schemas that have the same names as the fixed database roles if you do not need them. You cannot drop the following schemas:

* dbo
* guest
* sys
* INFORMATION\_SCHEMA

If you drop them from the model database, they will not appear in new databases.

|  |
| --- |
| **Description: NoteNote** |
| The sys and INFORMATION\_SCHEMA schemas are reserved for system objects. You cannot create objects in these schemas and you cannot drop them. |

#### The dbo Schema

The dbo schema is the default schema for a newly created database. The dbo schema is owned by the dbo user account. By default, users created with the CREATE USER Transact-SQL command have dbo as their default schema.

Users who are assigned the dbo schema do not inherit the permissions of the dbo user account. No permissions are inherited from a schema by users; schema permissions are inherited by the database objects contained in the schema.

|  |
| --- |
| **Description: NoteNote** |
| When database objects are referenced by using a one-part name in SQL Server 2005, SQL Server first looks in the user's default schema. If the object is not found there, SQL Server looks next in the dbo schema. If the object is not in the dbo schema, an error is returned. |

.NET Framework 4

**Authorization and Permissions in SQL Server (ADO.NET)**

When you create database objects, you must explicitly grant permissions to make them accessible to users. Every securable object has permissions that can be granted to a principal using permission statements.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifThe Principle of Least Privilege

Developing an application using a least-privileged user account (LUA) approach is an important part of a defensive, in-depth strategy for countering security threats. The LUA approach ensures that users follow the principle of least privilege and always log on with limited user accounts. Administrative tasks are broken out using fixed server roles, and the use of the **sysadmin** fixed server role is severely restricted.

Always follow the principle of least privilege when granting permissions to database users. Grant the minimum permissions necessary to a user or role to accomplish a given task.

|  |
| --- |
| **Description: Security noteSecurity Note** |
| Developing and testing an application using the LUA approach adds a degree of difficulty to the development process. It is easier to create objects and write code while logged on as a system administrator or database owner than it is using a LUA account. However, developing applications using a highly privileged account can obfuscate the impact of reduced functionality when least privileged users attempt to run an application that requires elevated permissions in order to function correctly. Granting excessive permissions to users in order to reacquire lost functionality can leave your application vulnerable to attack. Designing, developing and testing your application logged on with a LUA account enforces a disciplined approach to security planning that eliminates unpleasant surprises and the temptation to grant elevated privileges as a quick fix. You can use a SQL Server login for testing even if your application is intended to deploy using Windows authentication. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifRole-Based Permissions

Granting permissions to roles rather than to users simplifies security administration. Permission sets that are assigned to roles are inherited by all members of the role. It is easier to add or remove users from a role than it is to recreate separate permission sets for individual users. Roles can be nested; however, too many levels of nesting can degrade performance. You can also add users to fixed database roles to simplify assigning permissions.

Starting with SQL Server 2005, you can grant permissions at the schema level. Users automatically inherit permissions on all new objects created in the schema; you do not need to grant permissions as new objects are created.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifPermissions Through Procedural Code

Encapsulating data access through modules such as stored procedures and user-defined functions provides an additional layer of protection around your application. You can prevent users from directly interacting with database objects by granting permissions only to stored procedures or functions while denying permissions to underlying objects such as tables. SQL Server achieves this by ownership chaining.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifPermission Statements

The three Transact-SQL permission statements are described in the following table.

|  |  |
| --- | --- |
| **Permission Statement** | **Description** |
| GRANT | Grants a permission. |
| REVOKE | Revokes a permission. This is the default state of a new object. A permission revoked from a user or role can still be inherited from other groups or roles to which the principal is assigned. |
| DENY | DENY revokes a permission so that it cannot be inherited. DENY takes precedence over all permissions, except DENY does not apply to object owners or members of **sysadmin**. If you DENY permissions on an object to the **public** role it is denied to all users and roles except for object owners and **sysadmin** members. |

* The GRANT statement can assign permissions to a group or role that can be inherited by database users. However, the DENY statement takes precedence over all other permission statements. Therefore, a user who has been denied a permission cannot inherit it from another role.

|  |
| --- |
| **Description: NoteNote** |
| Members of the **sysadmin** fixed server role and object owners cannot be denied permissions. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifOwnership Chains

SQL Server ensures that only principals that have been granted permission can access objects. When multiple database objects access each other, the sequence is known as a chain. When SQL Server is traversing the links in the chain, it evaluates permissions differently than it would if it were accessing each item separately. When an object is accessed through a chain, SQL Server first compares the object's owner to the owner of the calling object (the previous link in the chain). If both objects have the same owner, permissions on the referenced object are not checked. Whenever an object accesses another object that has a different owner, the ownership chain is broken and SQL Server must check the caller's security context.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifProcedural Code and Ownership Chaining

Suppose that a user is granted execute permissions on a stored procedure that selects data from a table. If the stored procedure and the table have the same owner, the user doesn't need to be granted any permissions on the table and can even be denied permissions. However, if the stored procedure and the table have different owners, SQL Server must check the user's permissions on the table before allowing access to the data.

|  |
| --- |
| **Description: NoteNote** |
| Ownership chaining does not apply in the case of dynamic SQL statements. To call a procedure that executes an SQL statement, the caller must be granted permissions on the underlying tables, leaving your application vulnerable to SQL Injection attack. SQL Server 2005 introduces new mechanisms, such as impersonation and signing modules with certificates, that do not require granting permissions on the underlying tables. These can also be used with CLR stored procedures. |

.NET Framework 4

**Data Encryption in SQL Server (ADO.NET)**

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

The fastest mode of encryption supported by the encryption functions is symmetric key encryption. This mode is suitable for handling large volumes of data. The symmetric keys can be encrypted by certificates, passwords or other symmetric keys.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifKeys and Algorithms

SQL Server 2005 supports several symmetric key encryption algorithms, including DES, Triple DES, RC2, RC4, 128-bit RC4, DESX, 128-bit AES, 192-bit AES, and 256-bit AES. The algorithms are implemented using the Windows Crypto API.

Within the scope of a database connection, SQL Server 2005 can maintain multiple open symmetric keys. An open key is retrieved from the store and is available for decrypting data. When a piece of data is decrypted, there is no need to specify the symmetric key to use. Each encrypted value contains the key identifier (key GUID) of the key used to encrypt it. The engine matches the encrypted byte stream to an open symmetric key, if the correct key has been decrypted and is open. This key is then used to perform decryption and return the data. If the correct key is not open, NULL is returned.

For an example that shows how to work with encrypted data in a database:

**Encrypt a Column of Data**

This topic describes how to encrypt a column of data by using symmetric encryption in SQL Server 2012 using Transact-SQL.

In This Topic

* Before you begin:

[Security](http://msdn.microsoft.com/en-us/library/ms179331.aspx#Security)

* [To encrypt a column of data, using Transact-SQL](http://msdn.microsoft.com/en-us/library/ms179331.aspx#TsqlProcedure)

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifBefore You Begin

### Security

#### Permissions

The following permissions are necessary to perform the steps below:

* CONTROL permission on the database.
* CREATE CERTIFICATE permission on the database. Only Windows logins, SQL Server logins, and application roles can own certificates. Groups and roles cannot own certificates.
* ALTER permission on the table.
* Some permission on the key and must not have been denied VIEW DEFINITION permission.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifUsing Transact-SQL

### To encrypt a column of data using a simple symmetric encryption

1. In Object Explorer, connect to an instance of Database Engine.
2. On the Standard bar, click New Query.
3. Copy and paste the following example into the query window and click Execute.

USE AdventureWorks2012;

--If there is no master key, create one now.

IF NOT EXISTS

(SELECT \* FROM sys.symmetric\_keys WHERE symmetric\_key\_id = 101)

CREATE MASTER KEY ENCRYPTION BY

PASSWORD = '23987hxJKL95QYV4369#ghf0%lekjg5k3fd117r$$#1946kcj$n44ncjhdlj'

GO

CREATE CERTIFICATE Sales09

WITH SUBJECT = 'Customer Credit Card Numbers';

GO

CREATE SYMMETRIC KEY CreditCards\_Key11

WITH ALGORITHM = AES\_256

ENCRYPTION BY CERTIFICATE Sales09;

GO

-- Create a column in which to store the encrypted data.

ALTER TABLE Sales.CreditCard

ADD CardNumber\_Encrypted varbinary(128);

GO

-- Open the symmetric key with which to encrypt the data.

OPEN SYMMETRIC KEY CreditCards\_Key11

DECRYPTION BY CERTIFICATE Sales09;

-- Encrypt the value in column CardNumber using the

-- symmetric key CreditCards\_Key11.

-- Save the result in column CardNumber\_Encrypted.

UPDATE Sales.CreditCard

SET CardNumber\_Encrypted = EncryptByKey(Key\_GUID('CreditCards\_Key11')

, CardNumber, 1, HashBytes('SHA1', CONVERT( varbinary

, CreditCardID)));

GO

-- Verify the encryption.

-- First, open the symmetric key with which to decrypt the data.

OPEN SYMMETRIC KEY CreditCards\_Key11

DECRYPTION BY CERTIFICATE Sales09;

GO

-- Now list the original card number, the encrypted card number,

-- and the decrypted ciphertext. If the decryption worked,

-- the original number will match the decrypted number.

SELECT CardNumber, CardNumber\_Encrypted

AS 'Encrypted card number', CONVERT(nvarchar,

DecryptByKey(CardNumber\_Encrypted, 1 ,

HashBytes('SHA1', CONVERT(varbinary, CreditCardID))))

AS 'Decrypted card number' FROM Sales.CreditCard;

GO

### To encrypt a column of data using symmetric encryption that includes an authenticator

1. In Object Explorer, connect to an instance of Database Engine.
2. On the Standard bar, click New Query.
3. Copy and paste the following example into the query window and click Execute.

USE AdventureWorks2012;

GO

--If there is no master key, create one now.

IF NOT EXISTS

(SELECT \* FROM sys.symmetric\_keys WHERE symmetric\_key\_id = 101)

CREATE MASTER KEY ENCRYPTION BY

PASSWORD = '23987hxJKL969#ghf0%94467GRkjg5k3fd117r$$#1946kcj$n44nhdlj'

GO

CREATE CERTIFICATE HumanResources037

WITH SUBJECT = 'Employee Social Security Numbers';

GO

CREATE SYMMETRIC KEY SSN\_Key\_01

WITH ALGORITHM = AES\_256

ENCRYPTION BY CERTIFICATE HumanResources037;

GO

USE [AdventureWorks2012];

GO

-- Create a column in which to store the encrypted data.

ALTER TABLE HumanResources.Employee

ADD EncryptedNationalIDNumber varbinary(128);

GO

-- Open the symmetric key with which to encrypt the data.

OPEN SYMMETRIC KEY SSN\_Key\_01

DECRYPTION BY CERTIFICATE HumanResources037;

-- Encrypt the value in column NationalIDNumber with symmetric

-- key SSN\_Key\_01. Save the result in column EncryptedNationalIDNumber.

UPDATE HumanResources.Employee

SET EncryptedNationalIDNumber = EncryptByKey(Key\_GUID('SSN\_Key\_01'), NationalIDNumber);

GO

-- Verify the encryption.

-- First, open the symmetric key with which to decrypt the data.

OPEN SYMMETRIC KEY SSN\_Key\_01

DECRYPTION BY CERTIFICATE HumanResources037;

GO

-- Now list the original ID, the encrypted ID, and the

-- decrypted ciphertext. If the decryption worked, the original

-- and the decrypted ID will match.

SELECT NationalIDNumber, EncryptedNationalIDNumber

AS 'Encrypted ID Number',

CONVERT(nvarchar, DecryptByKey(EncryptedNationalIDNumber))

AS 'Decrypted ID Number'

FROM HumanResources.Employee;

GO

.NET Framework 4

**CLR Integration Security in SQL Server (ADO.NET)**

Microsoft SQL Server 2005 introduces the integration of the common language runtime (CLR) component of the .NET Framework. CLR integration allows you to write stored procedures, triggers, user-defined types, user-defined functions, user-defined aggregates, and streaming table-valued functions, using any .NET Framework language, such as Microsoft Visual Basic .NET or Microsoft Visual C#.

The CLR supports a security model called code access security (CAS) for managed code. In this model, permissions are granted to assemblies based on evidence supplied by the code in metadata. SQL Server integrates the user-based security model of SQL Server with the code access-based security model of the CLR.