6

SQL Server 2008 Security

Security is often one of the most challenging aspects of designing and managing a database system. As a DBA, you want your servers to be as secure as possible without having to invest an inordinate amount of money or sacrifice user functionality. Unfortunately, many administrators and application developers are often skeptical about the benefits of security, believing that they are somehow immune to the myriad of threats that are out there. In reality, as long as users have access to data, there is a risk of a security breach. So what do you do? Take the SQL Server offline, put it in a locked room that only you have access to, and require that all database requests be processed manually through you?

Security isn't about guaranteeing a completely attack-proof system. It's about mitigating and responding to risk. It's about ensuring that you take the necessary steps to minimize the scope of the attack. Remember that simply giving users access to the database through the network will introduce an element of risk. This chapter takes a look at SQL Security from the outside in. You will learn about the different types of accounts and principals that are available. You will see how to control access to database objects, and how to encrypt and protect your data. This chapter also includes some guidelines for providing a secure solution for deploying and managing your SQL Server.

Because SQL Server 2008 is designed to work with Windows Server 2008, some of the examples in this chapter may behave a little differently in other operating systems, such as Windows Vista, Windows XP, or Windows Server 2003. All examples in this chapter use Windows Server 2008 as the baseline. Also, many of the examples used in this chapter refer to the server AughtEight, which I configured in Chapter 2. Remember to replace *AughtEight* with your own server name.

SQL Server Authentication Modes

Microsoft SQL Server 2008 offers two options for authenticating users. The default mode is *Windows Authentication mode*, which offers a high level of security by using the operating system's authentication mechanism to authenticate credentials that will need access to the server. The other, *SQL Server and Windows Authentication mode*, offers the ability to allow both Windows-based and SQL-based authentications. For this reason, it is also sometimes referred to as *Mixed mode*. Although Windows Authentication mode typically provides better security than SQL Server and Windows Authentication mode, the design of your application may require SQL-based logins.

Windows Authentication mode allows you to use existing accounts stored in the local computer's Security Accounts Manager (SAM) database, or, if the server is a member of an Active Directory domain, accounts in the Microsoft Windows Active Directory database. The benefits of using the Windows Authentication mode include reducing the administrative overhead for your SQL or database administrators by allowing them to use accounts that already exist and the ability to use stronger authentication protocols, such as Kerberos or Windows NT LAN Manager (NTLM).

In Windows Authentication mode, SQL does not store or need to access password information for authentication. The Windows Authentication Provider will be responsible for validating the authenticity of the user.

Mixed mode authentication allows you to create logins that are unique to the SQL Server and do not have a corresponding Windows or Active Directory account. This can be helpful for applications that require users who are not part of your enterprise to be able to authenticate and gain access to securable objects in your database. When SQL logins are used, the SQL Server stores username and password information in the `master` database, and the SQL Server is responsible for authenticating these credentials.

When deciding on the authentication method, it is important to identify how users will be connecting to the database. If the SQL Server and your database users are all members of the same Active Directory forest, or even different forests that share a trust, using Windows Authentication can simplify the process of creating and managing logins. However, if your SQL Server is not in an Active Directory domain or your database users are not internal to your organization, consider the use of SQL-based logins to create a clear distinction between security contexts.

In Chapter 2, you learned how to install Microsoft SQL Server 2008, and you selected which authentication mode to use. If you wish to change the authentication mode after the installation, be aware that this will require you to restart the SQL Server service.

Changing the Authentication Mode from Management Studio

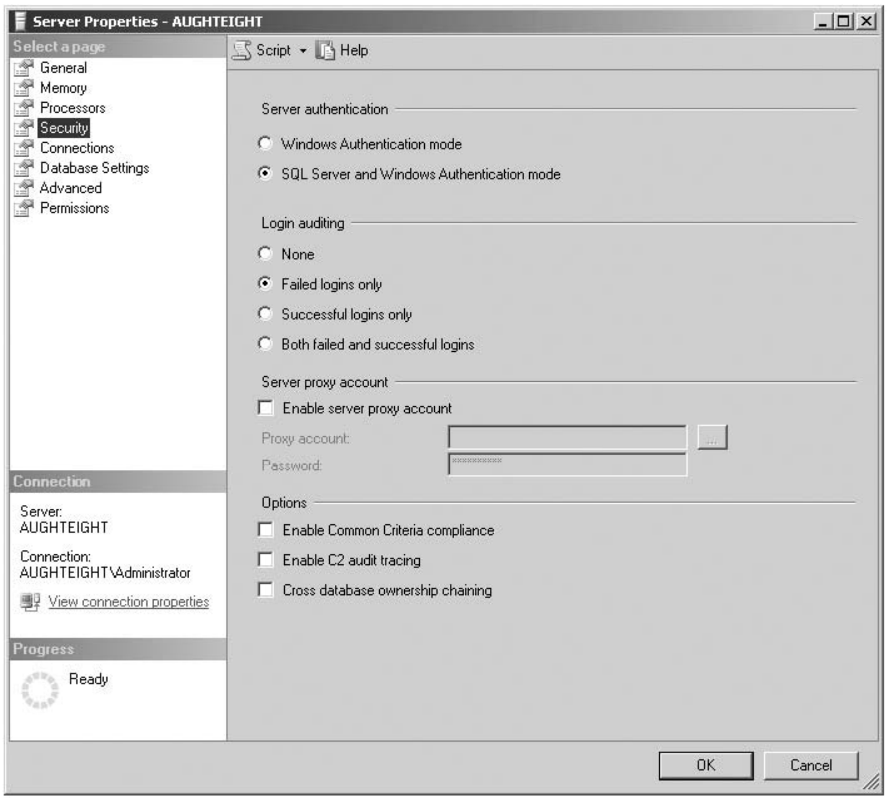
To change the authentication mode from Management Studio, follow these steps:

1. Launch SQL Server Management Studio.
2. In Object Explorer, select your server.
3. Right-click on your server and select Properties.
4. Under the ''Select a page'' pane, select Security.
5. Under the heading ''Server authentication,'' select or review the appropriate authentication mode (Figure 6-1).

Using the `xp_instance_regwrite` Extended Stored Procedure

You can also change the authentication mode using the `xp_instance_regwrite` extended stored procedure, as long as you have administrative permissions on the local server. The following example shows you how to change the authentication mode to SQL Server and Windows Authentication mode:

```
USE master
EXEC xp_instance_regwrite N'HKEY_LOCAL_MACHINE',
N'Software\Microsoft\MSSQLServer\MSSQLServer', N'LoginMode', REG_DWORD, 2
```



Server authentication
- Windows Authentication mode
- [✓ SQL Server and Windows Authentication mode]

Login auditing
- None
- [✓ Failed logins only]
- Successful logins only
- Both failed and successful logins

Server proxy account
- Enable server proxy account

Proxy account:
Password: **********

Options
- Enable Common Criteria compliance
- Enable C2 audit tracing
- Cross database ownership chaining

Server:
AUGHTEIGHT

Connection:
AUGHTEIGHT\Administrator

Ready

Figure 6-1: Server Properties screen.

You can also change the authentication mode to Windows Authentication mode by changing the `DWORD` value to 1, as shown in this example:

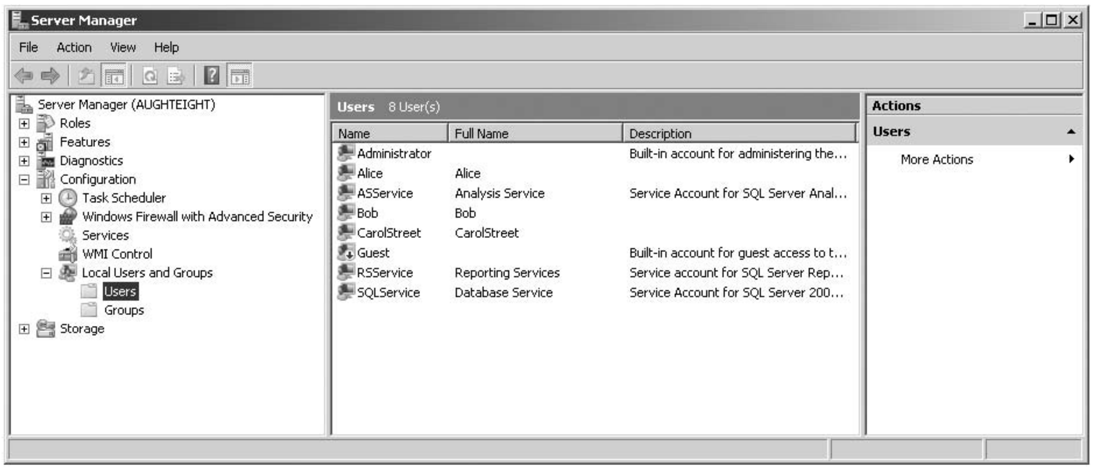
```
USE master
EXEC xp_instance_regwrite N'HKEY_LOCAL_MACHINE',
N'Software\Microsoft\MSSQLServer\MSSQLServer', N'LoginMode', REG_DWORD, 1
```

During the installation of SQL Server, the `sa` account is disabled by default. If you are changing the authentication mode from Windows Authentication mode to SQL Server and Windows Authentication mode, the account remains disabled with the password you specified during the Installation Wizard. I recommend against using the `sa` account in a production environment, especially when multiple people have administrative access to the SQL Server because of the lack of accountability. When multiple people can log in as the `sa` account, you lose the ability to associate an auditable action with a specific person.

Principals

The term *principal* is used to describe individuals, groups, and processes that will interact with the SQL Server. The resources available to a principal are dependent on where the principal resides. Microsoft SQL Server supports several different types of principals defined at three different levels: the Windows level, the SQL Server level, and the database level. Each type of principal is identified here, and the way they are used. To prepare for some of the exercises in this chapter, you will want to create some local Windows accounts as follows:

1. From the Start Menu, right-click My Computer and select Manage.
2. In the Server Manager window, expand Configuration, then ''Local Users and Groups'' (see Figure 6-2).



Name	Full Name	Description
Administrator		Built-in account for administering the...
Alice	Alice	
ASService	Analysis Service	Service Account for SQL Server Anal...
Bob	Bob	
CarolStreet	CarolStreet	
Guest		Built-in account for guest access to t...
RSService	Reporting Services	Service account for SQL Server Rep...
SQLService	Database Service	Service Account for SQL Server 200...

Figure 6-2: Server Management screen.

3. Right-click on the Users folder and select ''New User.''
4. In the User Name box, enter **Bob**.
5. In the Password and Confirm Password boxes, enter **P@ssw0rd**.
6. Clear the check next to the ''User must change password and next login'' box.
7. Click Create.
8. In the User Name box, enter **CarolStreet**.
9. In the Password and Confirm Password boxes, enter **P@ssw0rd**.
10. Clear the check next to the ''User must change password and next login'' box.
11. Click Create.
12. In the User Name box, enter **Alice**.
13. In the Password and Confirm Password boxes, enter **P@ssw0rd**.
14. Clear the check next to the ''User must change password and next login'' box.
15. Click Create.
16. Click Close.

17. Right-click on the Groups folder and select "New Group."
18. In the Group Name Box, enter **G NorthWest Sales**.
19. Click Create.
20. Click Close.
21. Close the Server Manager window.

Logins

Microsoft SQL Server 2008 offers two kinds of logins for authentication. *Windows logins* are associated with user or group accounts stored in the Active Directory or the local Security Accounts Manager (SAM) database. *SQL logins* are used to represent an individual or entity that does not have a Windows account and, therefore, must rely on the SQL Server for storage and management of account information.

Windows logins, whether they represent an individual or a group, are bound by the password policy of either the domain or the SAM in which the account exists. When a login is created for a Windows user or group, no password information is stored in the SQL Server. The password for a Windows login is stored as `NULL`, but even if this field is populated with a value, the value is ignored. Windows logins are also authenticated prior to connecting to the SQL Server. This means that Active Directory or the operating system will have already verified the principal's identity.

When a Windows login is created for a group, all members of that group have the ability to authenticate against the SQL Server without having to create separate logins for each user.

SQL Server logins, however, must authenticate against the SQL Server. This makes the SQL Server responsible for verifying the user's identity. SQL stores the login and a hash of the login's password information in the `master` database. It is important that passwords for SQL logins adhere to security best practices, such as enabling complexity requirements, prohibiting non-expiring passwords, and requiring that passwords be changed regularly. In fact, options in Microsoft SQL Server 2008 allow you to enforce requirements for password complexity and expiration for SQL logins based on your Windows or Active Directory policies. Complex passwords are typically defined as having a combination of at least three of the following four criteria:

- Uppercase alpha characters
- Lowercase alpha characters
- Non-negative integers (0–9)
- Special characters ($, %, *, &)

If the SQL Server is a member of an Active Directory domain, the password policy is usually defined in a Group Policy object linked to the domain. For SQL logins, or logins based on a local Windows account, this may be superseded by a Group Policy object linked to an Organizational Unit. If the SQL Server is not a member of an Active Directory domain, the password policy is defined in the Local Group Policy object or the Local Security Policy (which is a subset of the local GPO).

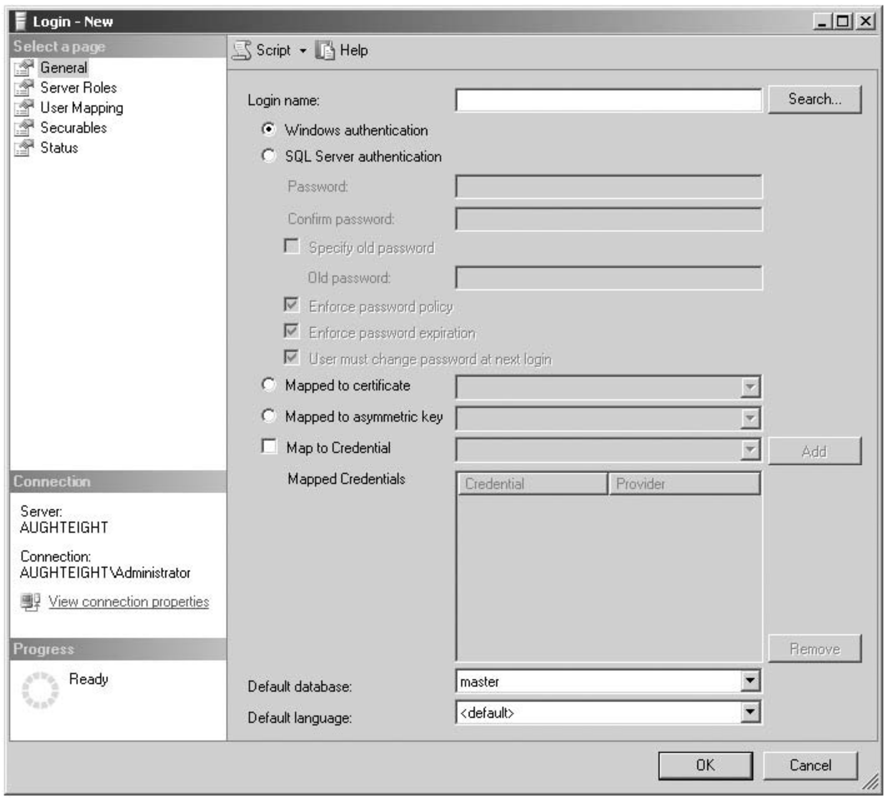
Unlike previous versions of SQL, SQL Server 2008 does not automatically create logins for the [BUILTIN\Administrators] group, which would allow anyone with local administrative rights on the server to log in to the SQL Server. Instead, administrators must be added during the user-provisioning step in the Installation Wizard (see Chapter 2), or added to the `sysadmin` role (discussed later in the chapter) after installation. A SQL login, `sa`, is also created. The `sa` account has full administrative access to all SQL functions. During installation, you are prompted to specify a password for the `sa` account. Regardless of whether you install SQL Server using Windows Authentication mode or Mixed mode, the `sa` account is disabled and remains disabled until you choose to enable the account.

Another new feature in SQL Server 2008 is the ability to create a SQL Server login that is mapped to a certificate or asymmetric key through the GUI. SQL Server 2005 had allowed this mapping to be created only through T-SQL. This mapping must be specified during login creation, and the certificate or asymmetric key must be created before the login can be mapped to it. Creation and management of certificates and symmetric keys are covered later in this chapter.

Creating Logins in Management Studio

To create logins from Management Studio, follow these steps:

1. From Object Explorer, expand your server.
2. Expand the Security folder.
3. Right-click Logins and select ''New Login.''
4. In the Login–New dialog box (see Figure 6-3), either type the Login name you want to add or click the Search button to browse for a Windows account.



Login name:
(● Windows authentication) ○ SQL Server authentication
Password:
Confirm password:
Old password:
○ Mapped to certificate
○ Mapped to asymmetric key
☐ Map to Credential
Mapped Credentials
Default database: master
Default language: <default>
Server: AUGHTEIGHT
Connection: AUGHTEIGHT\Administrator

Figure 6-3: New login dialog box.

5. If you are creating a SQL Login, select the ''SQL Server authentication'' radio button.
6. Also, when you select ''SQL Server authentication,'' you can choose not to enforce the password policies.
7. You may also want to change the user's default database and language.

Creating a New Login for Alice

To create a new login for Alice, follow these steps:

1. From Object Explorer, expand your server.
2. Expand the Security folder.
3. Right-click Logins and select ''New Login.''
4. In the New Login dialog box, click Search.
5. In the ''Select User or Group'' dialog box, type **Alice** and click OK.
6. Select `AdventureWorks2008` as the default database.
7. Click OK.

Creating Logins Using T-SQL

Alternatively, you can use the `CREATE LOGIN` statement. `CREATE LOGIN` allows you to create either Windows or SQL logins. This statement is designed to replace two stored procedures that were used in previous versions of SQL, `sp_grantlogin` and `sp_addlogin`. Both of these stored procedures are still available in SQL Server 2008, primarily for backward compatibility, but they have been deprecated and may be removed in a future version of SQL. Use the following format for the `CREATE LOGIN` statement:

```
CREATE LOGIN [name] {WITH <options> | FROM <source>}
```

The following tables show the options available with this statement:

[redacted]

Continued

[redacted]

[redacted]

SQL Server will automatically hash a password before storing it in the database. Be careful about using the `HASHED` *option unless you are sure that the password you are supplying has already been hashed by SQL Server. For example, if you type the following statement:*

```
CREATE LOGIN Bill WITH PASSWORD = 'P@ssw0rd' HASHED
```

SQL will assume that `P@ssw0rd` *is a hash of another value. So, when Alice tries to log in with* `P@ssw0rd`*, the authentication will fail. You can use the* `loginproperty` *function to obtain the hashed value of an existing user's password, as shown in the following example:*

```
SELECT LOGINPROPERTY('bill', 'passwordhash')
```

Managing Logins

SQL Server Management Studio includes several property sheets to configure logins, which are addressed later in this chapter. In addition to the General property sheet, you should also be familiar with the Status page, which allows you to enable or disable the login, unlock the login, and specifically grant or deny access to connect to this SQL Server.

From the General property sheet, you can change the following attributes:

- Password
- Password Policy
- Password Expiration
- Force the user to change the password at the next login
- Default Database
- Default Language

Logins can also be managed using the `ALTER LOGIN` statement. In addition to many of the options listed previously for the `CREATE LOGIN` statement, the `ALTER LOGIN` statement uses the following format:

```
ALTER LOGIN name {<status> | WITH <options>}
```

The following table shows the additional options available with this statement:

[redacted]

Using CREATE LOGIN

To create a new login in Transact-SQL, use the `CREATE LOGIN` statement. The following example creates a new login for a user account named *Bob* on the AughtEight server:

```
CREATE LOGIN [AughtEight\Bob] from Windows;
GO
```

To create a new login for a Windows group, use the following example:

```
CREATE LOGIN [AughtEight\G NorthWest Sales] from Windows;
GO
```

To create a new SQL Server login for Carol, use the following syntax:

```
CREATE LOGIN Carol
WITH PASSWORD = 'Th1sI$|\/|yP@ssw0rd';
GO
```

To change Carol's password to use the all-lowercase *newpassword*, use the following command:

```
ALTER LOGIN Carol WITH PASSWORD = 'newpassword',
CHECK_POLICY=OFF;
GO
```

To remove an existing login, use the `DROP LOGIN` statement. For example, if you want to remove Bob's login (remember, Bob has a Windows-based login), use the following:

```
DROP LOGIN [AughtEight\Bob];
GO
```

For More Information

For backward compatibility, Microsoft SQL Server 2008 supports the stored procedures for managing logins listed in the following table. Because these stored procedures have been deprecated, you should use the `CREATE LOGIN` and `ALTER LOGIN` statements.

[redacted]

Credentials

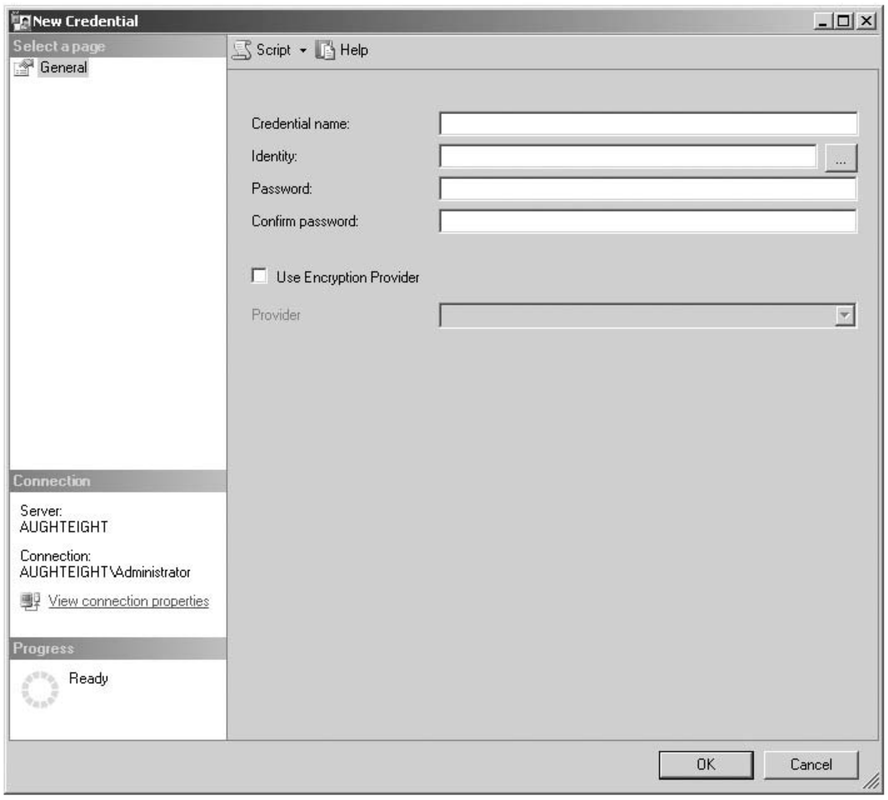
Microsoft SQL Server 2008 also includes a feature for mapping SQL Server logins to external Windows accounts. This can be extremely useful if you need to allow SQL Server logins to interact with the resources outside the scope of the SQL Server itself (such as a linked server or a local file system). They can also be used with assemblies that are configured for `EXTERNAL_ACCESS` permissions.

Credentials can be configured as a one-to-one mapping or a many-to-one mapping, allowing multiple SQL Server logins to use one shared Windows account for external access. In SQL Server 2008, logins can now be associated with multiple credentials, whereas SQL Server 2005 only allows a login to be mapped to a single credential. Credentials can also be configured to use an EKM provider

Creating a New Credential

When creating a new credential, follow these steps:

1. In Object Explorer, expand your server.
2. Expand the Security folder.
3. Right-click Credentials and select ''New Credential'' (see Figure 6-4).


New Credential

Select a page
General

Script
Help

Credential name:
Identity:
Password:
Confirm password:

☐ Use Encryption Provider

Provider

Connection
Server:
AUGHTEIGHT
Connection:
AUGHTEIGHT\Administrator
View connection properties

Progress
Ready

OK
Cancel

Figure 6-4: New Credential properties screen.

4. Type a name for the credential.
5. In the Identity section, either type the name of a Windows account or click the '' . . . '' button to browse for an account.

6. Enter the password for the account.
7. Re-enter the password to confirm.
8. Enable Use Encryption Provider (if desired).
9. Select a valid External Key Management provider (if the above option is selected).
10. Click OK.

Using Transact-SQL

You can use the `CREATE CREDENTIAL` statement as an alternative means to create a new SQL credential object. The syntax is as follows:

```
CREATE CREDENTIAL name WITH IDENTITY = 'identity_name' [, SECRET = 'secret']
 [FOR CRYPTOGRAPHIC_PROVIDER provider_name]
```

Likewise, the `ALTER CREDENTIAL` statement can be used to alter the name of the credential, the identity it's associated with, and the password. Once the credential is no longer needed, it can be removed with the `DROP CREDENTIAL` command, as follows:

```
DROP CREDENTIAL name
```

Create a New Credential for a Windows Account

Earlier in the chapter, you created a Windows account named `CarolStreet` with a password of `P@ssw0rd`. You will now create a new credential named `StreetCred` for that user. When running the following script, replace *AughtEight* with your own server name:

```
USE master
CREATE CREDENTIAL StreetCred
WITH IDENTITY = 'AughtEight\CarolStreet',
SECRET = 'P@ssw0rd';
GO
```

You can then associate Carol's SQL Server login with the `StreetCred` credential:

```
ALTER LOGIN Carol WITH CREDENTIAL = StreetCred;
GO
```

Server Roles

Microsoft SQL Server 2008 defines eight server-level roles that are available to simplify management (and the delegation of management) for SQL logins. These are often referred to as *fixed server roles* because membership is the only thing you can change about these roles. The fixed server roles are designed to allow you to automatically assign a common set of permissions to a login, based on the purpose of the role.

Additionally, SQL Server 2008 also includes a `public` server role. In addition to customizing the member list of the `public` server role, you can also define protocol-specific permissions for Tabular Data Stream

(TDS) endpoints. These endpoints are covered in more detail in Chapter 7. By default, all logins are members of the `public` server role.

Using Fixed Server Roles

The following table shows the fixed server roles in the order they appear on the server:

[redacted]

To add a login to a fixed server role, use the `sp_addsrvrolemember` stored procedure. The stored procedure uses the following format:

```
sp_addsrvrolemember [ @loginame= ] 'login' , [ @rolename = ] 'role'
```

Simply provide the login name and the role name. To add Ted to the `securityadmin` role, use the following command:

```
USE master
CREATE LOGIN Ted WITH PASSWORD = 'P@ssw0rd';
GO
EXEC sp_addsrvrolemember 'Ted', 'securityadmin';
GO
```

Use `sp_dropsrvrolemember` to remove a login from a fixed server role. The syntax is similar to the `sp_addsrvrolemember` stored procedure, as shown in the following example:

```
USE master
EXEC sp_dropsrvrolemember 'Ted', 'securityadmin';
GO
```

For More Information

You can query the Security Catalog Views to find out more information about principals at the server scope. The following table shows views that identify server-level principals:

[redacted]

Database Users

Database users are another component of the security model employed by Microsoft SQL Server 2008. Users are granted access to securable database objects, either directly or through membership in one or more database roles. Users are also associated with ownership of objects such as tables, views, and stored procedures.

When a login is created, unless it is a member of a fixed server role with administrative privileges to all databases, that login has no explicit permissions within the various databases attached to the server. When this happens, the login is associated with the guest database user and inherits the permissions of that user account.

When managing database users in SQL Server Management Studio, you have several options from which to select. On the General property sheet (see Figure 6-5), you will be able to specify a name for the user and associate the user with an existing login. Note that the username does not have to match the login name. For ease of administration, it is best practice to try to use a consistent naming convention, but it is not required. Also, note that there are radio buttons that show whether the user is mapped to a login, a certificate, a key, or without any association. Through the Graphical User Interface (GUI), you can only create a user mapped to a login. In the next section, you see how to create users with other mappings.

Other options you can configure from the General page include specifying the user's default schema, schemas owned by this user (if any), and to which database roles the user belongs. In the Securables page, you can list all the securable objects the user has permissions to and what permissions they have. Finally, you have the Extended Properties page, which allows you to designate or view additional metadata information about this user.

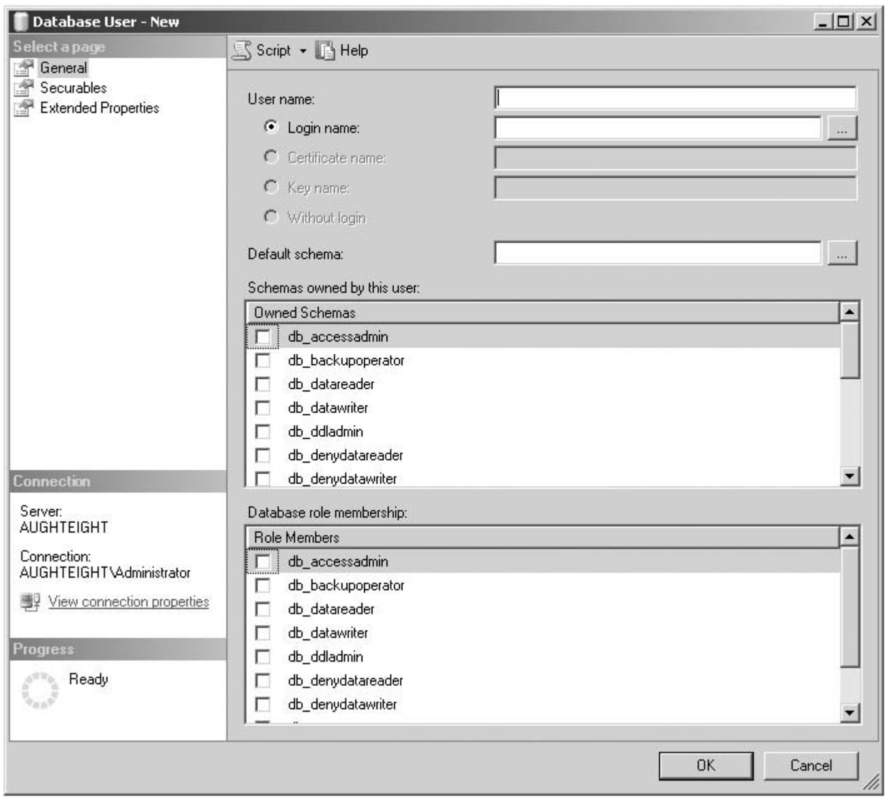


Figure 6-5: Database User–New General property page.

Create a New User and Default Schema

For this example, you will create a new database user in the `AdventureWorks2008` database for Carol and set her default schema to the `Sales` schema.

1. In Object Explorer, expand Databases.
2. Expand `AdventureWorks2008` (see Figure 6-6).
3. Expand Security.
4. Right-click Users and select ''New User.''
5. Type **Carol** in the User Name box.
6. Type **Carol** in the ''Login name'' box, or select her login using the ''...'' button.
7. Type **Sales** in the ''Default schema'' box.
8. Click OK.

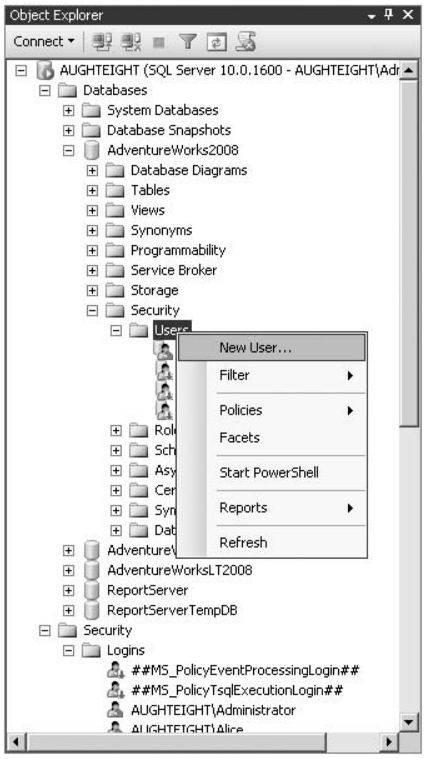


Figure 6-6: New database user.

Now that Carol has a database user account in the `AdventureWorks2008` database, she has inherited the permissions granted to the public database role. Database roles and permissions are covered later in this chapter.

CREATE USER

The `CREATE USER` statement can also be used for creating new database users. `CREATE USER` offers more options over how the user is created than the GUI allows. For example, you can create a user based on an existing certificate or key, or even create a user who is not associated with a login. Although reasons for implementing these types of users will be limited, they can have access to database objects without being associated with a specific login. They can be used to access resources that have specific security requirements. For example, a stored procedure might contain the `EXECUTE AS` clause, in which case, the stored procedure runs as the user associated with a particular certificate, or asymmetric key. The caveat, though, is that these users are valid only in the database in which they were created. If they attempt to access resources in another database, they will access the other database as `guest`. If the `guest` user is disabled in the other database, then they will be denied access.

Each database has two users created by default. The `dbo` user (also known as the *database owner*) has all rights and privileges to perform any operation in the database. Members of the fixed server role, `sysadmin`, as well as the `sa` account, are mapped to `dbo`. Any object created by a `sysadmin` is automatically owned by `dbo`. The `dbo` user is also the owner of the default schema, also called `dbo`. The `dbo` user cannot be deleted.

The guest account is also present in every database, but is disabled by default. The guest account is commonly used when a person has login access to the SQL Server, but no explicit user access to a database. If the database has a guest account and it is enabled, then the login will connect to that database with guest access. guest is a member of the public role and has all of the permissions assigned to that role, but can be granted explicit permissions to securables as well.

You may also notice two other ''users,'' sys and INFORMATION_SCHEMA. Although they are not users in the conventional sense, they do own objects in the database, primarily for storing and retrieving metadata. These users are not mapped to any login and are disabled by default.

The following is the syntax and options for the CREATE USER statement:

```
CREATE USER name [{{FOR | FROM} source | WITHOUT LOGIN]
     [WITH DEFAULT_SCHEMA = schema_name]
```

The following tables explain the options that are available:

[redacted]

[redacted]

Create a New User

Take a look at the CREATE USER statement in action. In an earlier example, you created a new SQL Server login called Carol and an associated user in the AdventureWorks2008 database. If you wanted to create a user for Carol in the tempdb database, you could execute the following statement:

```
USE tempdb;
CREATE USER Carol;
GO
```

That's all there is to creating a new user.

Look at another example. If you executed the DROP LOGIN [AughtEight\Bob] statement earlier, you'll need to re-create his login. In this example, you'll create a database user named BillyBob who will be mapped to Bob's login, and set BillyBob's default schema to the Sales schema:

```
USE master;
CREATE LOGIN [AughtEight\Bob] FROM WINDOWS;
USE AdventureWorks2008;
CREATE USER BillyBob FOR LOGIN [AughtEight\Bob]
WITH DEFAULT_SCHEMA = sales;
```

The last example shows creating a new user from an existing certificate. Certificates are covered later in this chapter, but for this example, create the certificate first, and then create the user:

```
USE AdventureWorks2008;
CREATE CERTIFICATE SalesCert
   ENCRYPTION BY PASSWORD = 'P@ssw0rd'
    WITH SUBJECT = 'Sales Schema Certificate',
    EXPIRY_DATE = '12/31/2010';
GO
CREATE USER SalesSecurity FOR CERTIFICATE SalesCert;
GO
```

You can also use the ALTER USER statement to make changes to a user account. This is another example where Transact-SQL gives you greater flexibility than Management Studio. ALTER SCHEMA lets you modify both the name property and the DEFAULT_SCHEMA property. If you wish to change the Windows or SQL login that the account is associated with, you can use the LOGIN = option, as well. Be aware that the LOGIN option can only be used to associate a user to a login that is the same type as the one it was originally created as. This will not work for users created as certificate or keys. These options are illustrated in the following examples:

```
USE AdventureWorks2008
ALTER USER SalesSecurity
WITH NAME = SalesSchemaSecurity;
GO

USE AdventureWorks2008
ALTER USER BillyBob
WITH DEFAULT_SCHEMA = Production;
GO

--Create a new login
```

```
USE master
CREATE LOGIN TempCarol WITH PASSWORD = 'MyPassword',
CHECK_POLICY = OFF;
GO
USE tempdb
ALTER USER Carol WITH Login = TempCarol;
GO
```

Finally, once a user has outlived its usefulness, use the `DROP USER` statement to remove it from the database. The `DROP USER` statement is straightforward, as seen in the following example:

```
USE AdventureWorks2008
DROP USER BillyBob;
GO
```

Older versions of SQL explicitly tied an object owner into the naming context of the object. For example, if a user named `BillyBob` created a table called `Orders`, the table would be called `BillyBob.Orders`. SQL Server 2008 allows you to separate an object's schema from its owner. This helps to minimize orphaned objects when a user is dropped by keeping those objects part of a schema that may be owned by a role or a Windows group. Although it was easier to manage objects that were all owned by `dbo`, as seen in previous versions, using schemas helps provide a more logical, hierarchical security design.

Fixed Database Roles

Every SQL database has a list of fixed database roles that allow you to delegate permissions to users as necessary. As with the fixed server roles, membership is the only thing you can change about these roles. It is important to know how and when to use these roles.

The following table shows the fixed database roles:

[redacted]

Continued

[redacted]

Note that the fixed database roles include `db_denydatareader` and `db_denydatawriter`. These roles explicitly deny Read or Write access to user tables in the database and should be used sparingly. Deny permissions are authoritative and cannot be overridden.

User-defined database roles offer greater control over managing permissions and access to resources within a database. Frequently, when using a role-based security model, you may find that built-in principals (such as groups in Windows or roles in SQL) offer either too much access or not enough. In this case, you can create user-defined roles that allow you to control access to securable objects for an entire collection of users at once. Database roles are very similar in concept to Windows groups. You can create a database role to identify a group of users, all of whom need access to a common set of resources, or you can use roles to identify the permissions being granted to a securable in the database. Regardless of the purpose of your role, its function should be clearly identified by the name of the role.

Creating a New User-Defined Database Role in Management Studio

In the New Role dialog box, you are prompted to provide a name for the role, as well as identify an owner for the role. The owner of the role can modify it at any time. You can also select existing schemas that will be owned by this role and add users as members to this role. In addition to the General property sheet, you also have the Securables page and the Extended Properties page, which you can use to assign permissions or set additional attributes, respectively.

In this example, you can create a new database role called `ProductionRole` and then add Carol as a member:

1. In Object Explorer, expand Databases.
2. Expand `AdventureWorks2008` and then expand Security.
3. Expand Roles and then expand Database Roles.
4. Right-click ''Database Roles'' and select ''New Database Role.''
5. In the ''Role name'' box, type **ProductionRole** (see Figure 6-7).
6. Under the list of members of this role (which should be empty), click Add.
7. Enter **Carol** in the window and click ''Check Names.'' This should resolve her name. Click OK.
8. In the Database Role–New window, click OK.

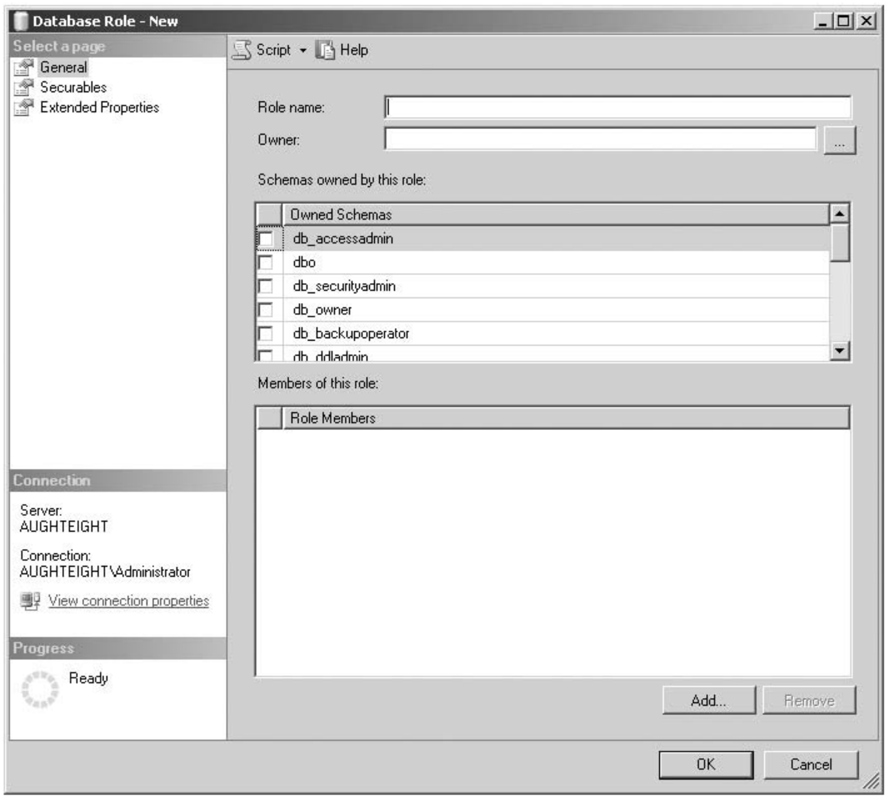


Figure 6-7: Database Role–New properties screen.

CREATE ROLE

`CREATE ROLE` is the Transact-SQL equivalent for creating a new user-defined database role. When using the `CREATE ROLE` statement as shown here, you can also specify the owner of the role. Note that if you are assigning a user as the owner of a role, you must have the `IMPERSONATE` permission, and if you're assigning another role as the owner, you must either be a member of that role or have `ALTER` permission on the role. The following statement creates a role called `SalesStaff`, designating Carol as the owner:

```
USE AdventureWorks2008
CREATE ROLE SalesStaff
AUTHORIZATION Carol;
GO
```

The `ALTER ROLE` statement is fairly limited, allowing you to change only the name of the role:

```
USE AdventureWorks2008
ALTER ROLE SalesStaff
WITH NAME = SalesStaffRole;
GO
```

`DROP ROLE rolename` will let you remove a role from the database once it is no longer needed:

```
USE AdventureWorks2008
DROP ROLE SalesStaffRole;
GO
```

As with fixed server roles, database roles (both fixed and user-defined) can have users added to them either through SQL Server Management Studio or through a stored procedure. The stored procedure for database roles is `sp_addrolemember`. Unlike the stored procedures for adding and dropping members from server roles, `sp_addrolemember` and `sp_droprolemember` identify the role as the first variable and the user as the second.

The following example adds the database user `Carol` to the `db_datareader` role:

```
USE AdventureWorks2008
EXEC sp_addrolemember 'db_datareader', 'Carol';
GO
```

To remove Carol from the `db_datareader` role, use the following stored procedure:

```
USE AdventureWorks2008
EXEC sp_droprolemember 'db_datareader', 'Carol';
GO
```

Application Roles

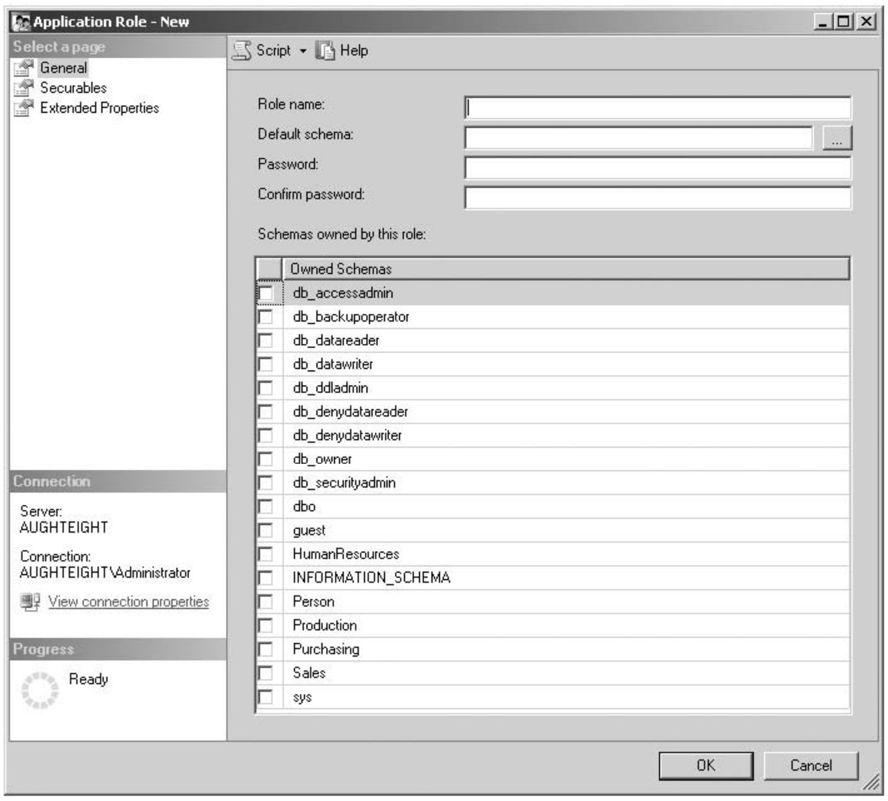
Another type of role that can be used to help secure the database environment is the *application role*. Application roles are quite different from standard role types. They do not have members, and they can (and should) be configured to authenticate with a password. Application roles are typically used when database access must be the same for all users who run a particular application. Rather than depending on the individual user to have the appropriate access for the application to work properly, the application can instantiate the application role without prompting the user to provide a username and password.

You can create a new application role from the Application Roles folder within SQL Server Management Studio. The dialog box for creating a new application role is very similar to the standard database role dialog, with the exception of the password field and the lack of a members list.

Create an Application Role

In this example, you create a new application role named `PurchasingOrderEntry`, with a password of `POEpass1`:

1. In Object Explorer, expand Databases.
2. Expand `AdventureWorks2008` and then expand Security.
3. Expand Roles and then expand Application Roles.
4. Right-click "Application Roles" and select "New Application Role."
5. Type **PurchasingOrderEntry** for the Role name (see Figure 6-8).



Application Role - New

Select a page
General
Securables
Extended Properties

Script Help

Role name:
Default schema:
Password:
Confirm password:

Schemas owned by this role:

	Owned Schemas
☐	db_accessadmin
☐	db_backupoperator
☐	db_datareader
☐	db_datawriter
☐	db_ddladmin
☐	db_denydatareader
☐	db_denydatawriter
☐	db_owner
☐	db_securityadmin
☐	dbo
☐	guest
☐	HumanResources
☐	INFORMATION_SCHEMA
☐	Person
☐	Production
☐	Purchasing
☐	Sales
☐	sys

Connection
Server:
AUGHTEIGHT
Connection:
AUGHTEIGHT\Administrator
View connection properties

Progress
Ready

OK Cancel

Figure 6-8: Application Role–New properties screen.

6. Set the Default schema to ''Purchasing.''
7. Enter **POEpass1** in the Password and ''Confirm password'' boxes.
8. Click OK.

In the next section, you see how to instantiate that role.

Using CREATE APPLICATION ROLE

The `CREATE APPLICATION ROLE` does what the name suggests. When using this statement, specify the name of the application role, a password for the application role, and, optionally, a default schema for the application role. The following example creates an application role named `SalesApp`:

```
USE AdventureWorks2008
CREATE APPLICATION ROLE SalesApp
WITH PASSWORD = 'P@ssw0rd',
DEFAULT_SCHEMA = Sales;
GO
```

To use an application role, you can execute the `sp_setapprole` stored procedure. This can be called from an application, or you can test it from your Query window. The stored procedure includes options to activate the application role by providing an encrypted password, creating a cookie, and setting information in the cookie. The following command activates the `SalesApp` application role and then returns the username:

```
USE AdventureWorks2008
GO
DECLARE @cookie varbinary(8000);
EXEC sp_setapprole 'SalesApp', 'P@ssw0rd'
    , @fCreateCookie = true, @cookie = @cookie OUTPUT;
GO
SELECT USER_NAME();
```

Once you've executed the preceding script, all activity performed from that connection operates under the application role. When the connection is closed, the application role session ends.

With the `ALTER APPLICATION ROLE` statement, you can change the name of the application role, the password, and the default schema. The following example changes the `SalesApp` role name to `OrderEntry` and sets a new password:

```
USE AdventureWorks2008
ALTER APPLICATION ROLE SalesApp
WITH NAME = OrderEntry,
PASSWORD = 'newP@ssw0rd';
GO
```

If you intend to run the `ALTER APPLICATION ROLE` script listed previously, ensure that you don't do it while connected as that application role. Opening a new Query window under your own credentials will prevent errors.

`DROP APPLICATION ROLE` *`rolename`* will remove an application role from the database. Ensure that you do not have any applications still using the application role; otherwise, the application will be unable to connect to the database. For example:

```
USE AdventureWorks2008
DROP APPLICATION ROLE OrderEntry;
GO
```

For More Information

The following Security Catalog Views can be used to identify which principals exist in your database, and their role membership:

[redacted]

For backward compatibility, Microsoft SQL Server 2008 supports the following stored procedures. Keep in mind that these stored procedures are considered ''legacy'' tools and may disappear in a future update or release of SQL Server.

[redacted]

Permissions

A well-implemented security solution answers three questions about security access. Who are you? What can you do? And what have you done? The Kerberos security protocol, which was developed at MIT, is designed to answer these questions through the processes of Authentication (who are you?), Authorization (what can you do?), and Auditing (what have you done?). In an Active Directory forest, SQL Server uses Microsoft's implementation of the Kerberos protocol (named after the three-headed dog who guarded the entrance to Hades), for the Authentication of logins that are associated with Active Directory accounts. Permissions, or Authorizations, are managed from within SQL Server itself and may be configured on server or database objects.

A typical statement to define the permissions on an object or resource will be structured to define a permission state, an action, the object to which the permission and action will apply, and the security principal to whom the permission and action will apply on the defined object. Put simply, it will look like the following:

```
PermissionState Action ON Object TO Principal
```

or

```
GRANT SELECT ON Person.EmailAddress TO Carol
```

To begin with, you should understand there are essentially three permission states that exist: `GRANT`, `GRANT_W_GRANT`, and `DENY`. In addition, when a principal does not have an explicit permission defined, the permission is considered ''revoked.'' The following table shows the different permission states:

[redacted]

To control permission states, you can use Object Explorer or Transact-SQL. The three commands that you can use to control permission states are `GRANT`, `REVOKE`, and `DENY`, which are described in the following table:

[redacted]

The following table shows a general list of the actions you can grant, deny, or revoke, and the types of objects on which you can grant them. A short description is provided for each:

[redacted]

Continued

[redacted]

Now that you understand the permissions and permission states, take a look at the specific permissions available. SQL Server 2008 uses a hierarchical security model that allows you to specify permissions that can be granted at the server, database, schema, or object levels. You can also assign permissions within tables and views for selected columns.

The next section identifies the scopes in which the different securable objects reside and how you can use them to control access to your data. Best practices recommend using a role-based administrative model to simplify the process of creating a secure environment, not only for your databases and database servers, but also for all of your operations.

There are two key strategies you should use when securing your database servers:

- The first strategy you should use when granting permissions is known as the *principle of least privilege*. This strategy mandates that you give your users appropriate permissions to do their jobs, and nothing more. By keeping such tight constraints on your database environment, you can offer a solution that minimizes the attack surface of your servers while maintaining operational functionality.
- The second key strategy is *defense in depth*. A good security implementation will provide security at all layers of a database application. This might include using IPSec or SSL for communications between clients and servers, strong password encryption on your authentication servers, and configuring column-level permissions within a table or view.

When evaluating the different securable objects within SQL Server, you should have a good understanding of where and how permissions apply and how you can use some of the native features of the hierarchical model to your advantage. Permission applied to a specific class of objects at a higher level in the hierarchy allows for permission inheritance. For example, if you want Ted to be able to update any row on every table within the `Sales` schema, you could simply use the following command:

```
USE AdventureWorks2008
--First, create the user
CREATE USER Ted WITH DEFAULT_SCHEMA = Sales;
-- Next, Grant Ted update permissions on the Sales Schema
GRANT UPDATE ON SCHEMA :: Sales to Ted;
GO
```

Alternatively, if you wanted Ted to have the ability to update any object in the database, you could use the following:

```
Use AdventureWorks2008
GRANT UPDATE TO Ted;
GO
```

Take a quick look at the different levels in your security hierarchy. Figure 6-9 outlines the different levels of security you need to manage. In the Windows scope, you create and manage Windows and Active Directory security principals (like users and groups) and manage the files and services needed by the SQL Server and the behavior of the server itself. In the server scope, you manage logins, endpoints, and databases. In the database scope, you work with users, keys, certificates, roles, assemblies, and other database objects. Also in this scope are schemas, which aren't really objects as much as they are object containers. Finally, within the schema scope, you have data types, XML schema collections, and objects. These objects include your tables, views, stored procedures, and more.

Server Permissions

Server control permissions can be managed by simply specifying the permission and the login the permission will be assigned to. For example, to grant permissions to create databases to the login Ted, you could use the following statement:

```
USE master
GRANT CREATE ANY DATABASE TO Ted;
GO
```

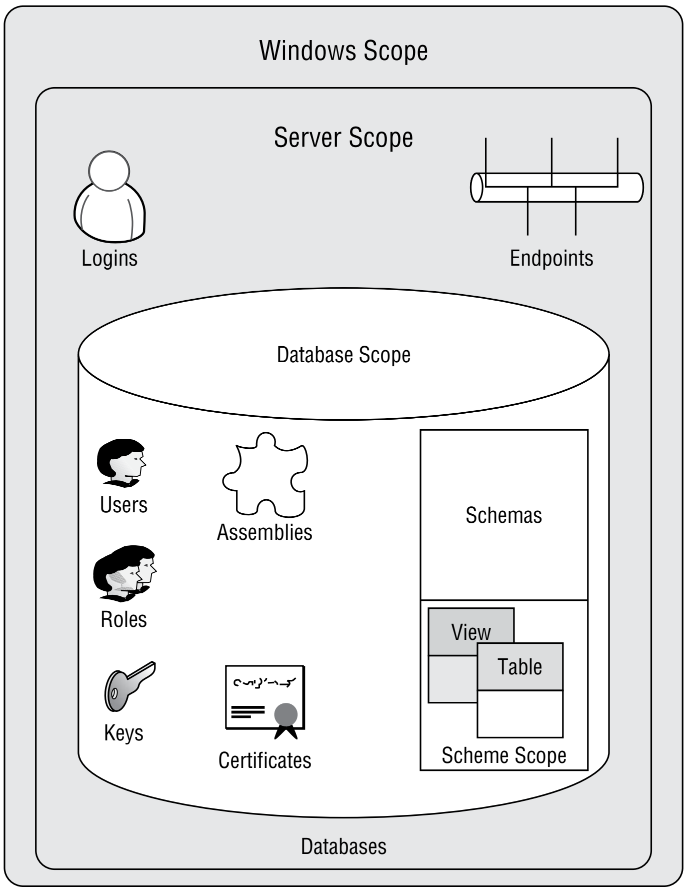


Figure 6-9: Security levels.

If you also wanted Ted to be able to have the permissions to alter logins and to allow others to alter logins, you could use the following statement:

```
USE Master
GRANT ALTER ANY LOGIN TO Ted
WITH GRANT OPTION;
GO
```

To remove Ted's ability to alter logins, you could use the following statement:

```
USE master
REVOKE ALTER ANY LOGIN TO Ted CASCADE;
GO
```

The `CASCADE` keyword is required because you gave Ted the `GRANT_W_GRANT` permission. This ensures that not only will Ted lose his ability to alter any login, but so will anyone that Ted granted the `ALTER ANY LOGIN` permission. If you had not used `GRANT OPTION`, the `CASCADE` keyword would have been optional.

Note that the preceding example revokes a permission that had been previously granted to Ted. If Ted were a member of the `securityadmin` fixed server role, he would still have the ability to alter logins for that server.

Now, if you want to prohibit Ted from being able to create a new database, you could use the `DENY` statement as follows:

```
USE master
DENY CREATE ANY DATABASE TO Ted;
GO
```

Contrary to what I said earlier, the `DENY` permission state isn't always the end-all be-all answer to whether or not a login or user will be able to perform a certain action. If a login is a member of the `sysadmin` fixed server role, that login has complete control over the SQL Server and its resources, and it wouldn't make a lot of sense to prevent that login from being able to access any object on the server. Even if the `DENY` permission statement were successfully executed on an object, the `sysadmin` role can always change the permissions on that object.

Also, if the `GRANT OPTION` was specified in the `GRANT` statement, as with the `REVOKE` keyword, you will need to ensure that you use the `CASCADE` option.

The following table identifies the permissions that can be used to control the server, as well as granting blanket permissions to any resource of a particular type on the server. You can control access by using the following statement:

```
{GRANT | REVOKE | DENY} action on object to principal WITH {options}
```

[redacted]

Continued

[redacted]

Endpoints are server-level objects that use a slightly different syntax from server permissions when granting, revoking, or denying. The following example creates an endpoint named ServiceBroker that will be used for a Service Broker application (endpoints are covered in Chapter 7, and Service Broker is introduced in Chapter 19), and then grants the ALTER permission for that endpoint to Ted:

```
CREATE ENDPOINT ServiceBroker
STATE = STARTED
AS TCP( LISTENER_PORT = 5162 )
FOR SERVICE_BROKER (AUTHENTICATION=WINDOWS);
GO

USE master
GRANT ALTER ON ENDPOINT :: ServiceBroker TO Ted;
GO
```

The following table lists the permissions you can grant for endpoints:

[redacted]

The next server-level object you can set permissions for is logins. The syntax for setting permissions on logins is similar to the syntax for setting permissions on endpoints. For example, to give Carol the ability to alter Ted's login, you would use the following statement:

```
USE master
GRANT ALTER ON LOGIN :: Ted TO Carol
WITH GRANT OPTION;
GO
```

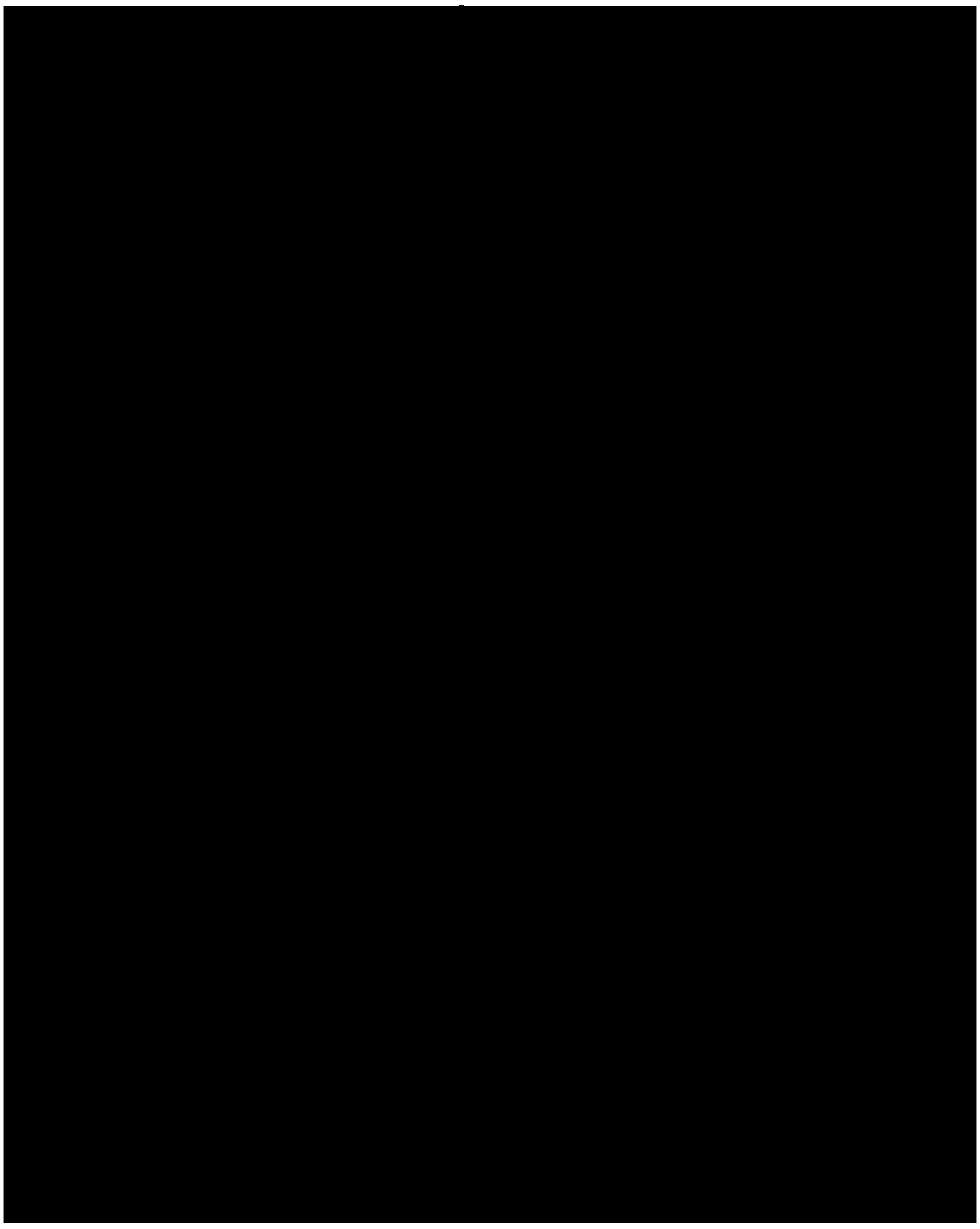
The following table shows how you can control these permissions for logins:

[redacted]

Finally, the last object type at the server level is the database object. Unlike logins and endpoints, database permissions are specified for database users. This keeps the security of the database within the database itself. Additional options may be available based on whether you are granting, denying, or revoking. The following table lists permissions that can be granted on the database object.

[redacted]

Continued



[redacted]

Database Scope Permissions

In the database scope, there are additional permissions you can assign based on the different types of securable objects you have. Permissions assigned to an object class allow you to perform the defined action on all members of that class. However, an object can be explicitly identified by declaring the class and then the object name. The syntax for assigning permissions to database securables is as follows:

```
{GRANT | REVOKE | DENY} action ON class :: object TO principal
```

In the following example, you can grant the `CONTROL` permission for the `Sales` schema to the user `Alice`:

```
USE AdventureWorks2008
CREATE USER Alice FOR LOGIN [AughtEight\Alice]
WITH DEFAULT_SCHEMA = SALES;
GO

GRANT CONTROL ON SCHEMA :: Sales TO Alice;
GO
```

The following table lists the various permissions and the database objects and classes to which you can assign them:

[redacted]

[redacted]

Continued

Schema Scope Permissions

Finally, within the scope of a schema, there are additional permissions you can assign to objects, data types, and XML schema collections. When granting permissions to schema-level objects, the syntax is similar to what you saw earlier:

```
{GRANT | REVOKE | DENY} action ON class :: securable TO principal
```

When the class is an `OBJECT`, you can omit `OBJECT ::` as long as the schema name is included with the object name, as in the following example:

```
Use AdventureWorks2008
GRANT SELECT, UPDATE ON Person.Person to Alice;
GO
```

Schema objects include the following:

- ❑ Aggregates
- ❑ Constraints
- ❑ Functions
- ❑ Procedures

- ❑ Queues
- ❑ Statistics
- ❑ Synonyms
- ❑ Tables
- ❑ Views

The following table lists the schema classes and the permissions that can be set for each of them. Remember that not all permissions are valid for every object type. You can't expect to grant `EXECUTE` on a table, or `SELECT` on a stored procedure.

[redacted]

Using SQL Server Management Studio for Managing Permissions

You can also use Object Explorer in SQL Server Management Studio to set or view permissions on objects. In this section, you will learn how to use the GUI to control access to SQL resources.

The first thing to look at is auditing permissions on the objects themselves.

For the next example, create a new login, a new database user for the `AdventureWorks2008` database, and then grant control permissions to the `Sales` schema for this new user. Use the following code:

```
USE master
CREATE LOGIN Chris WITH PASSWORD = 'P@ssw0rd',
DEFAULT_DATABASE = AdventureWorks2008;
GO

USE AdventureWorks2008
CREATE USER Chris WITH DEFAULT_SCHEMA = Sales;
GO

GRANT CONTROL ON SCHEMA :: SALES TO Chris;
GO
```

Now, use Object Explorer to see what permissions have been granted to Chris. First, look at the database itself:

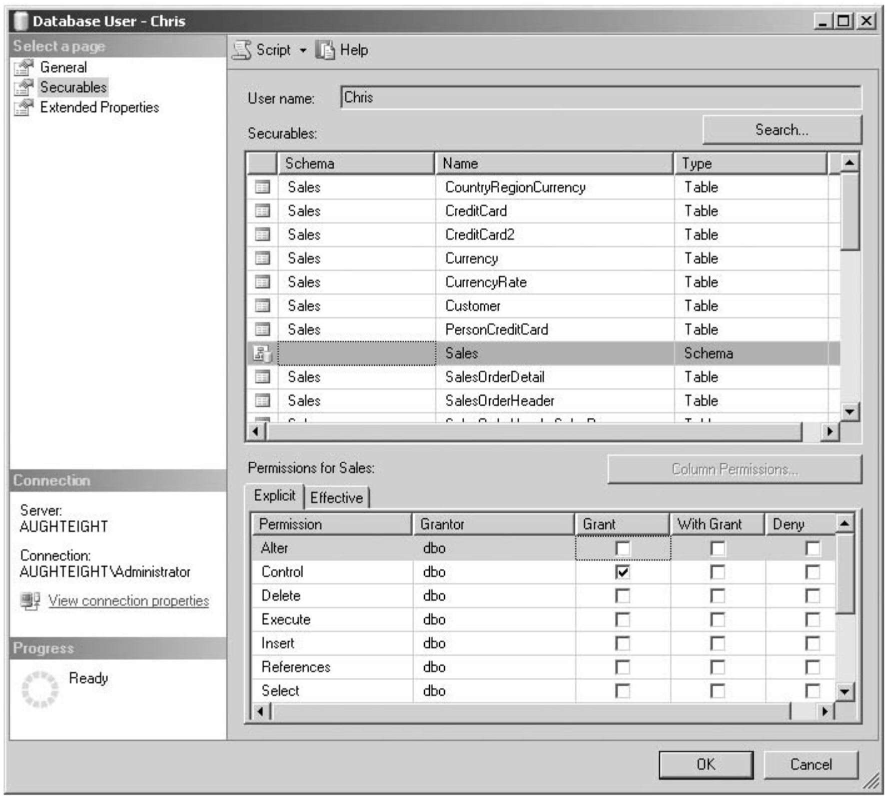
1. Expand your server.
2. Expand Databases.
3. Right-click `AdventureWorks2008` and select Properties.
4. Select Permissions.
5. In the Users or Roles pane, select ''Chris.''

Under ''Explicit permissions for Chris,'' scroll down until you find ''Connect.'' Note that the user who granted the permission, in this case the `dbo`, is also listed in the `Grantor` column.

Next to the list of explicit permissions for this user, there is an ''Effective Permissions'' tab. Clicking on this tab will give you a list of the permissions the user has for this resource, including those that were granted through membership in a role or group. This new feature can really help simplify the process of auditing your security settings, or troubleshooting why a user is having problems accessing a resource.

Because you granted control of the `Sales` schema to Chris, take a look at what permissions have actually been assigned to that schema and the objects within it. To do this, open the property sheet for Chris's user account in the `AdventureWorks2008` database (see Figure 6-10):

1. Close the Database Properties — AdventureWorks2008 window by clicking OK or Cancel.
2. In Object Explorer, expand `AdventureWorks2008`.



Database User - Chris

Select a page: General, Securables, Extended Properties

User name: Chris

Securables:

	Schema	Name	Type
	Sales	CountryRegionCurrency	Table
	Sales	CreditCard	Table
	Sales	CreditCard2	Table
	Sales	Currency	Table
	Sales	CurrencyRate	Table
	Sales	Customer	Table
	Sales	PersonCreditCard	Table
		Sales	Schema
	Sales	SalesOrderDetail	Table
	Sales	SalesOrderHeader	Table

Permissions for Sales:

Explicit | Effective

Permission	Grantor	Grant	With Grant	Deny
Alter	dbo	☐	☐	☐
Control	dbo	☑	☐	☐
Delete	dbo	☐	☐	☐
Execute	dbo	☐	☐	☐
Insert	dbo	☐	☐	☐
References	dbo	☐	☐	☐
Select	dbo	☐	☐	☐

Connection

Server:
AUGHTEIGHT

Connection:
AUGHTEIGHT\Administrator

View connection properties

Progress

Ready

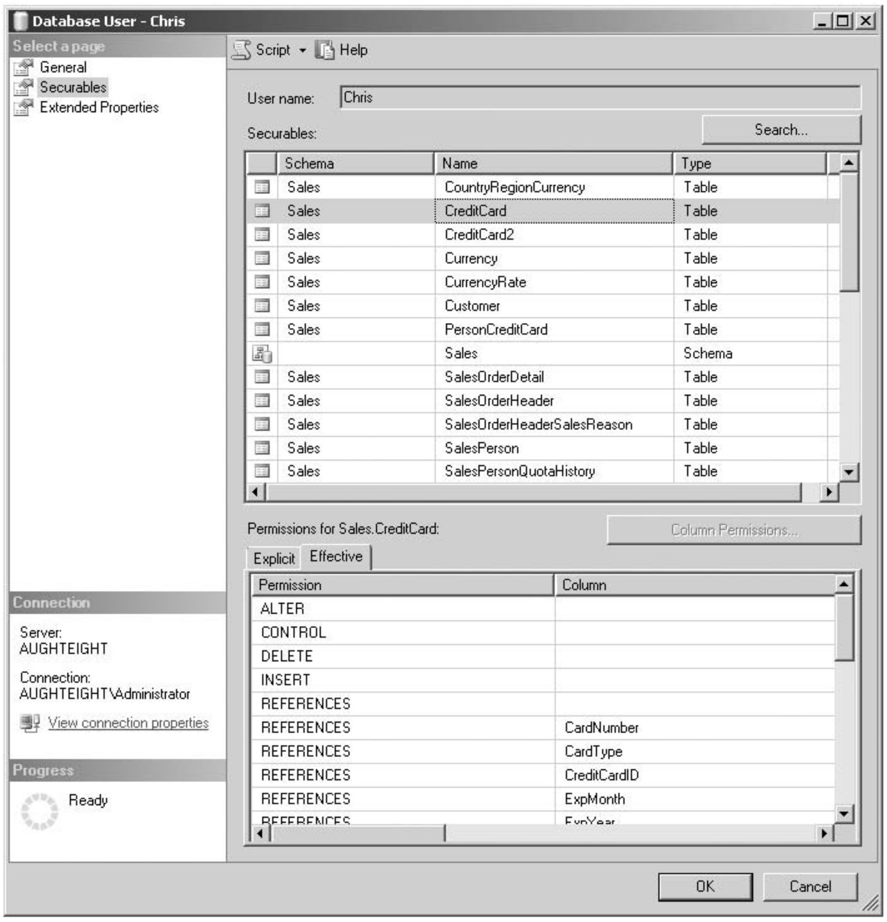
Figure 6-10: Property sheet for Chris.

3. Expand Security.
4. Expand Users.
5. Right-click ''Chris'' and select ''Properties.''
6. Select the Securables page and click Search.
7. Select ''All objects belonging to the schema.''
8. From the ''Schema name'' dropdown list, select ''Sales.''
9. Click OK.

If you look at the list of explicit permissions on the Sales schema, notice that Chris only has `CONTROL` permissions. Clicking the ''Effective Permissions'' tab will show you that the user has full access to any object in the schema.

Now, take a look at specific objects in the `Sales` schema. Select CreditCard in the list of Securables, and select the ''Effective Permissions'' tab.

Look at the list of explicit permissions for `Sales.CreditCard` (see Figure 6-11), and notice that Chris has no explicit permissions on this table. Clicking the ''Effective Permissions'' tab will show you that the user has full access to the table and its contents.



User name: Chris

Server: AUGHTEIGHT

Connection: AUGHTEIGHT\Administrator

Securables:

Schema	Name	Type
Sales	CountryRegionCurrency	Table
Sales	CreditCard	Table
Sales	CreditCard2	Table
Sales	Currency	Table
Sales	CurrencyRate	Table
Sales	Customer	Table
Sales	PersonCreditCard	Table
	Sales	Schema
Sales	SalesOrderDetail	Table
Sales	SalesOrderHeader	Table
Sales	SalesOrderHeaderSalesReason	Table
Sales	SalesPerson	Table
Sales	SalesPersonQuotaHistory	Table

Permissions for Sales.CreditCard:

Explicit | Effective

Permission	Column
ALTER	
CONTROL	
DELETE	
INSERT	
REFERENCES	
REFERENCES	CardNumber
REFERENCES	CardType
REFERENCES	CreditCardID
REFERENCES	ExpMonth
REFERENCES	ExpYear

Figure 6-11: `Sales.CreditCard` **permissions.**

You now have a user with full access to the `Sales` schema, but no access to resources outside of it. Any attempt to query a view in another schema will result in the following error:

[redacted]

Also note that you can add permissions for database objects in the User Properties dialog box. You can use the Management Studio to assign permissions by editing the properties of the securable, or by editing the properties of a principal.

SQL Server Encryption

Protecting data, both in storage and during transmission, is important for the integrity of your applications and services. Microsoft SQL Server 2008 offers several options for both. In this section, you will see some of the tools available for protecting your data.

First of all, whether you're using symmetric keys, asymmetric keys, or certificates, there are two main components to encrypting data: the *encryption algorithm* and the *key value*. The encryption algorithms available include Data Encryption Standard (DES), Triple Data Encryption Standard (3DES), RC4, and Advanced Encryption Standard (AES_256), as well as others. An encryption algorithm is simply a mathematical formula that dictates how to turn the data from plain text into cipher text. The key is a value that is used within that formula to determine the actual output based on the input. It's not unlike basic algebra, where you take a statement like $x + y = z$. In this case, x is the plain-text value, y is the encryption key, and z is the cipher text. Fortunately, the encryption algorithms are significantly more complex than that, but you get the idea.

Keys come in two flavors: symmetric and asymmetric. *Symmetric keys* use the same data key value to both encrypt and decrypt data. This is actually very good for encrypting large amounts of data, but has a relatively low level of security. *Asymmetric keys* use one key value for encrypting data and a different value for decrypting data. This provides a higher level of security than symmetric keys, but is a costly operation and not good for large amounts of data. A well-designed encryption method encrypts data using symmetric keys, and encrypts the symmetric keys using asymmetric keys. Certificates use asymmetric keys, but have additional functionality that can be used for authentication and non-repudiation.

Now, take a look at how SQL provides encryption services. Figure 6-12 shows a high-level overview of the encryption hierarchy used by SQL Server 2008. At the top level is the Windows layer, which includes the Windows Data Protection API (DPAPI). The DPAPI is responsible for encrypting the server's service master key using the server's local machine key. The *service master key* is the top of the encryption food chain within the SQL environment. The service master key is automatically generated the first time a lower-level key is created.

Beneath the service master key is the *database master key*. The database master key can protect the private keys of all certificates and asymmetric keys within a database. It is a symmetric key that is encrypted using the 3DES algorithm and a password. Copies of the key are encrypted using the service master key and are stored in both the master database and the database for which it was created. If the database is moved to another server, the database master key can be decrypted by using the `OPEN MASTER KEY` statement and providing the password used to encrypt it.

Also in the database scope are symmetric and asymmetric keys you can create for encrypting data, as well as certificates that can also be used for digital signing and non-repudiation. Creating and managing the different key types are discussed in the next section.

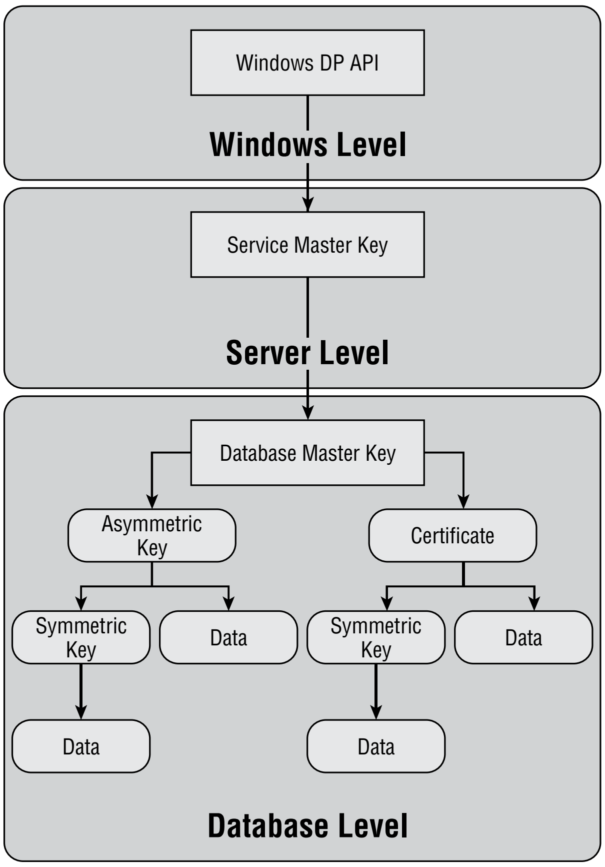


Figure 6-12: Encryption hierarchy.

One of the first steps you should take is creating the database master key. Remember that the database master key is a symmetric key that encrypts all private key data within the database. This is helpful if you are using asymmetric keys or certificates, in that they can be created without having to supply a password or other mechanism to protect the private keys associated with both. To create a new master key for the `AdventureWorks2008` database, you can execute the following command:

```
USE AdventureWorks2008
CREATE MASTER KEY
ENCRYPTION BY PASSWORD = 'P@ssw0rd';
GO
```

Creation of a master key requires `CONTROL` permission on the database. Also, if you already have a master key created, you must drop the existing one if you need to create a new master key. An existing master key cannot be dropped if it is being used to encrypt a private key in the database.

Once you've created your master key, you can query the `sys.databases` catalog view to see if the database master key has been encrypted using the service master key by looking at the value of the is_master_key_encrypted_by_server column. This column uses a Boolean value to indicate whether the database master key is encrypted with the service master key. The value may be 0 if the database master key was created on another server.

```
SELECT NAME, [is_master_key_encrypted_by_server] FROM sys.databases
GO
```

Before continuing on to the subject of working with other keys to encrypt database information, let's look at the topic of backing up your service master key and database master keys. This can be extremely valuable in case you have to perform a disaster-recovery operation and need to recover data that had been protected or encrypted with one of these keys. The syntax for both keys is similar, but an additional step is required to back up an encrypted database master key.

Let's start with the service master key first. Quite simply, use the BACKUP SERVICE MASTER KEY statement with a file path, which can be a local or UNC path, and a password that meets your password-complexity requirements. Using a password on the backup file prevents someone from being able to restore your master key on another server and then being able to decrypt your database master keys. The following example will save a backup of the service master key to a folder called *C:\KeyBackups* (this folder must already exist):

```
BACKUP SERVICE MASTER KEY TO FILE = 'C:\KeyBackups\ServiceMasterKey'
ENCRYPTION BY PASSWORD = 'c@MplexP@ssw0rd';
GO
```

If you need to restore the service master key, you can issue the following statement:

```
RESTORE SERVICE MASTER KEY FROM FILE = 'C:\KeyBackups\ServiceMasterKey'
DECRYPTION BY PASSWORD = 'c@MplexP@ssw0rd';
GO
```

To back up and restore a database master key, use the following examples:

```
--Backup the database master key
USE AdventureWorks2008;
OPEN MASTER KEY
 DECRYPTION BY PASSWORD = 'P@ssw0rd'
BACKUP MASTER KEY TO FILE = 'C:\KeyBackups\AWorksMasterKey'
ENCRYPTION BY PASSWORD = 'dn9e8h93ndwjKJD';
GO
--Restore the database master key
USE AdventureWorks2008;
RESTORE MASTER KEY FROM FILE = 'c:\KeyBackups\AWorksMasterKey'
DECRYPTION BY PASSWORD = 'dn9e8h93ndwjKJD'
ENCRYPTION BY PASSWORD = 'P@ssw0rd'
GO
```

There are a couple of things to note about the previous example. First, in order to back up the database master key, you must decrypt it by using the password that was originally used to encrypt it. Also note that when you use the RESTORE MASTER KEY statement, you need to provide a password for encrypting the database master key. The command will fail without this step.

Extensible Key Management (EKM)

One of the most important new features of SQL Server 2008 is Extensible Key Management, or EKM for short. EKM works with the Microsoft Cryptographic API (MSCAPI) to allow encryption keys that are used for data encryption, as well as encryption of other keys, to be generated and stored outside of the SQL Server 2008 environment. This provides a more robust and flexible mechanism for key management, given that you can now separate the keys from the data they protect.

This is often accomplished through the use of Hardware Security Models (HSM). HSM vendors can create a provider that will interface with the MSCAPI, exposing at least some of the features of the HSM to SQL Server 2008 and other applications that leverage the MSCAPI. Unfortunately, because MSCAPI acts as a middle tier between the HSM and the SQL Server, not all of the features of the HSM may be exposed to the SQL Server.

In order to use EKM, you must first enable it on the server. It is turned off by default, but can be turned on with the `sp_configure` command. Because enabling EKM is considered an advanced feature, the `show advanced` configuration must also be specified. The following example shows you how to turn on EKM for your server:

```
sp_configure 'show advanced', 1;
GO
RECONFIGURE
GO
sp_configure 'EKM provider enabled', 1;
GO
RECONFIGURE
GO
```

With EKM enabled, you can now store your encryption keys on HSM modules, smart cards, or USB devices. Whenever data is encrypted using a key stored on one of these devices, that same device must be present in order to decrypt the data. This can protect against an unauthorized user copying and attaching the database files to a rogue SQL Server and being able to access all your confidential data.

EKM can also be leveraged to provide the following benefits:

- ❑ Additional authorization checks
- ❑ Easier key recovery
- ❑ Encryption key retrieval
- ❑ External key generation and storage
- ❑ External key retention and rotation
- ❑ Higher performance when using hardware-based encryption and decryption
- ❑ Manageable key distribution
- ❑ Secure key disposal

Encryption Tools

Now that you understand some of the basics of encryption, take a look at creating and managing encryption tools. Each of the objects in this section serves a specific purpose. After you learn how to create symmetric keys, asymmetric keys, and certificates, you will learn how to use them.

Symmetric Keys

As mentioned earlier, symmetric keys offer an efficient model for being able to encrypt large amounts of data. The resource overhead is minimized by using the same keys for both encryption and decryption. Here's the syntax for generating symmetric keys:

```
CREATE SYMMETRIC KEY name [AUTHORIZATION owner] [FROM PROVIDER] providername
 WITH options
ENCRYPTION BY mechanism
```

The following table shows the arguments that can be used:

[redacted]

Create a Symmetric Key

The following example creates a new symmetric key named `SalesKey1`, which uses the 192-bit Triple DES 3-Key algorithm:

```
USE AdventureWorks2008
GO
--Create Symmetric Key
```

```
CREATE SYMMETRIC KEY SalesKey1
     WITH ALGORITHM = TRIPLE_DES_3KEY,
     KEY_SOURCE = 'The quick brown fox jumped over the lazy dog',
     IDENTITY_VALUE = 'FoxAndHound'
     ENCRYPTION BY PASSWORD = '9348hsxasnA@B';
GO
```

You can add or remove methods for encrypting the key with the `ALTER SYMMETRIC KEY` statement, and you can remove a symmetric key by using the `DROP SYMMETRIC KEY` statement.

In this example, use the `SalesCert` certificate created in the earlier section, ''Database Users,'' to encrypt the symmetric key and remove the password encryption from the previous example:

```
--Open the symmetric key
OPEN SYMMETRIC KEY SalesKey1
 DECRYPTION BY PASSWORD = '9348hsxasnA@B'
--Add encryption using the certificate created earlier
ALTER SYMMETRIC KEY SalesKey1
 ADD ENCRYPTION BY CERTIFICATE SalesCert
--Remove the password encryption
ALTER SYMMETRIC KEY SalesKey1
 DROP ENCRYPTION BY PASSWORD = '9348hsxasnA@B'
--Close the symmetric key
CLOSE SYMMETRIC KEY SalesKey1
```

Asymmetric Keys

Asymmetric keys use a pair of keys rather than a single one. These keys are often referred to as the *public key* and the *private key*. One key is used for encryption, and the other is used for decryption. It doesn't really matter which key is used for encryption, but the data cannot be decrypted without the corresponding key. The process for creating an asymmetric key pair is similar to creating a symmetric key. Here's the syntax for generating symmetric keys:

```
CREATE ASYMMETRIC KEY name [AUTHORIZATION owner] [FROM key_source]
 WITH ALGORITHM = algorithm [ENCRYPTION BY PASSWORD = 'password']
```

The following table shows the arguments that can be used:

[redacted]

[redacted]

When creating an asymmetric key pair, you can specify the owner of the key pair and the key source (which is either a strong-name file, an assembly, or an executable assembly file). Alternatively, you can use an algorithm that determines the number of bits used by the private key, selecting a key length using 512, 1,024, or 2,048 bits. You can also use the `ENCRYPTION BY PASSWORD` option to encrypt the private key. If you do not specify a password, the database master key will encrypt the private key.

```
USE AdventureWorks2008
CREATE ASYMMETRIC KEY HumanResources
    WITH ALGORITHM = RSA_2048;
GO
```

You can use the `ALTER ASYMMETRIC KEY` statement to change the properties of a key pair. You can use the `REMOVE PRIVATE KEY` option to take the private key out of the database (make sure you have a backup of the private key first!), or you can change the way the private key is protected. For example, you can change the password used to encrypt the private key and then change the protection from password to database master key, or vice versa.

In the following example, use the following code to encrypt the private key from the `HumanResources` key pair created in the earlier example using a password:

```
USE AdventureWorks2008
ALTER ASYMMETRIC KEY HumanResources
    WITH PRIVATE KEY (
    ENCRYPTION BY PASSWORD = 'P@ssw0rd');
GO
```

In the next example, you can change the password used to encrypt the private key by first decrypting it, and then re-encrypting it with a new password:

```
USE AdventureWorks2008
ALTER ASYMMETRIC KEY HumanResources
```

```
WITH PRIVATE KEY (
DECRYPTION BY PASSWORD = 'P@ssw0rd',
ENCRYPTION BY PASSWORD = '48ufdsjEHF@*hda');
GO
```

Certificates

Certificates (also known as *public key certificates*) are objects that associate an asymmetric key pair with a credential. Certificates are objects that can be used not only for encryption, but also for authentication and non-repudiation. This means that not only can you obfuscate data that would normally be in plain text, but you can also provide a means of guaranteeing the source, the trustworthiness of that source, or that the data has not changed since it was signed.

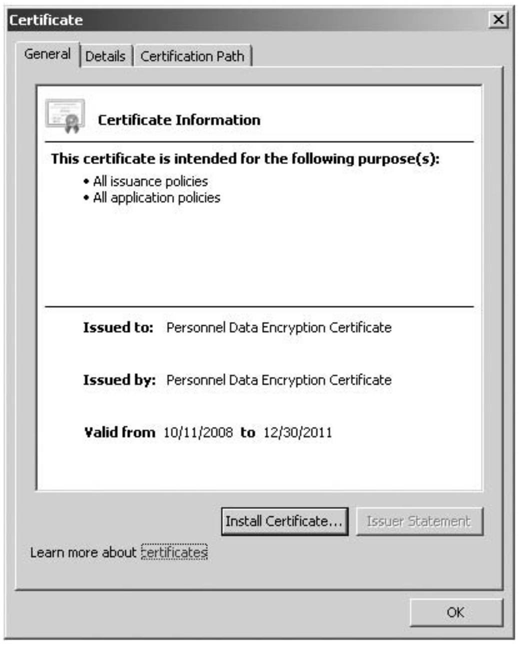
The details of a certificate identify when the certificate was created, the validity period of the certificate, who created the certificate, and what the certificate can be used for. It also identifies the public key associated with the certificate and the algorithm that can be used for digitally signing messages.

The ability to create and use certificates is a feature that was first introduced in SQL Server 2005, and one that even experienced DBAs may have trouble grasping at first. Certificates are part of the bigger scope of application security and identity, and the functionality extended to SQL Server 2008 is no different from how you would use certificates with other applications and services. This topic is almost like opening a Pandora's Box, but once you understand the basics of how certificates work and how they can be used to protect your services and data, you will appreciate their flexibility.

Certificates also have a feature that lets you trace the genealogy of the certificate, its ''family tree,'' if you will (see Figure 6-13). This certificate hierarchy identifies not only what *Certification Authority* (CA) issued the certificate, but what CA generated the certificate used by the CA to generate the certificate you have. This is known as the *certificate chain*. The certificate chain can be used to identify either a common Root CA (the highest authority in a chain) that can be trusted for authentication or another Root CA that is considered a trustworthy source. Many applications and operating systems include a list of commercial CAs that are automatically trusted. When the certificate from a Root CA is trusted, it is assumed that any certificate that can trace its genealogy back to that root is also trusted. If the certificate is not from a trusted certificate chain, the user may be warned that the certificate is not trusted, and they should proceed with caution. Commercial CAs are often used to obtain Server Authentication and SSL certificates, simply because many Web browsers already trust the most popular Root CAs.

Many organizations have developed their own *Public Key Infrastructure* (PKI). These companies have found it necessary to deploy and use certificates for a variety of reasons. Some might use certificates with smart cards for logging in to their computers. Some may use certificates for encrypting data on the NTFS file system, using Encrypting File System (EFS). Some organizations may use certificates for digitally signing applications and macros, so that their users can verify where the application came from or that it hasn't been modified. These organizations often have their own CA hierarchy. They may have a Root CA they manage themselves, or they may have the ability to generate their own certificates that are part of a third-party certificate chain.

Microsoft SQL Server 2008 has the ability to create its own self-signed certificates. In a way, SQL can be its own CA, but don't expect these certificates to be automatically trusted outside of the SQL instance. The certificates generated by SQL Server conform to the X.509 standard and can be used outside of the SQL Server if necessary, but they are not part of a trusted hierarchy. A more common approach is to use a certificate generated by another CA and import that into SQL Server. Certificates can be just as widely used in SQL Server as they can outside SQL. You can use them for server authentication, encryption, and digital signing.



Certificate Information

This certificate is intended for the following purpose(s):
- All issuance policies
- All application policies

Issued to: Personnel Data Encryption Certificate

Issued by: Personnel Data Encryption Certificate

Valid from 10/11/2008 to 12/30/2011

Figure 6-13: Certificate information.

On the subject of encryption, public key certificates operate in the same way as asymmetric keys. The key pair, however, is bound to this certificate. The public key is included in the certificate details, and the private key must be securely archived. Private keys associated with certificates must be secured using a password, the database master key, or another encryption key. When encrypting data, the best practice is to encrypt the data with a symmetric key and then encrypt the symmetric key with a public key.

When creating a certificate that will be self-signed, you can use the `CREATE CERTIFICATE` statement. You can choose to encrypt the private key using a strong password or by using the database master key. You can also use the `CREATE CERTIFICATE` statement to import a certificate and private key from a file. Alternatively, you can create a certificate based on a signed assembly.

Once the certificate has been created, you can modify the certificate with the `ALTER CERTIFICATE` statement. Some of the changes you can make include changing the way the private key is protected or removing the private key from the SQL Server. Removing the private key should be done only if the certificate is used to validate a digital signature. If the public key had been used to encrypt data or a symmetric key, the private key should be available for decryption.

It is a good idea when creating certificates to make a backup of the certificate and the associated private key with the `BACKUP CERTIFICATE` statement. You can make a backup of the certificate without archiving the private key, and use the public key for verification or encrypting messages that can only be decrypted with the private key.

Once a certificate is no longer needed, you can get rid of it with the `DROP CERTIFICATE` statement. Be aware that the certificate can't be dropped if it is still associated with other objects.

Create a New Certificate

In the following example, create a new certificate named `PersonnelDataCert`, which you will use later to encrypt data. After creating this certificate, back up the certificate to the file system (you can either change the path in the example or create a new folder on your C: drive called *certs*). Once that is done, the last step is to import the certificate into the `tempdb` database.

```
-- Create the Personnel Data Certificate
USE AdventureWorks2008;
CREATE CERTIFICATE PersonnelDataCert
   ENCRYPTION BY PASSWORD = 'HRcertific@te'
   WITH SUBJECT = 'Personnel Data Encryption Certificate',
   EXPIRY_DATE = '12/31/2011';
GO

--Backup the certificate and private key to the file system
Use AdventureWorks2008
BACKUP CERTIFICATE PersonnelDataCert TO FILE = 'c:\certs\Personnel.cer'
    WITH PRIVATE KEY (DECRYPTION BY PASSWORD = 'HRcertific@te',
    FILE = 'c:\certs\Personnelkey.pvk' ,
    ENCRYPTION BY PASSWORD = '@notherPassword' );
GO

--Import the certificate and private key into the TempDB database
USE tempdb
CREATE CERTIFICATE PersonnelDataCert
    FROM FILE = 'c:\certs\Personnel.cer'
     WITH PRIVATE KEY (FILE = 'c:\certs\Personnelkey.pvk',
    DECRYPTION BY PASSWORD = '@notherPassword',
     ENCRYPTION BY PASSWORD = 'TempDBKey1');
GO
```

In the next example, change the password used to encrypt the private key using the `ALTER CERTIFICATE` statement:

```
Use tempdb
ALTER CERTIFICATE PersonnelDataCert
    WITH PRIVATE KEY (ENCRYPTION BY PASSWORD = 'P@ssw0rd789',
    DECRYPTION BY PASSWORD = 'TempDBKey1');
GO
```

Now you can remove the private key from the `AdventureWorks2008` database. Because the certificate and the private key are backed up, you can perform this action safely.

```
Use AdventureWorks2008
ALTER CERTIFICATE PersonnelDataCert
    REMOVE PRIVATE KEY
GO
```

Finally, clean up the `tempdb` database:

```
USE tempdb
DROP CERTIFICATE PersonnelDataCert;
GO
```

Encrypting Data

Now that you've seen the different objects that can be used for encryption or non-repudiation, take a look at how you can actually use them. First of all, not everything needs to be encrypted. Because the process of encrypting and decrypting data can be resource-intensive, you should be mindful of what data you need to encrypt. Data that should be kept confidential (such as credit card or Social Security numbers) might fall into this category. An employee's middle name, no matter how embarrassing it might be, would not. Also note that not every data type can be encrypted with the `encryptbykey` function. The valid data types are `nvarchar`, `char`, `wchar`, `varchar`, and `nchar`.

It is also a good idea to know *when* to encrypt the data, and when not to. Frequently queried columns in tables or views should *not* be encrypted, because the process of decrypting large amounts of data that is queried over and over again can often become counterproductive. In this case, a better strategy might be to store the sensitive information in a separate table that has much tighter security on it. Remember that you can give insert or update permissions on a row in a foreign table without having to grant select permissions on that related table. HSMs may offset some of the overhead involved with the decryption process, but that may require significant testing to verify how well it will perform in production.

Prior to encrypting data, you must open the key that will perform the encryption process. Again, data is commonly protected with a symmetric key, which is, in turn, protected with an asymmetric key pair. If the symmetric key is protected with a password, then any user with `ALTER` permissions on the symmetric key and the password can open and close the symmetric key. If the symmetric key is protected by an asymmetric key or certificate, the user also needs `CONTROL` permissions on the asymmetric key or the certificate.

Create an Encrypted Column

Use the following sample code to create an encrypted column in the `Sales.CreditCard` table. In this example, use the symmetric key `SalesKey1` and the certificate `SalesCert`, both created earlier in this chapter:

```
ALTER TABLE Sales.CreditCard
    ADD EncryptedCardNumber varbinary(128);
GO

OPEN SYMMETRIC KEY SalesKey1 DECRYPTION BY
 CERTIFICATE SalesCert WITH PASSWORD = 'P@ssw0rd'

UPDATE Sales.CreditCard
SET EncryptedCardNumber
    = EncryptByKey(Key_GUID('SalesKey1'), CardNumber);
GO

CLOSE SYMMETRIC KEY SalesKey1;
GO
```

Because the symmetric key was used to encrypt the data, it will also be used for decryption. Using the preceding example as a template, you could use the following commands to create another new column that stores the decrypted data. A `SELECT` statement is included that allows you to view the original data, the encrypted data, and the decrypted data columns:

```
ALTER TABLE Sales.CreditCard
    ADD DecryptedCardNumber NVARCHAR(25);
```

```
GO

OPEN SYMMETRIC KEY SalesKey1 DECRYPTION BY
 CERTIFICATE SalesCert WITH PASSWORD = 'P@ssw0rd';
GO

UPDATE Sales.CreditCard
SET DecryptedCardNumber
    = DecryptByKey(EncryptedCardNumber);
GO

CLOSE SYMMETRIC KEY SalesKey1;
GO

Select TOP (10) CreditCardID, CardNumber AS Original, EncryptedCardNumber AS
 Encrypted, DecryptedCardNumber AS Decrypted
FROM Sales.CreditCard;
GO
```

You don't have to create a whole new column to view the decrypted data, though. The `DECRYPTBYKEY` function can be executed in a `SELECT` statement to view the unencrypted data. The following example shows you how:

```
OPEN SYMMETRIC KEY SalesKey1 DECRYPTION BY
 CERTIFICATE SalesCert WITH PASSWORD = 'P@ssw0rd';
GO

SELECT CreditCardID, CardNumber,EncryptedCardNumber
    AS 'Encrypted Card Number',
    CONVERT(nvarchar, DecryptByKey(EncryptedCardNumber))
    AS 'Decrypted Card Number'
    FROM Sales.CreditCard;
GO

CLOSE SYMMETRIC KEY SalesKey1;
GO
```

Transparent Data Encryption

Another new feature of SQL Server 2008 is Transparent Data Encryption (TDE). TDE is designed to perform real-time I/O encryption, using a Database Encryption Key (DEK), of the database and transaction log files for databases that have TDE enabled. The benefit of TDE is that it protects all data ''at rest.'' This means that anything not currently being read into memory is protected using the DEK. However, when a query is run, the data that is retrieved from that query is decrypted as it is being read into memory. Unlike the use of symmetric and asymmetric keys for decrypting data in a single table or column, it is not necessary to invoke a decryption function when reading from or writing to a table in a database protected by TDE (hence the use of the word *Transparent*).

Setting up TDE is slightly more complex than other encryption methods, in that there are certain dependencies that must be in place before you can enable it.

1. First of all, a Database Master Key must exist in the `master` database.
2. Secondly, you must either create a certificate or install a certificate in the `master` database that can be used to encrypt the DEK; or you may use an asymmetric key from an EKM provider.
3. Then you will need to create the DEK in the database that you will be encrypting. Finally, enable encryption on that database. The following script provides an example of these steps:

```
USE master
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'MyStrongP@ssw0rd';
GO
CREATE CERTIFICATE AughtEightTDE WITH SUBJECT =
 'TDE Certificate for the AUGHTEIGHT Server';
GO
USE AdventureWorks2008
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = TRIPLE_DES_3KEY
ENCRYPTION BY SERVER CERTIFICATE AughtEightTDE;
GO
ALTER DATABASE AdventureWorks2008
SET ENCRYPTION ON;
GO
```

You can also use SQL Server Management Studio to manage the Transparent Encryption Properties of a database. Do this by performing the following steps:

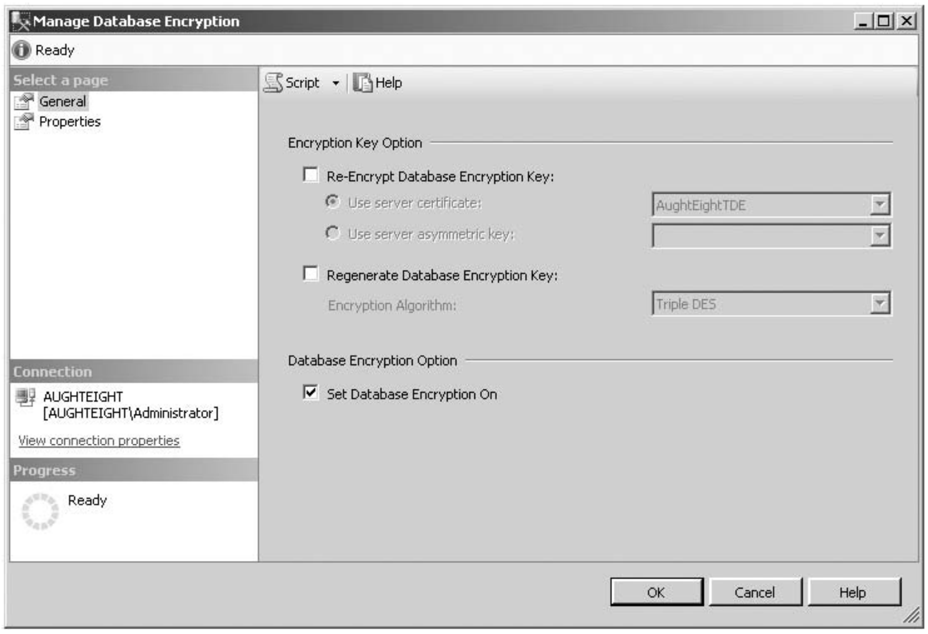
1. In Object Explorer, expand Databases.
2. Right-click on the `AdventureWorks2008` database, and select Tasks, then ''Manage Database Encryption.''

As you can see in Figure 6-14, you have several options you can perform from this window, including ''Re-Encrypt Database Encryption Key'' using a server certificate or server asymmetric key (stored in the master database), as well as regenerating the key using an AES 128, AES 192, AES 256, or Triple DES encryption algorithm. You can also enable or disable TDE for this database by checking (or unchecking) the box next to ''Set Database Encryption On.''

Digital Signatures

Digital signatures provide authentication and non-repudiation. Often, with public key pairs, the private key is used to digitally sign a message (or, in the case of a code-signing certificate, an application or assembly). Take a look at how digital signing works with e-mail messages as an example.

Bob sends Alice a message, and his e-mail client is configured to automatically add his digital signature to all outgoing messages. In this case, while the message is being prepared for delivery, a key is generated and passed to a hashing algorithm for a one-way transformation of the data into a hash value. The hash value is attached to the message, and the key that was used to generate the hash is encrypted with Bob's private key.


Manage Database Encryption

Ready

Encryption Key Option

☐ Re-Encrypt Database Encryption Key:

● Use server certificate: AughtEightTDE

○ Use server asymmetric key:

☐ Regenerate Database Encryption Key:

Encryption Algorithm: Triple DES

Database Encryption Option

☑ Set Database Encryption On

AUGHTEIGHT [AUGHTEIGHT\Administrator]

Ready

Figure 6-14: Manage Database Encryption.

The message is delivered to Alice, who receives the message in plain text, as well as receiving the hashed version of the message. Alice, who has access to Bob's public key, uses it to decrypt the key that was used to generate the hash. The key is then passed through the hashing algorithm, and a new hash is generated. If the new hash matches the hash that was sent with the message, Alice can feel confident that the message hasn't been changed during delivery. If the hash values do not match, then the message may have been altered since it was transmitted and should not be trusted.

In a similar vein, you can use digital signatures to sign SQL Server components (such as stored procedures) to associate the stored procedure with a hash value. If the stored procedure changes by a single bit, then the hash values will differ; and you'll know that someone must have used an `ALTER PROCEDURE` statement on it!

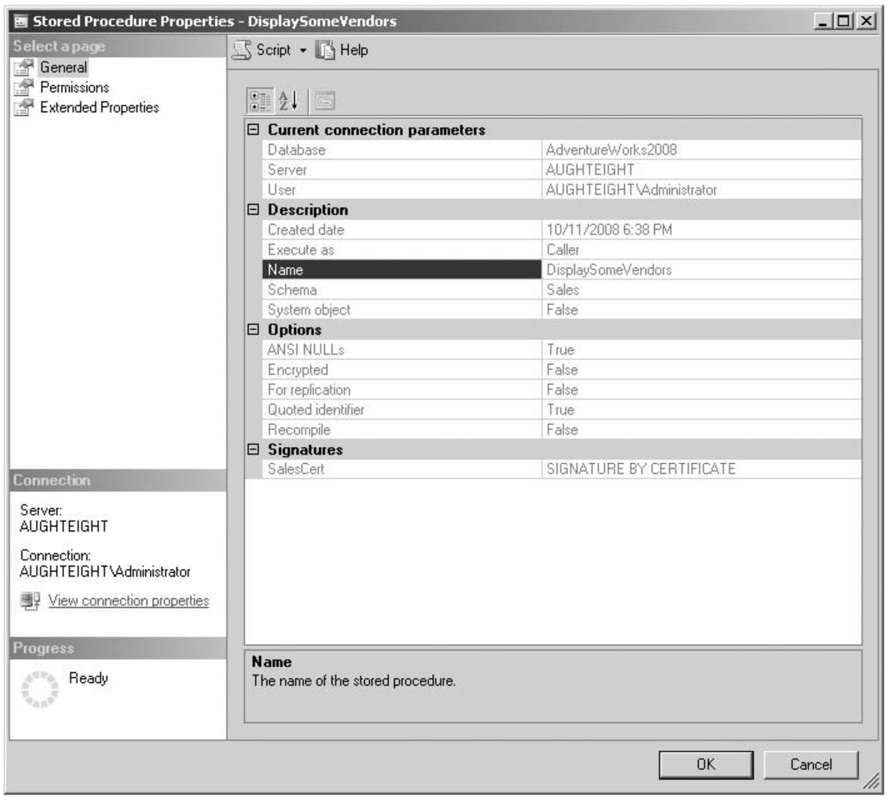
You can use both asymmetric keys and digital certificates to sign stored procedures, functions, or DML triggers in SQL Server. The following code creates a simple stored procedure called `Sales.DisplaySomeVendors`. You can then add a signature to that stored procedure using the `SalesCert` certificate from earlier. The private key will need to be decrypted to digitally sign the stored procedure.

```
CREATE PROCEDURE Sales.DisplaySomeVendors AS
     SELECT TOP (20) * FROM Purchasing.Vendor;
GO

USE AdventureWorks2008;
ADD SIGNATURE TO  Sales.DisplaySomeVendors
    BY CERTIFICATE SalesCert WITH PASSWORD = 'P@ssw0rd';
GO
```

If you look at the properties of the stored procedure, you can now see that the stored procedure has been digitally signed, and it was signed by the `SalesCert` certificate (see Figure 6-15). You can also query the `sys.crypt_properties` catalog view. This view will show any objects that have been digitally signed. In the next example, you will query the `sys.crypt_properties` view to see the digital signature assigned to the `Sales.DisplaySomeVendors` stored procedure. Then you can alter the procedure, query the view again, and note that the procedure is no longer digitally signed.

```
SELECT * FROM sys.crypt_properties
GO
ALTER PROCEDURE Sales.DisplaySomeVendors AS
 SELECT TOP (10) * FROM Purchasing.Vendor
GO
SELECT * FROM sys.crypt_properties
```


Stored Procedure Properties - DisplaySomeVendors

Current connection parameters
Database AdventureWorks2008
Server AUGHTEIGHT
User AUGHTEIGHT\Administrator

Description
Created date 10/11/2008 6:38 PM
Execute as Caller
Name DisplaySomeVendors
Schema Sales
System object False

Options
ANSI NULLs True
Encrypted False
For replication False
Quoted identifier True
Recompile False

Signatures
SalesCert SIGNATURE BY CERTIFICATE

Connection
Server: AUGHTEIGHT
Connection: AUGHTEIGHT\Administrator

Progress
Ready

Name
The name of the stored procedure.

Figure 6-15: Digital signature.

Best Practices

Like any other application or server product, there are a few guidelines you should follow to help increase the level of security in place. Remember that you will never be able to plan for and protect against every possible threat, but you can make it more difficult for malicious users to gain access to your data.

- **Use Strong Passwords** — As mentioned earlier in this chapter, you should take advantage of the password policies and require users to create complex passwords that get changed regularly. You should educate your users about the importance of strong passwords. While password policy enforcement for SQL Logins is managed at the server, you should provide an application or tool that allows users a way to change their passwords and be notified when their passwords are about to expire.
- **No One Should Log on as `sa`** — The `sa` account should rarely (if ever) log in. To provide more accurate auditing information, users should be forced to use their own logins (or log in through the membership in a group) in order to track what users are performing which actions. If everyone has the `sa` password and everyone is able to log in as that account, nothing would stop them from being able to steal or destroy your data. You wouldn't be able to hold that person accountable, because you may not know who that person is!
- **Use Least-Privilege Accounts for SQL Services** — Apply the principle of least privilege, and use accounts that have exactly the rights and permissions needed by the services, and nothing else. While it might be tempting to make the SQL Server account or the SQL Server Agent account a member of an administrative group, it is not necessary. Identify what resources outside of the SQL Server each of these accounts will be interacting with, and assign only the required permissions.
- **Audit Principals Regularly** — A diligent administrator will know what accounts have been created and who is responsible for these accounts, and identify what steps must be taken to disable or remove superfluous accounts.
- **Disable or Remove Any Unused Network Protocols** — In the SQL Configuration Manager, you have the ability to enable or disable protocols used by the SQL Server. Additionally, consider disabling the NetBIOS protocol for your network adapter if NetBIOS will not be used by your server or applications.
- **Use On-the-Wire Encryption to Protect Your Data in Transit** — It's not enough for you to protect the data while it sits idly on the server. As a database administrator, you should use technologies like Secure Sockets Layer (SSL) encryption and Internet Protocol Security (IPSec) to protect the data while it's moving from client to server, server to client, or server to server.
- **Do Not Place the SQL Server in a Location with Poor Physical Security** — There is a well-known article published by the Microsoft Security Response Center known as the ''10 Immutable Laws of Security.'' The first law dictates that if a malicious user has physical access to your computer, it's no longer your computer. Unless you can provide the means to control access to the hardware, your data can easily be stolen, compromised, damaged, or destroyed. Hardware locks, secure server rooms, and security personnel can all be instrumental in helping to protect your data.
- **Minimize the Visibility of the Server** — SQL Servers should never be publicly available. The Slammer worm should never have been a problem, had application architects and database administrators taken the necessary precautions to protect against that type of attack. Slammer was able to propagate so much, so fast, because few organizations recognized the harm in publishing SQL connectivity through their firewalls. A well-designed database application will use a robust and secure front-end, minimizing the exposure to the Database Engine.
- **Remove or Disable Unnecessary Services and Applications** — You should minimize the attack surface of your SQL Server as much as possible by turning off services and features that will not be used. Typically, it's a good idea to avoid running other services such as IIS, Active Directory, and Exchange on the same machine as SQL. Each one of these services can be a potential entry

point for a malicious user to exploit, thereby granting the user access to your data. Because SQL Server Reporting Services no longer requires IIS, this can help reduce the attack surface of your system.

- **Use Windows Authentication Whenever Possible** — Windows and Kerberos authentication are inherently more secure than SQL Authentication, but this is a design decision that you, your application developers, and security team must address.
- **Do Not Use Column Encryption on Frequently Searched Columns** — Encrypting frequently accessed or searched columns may cause more problems than it solves. If encrypting a column is the best, or only, way to protect the data, make sure that the performance impact is tested before implementing encryption in production.
- **Use TDE to Protect Data at Rest** — Encrypting the database and transaction log files can reduce the likelihood that someone can copy your data files and walk away with sensitive business data.
- **Always Back up Data Encryption Keys** — This is probably pretty self-explanatory, but make sure that any of the keys you use to back up your data, or other encryption keys, are safely and securely backed up. Test your backup and recovery strategy, as well.
- **Understand Your Role in the Company's Security Policy** — Most organizations have a documented security policy that defines acceptable use for the network and expectations for server or service behavior. As a database administrator, your responsibilities to configure and secure your servers may be documented as part of the overall security policy. What is expected of you and of your servers must be unambiguous. Your liabilities should also be clearly stated.

Summary

In this chapter, you learned about many of the security features available to you in SQL Server 2008. You should have a good understanding of the way security is applied to SQL Server from the top down, including:

- How to configure the different authentication modes
- How to create and manage server and database principals
- How to assign and control permissions
- How to protect your data on the server

You should also be able to apply some of the best practices discussed in this chapter to your own environments. Remember that you will never have a server that is 100 percent secure, and you should never be overconfident of your security design, because complacency leads to sloppiness, which leads to *ginormous* holes in your security design. But having read this chapter, you should feel confident in implementing the security mechanisms covered.

In Chapter 7, you will learn about creating and managing SQL endpoints and how you can enable access to database resources using a variety of connectivity methods.