1

A Security Roadmap

In many ways, securing SQL Server reminds me of paintings by Monet I saw at the Museum of Fine Art in Boston years ago. When you stand very close to Monet's paintings, all you see is little dots of color. It is only when you stand back that you see how the dots converge into a complete picture. Obviously, Monet had to focus on where he placed each spot of paint, but it is equally obvious that he knew where those daubs of paint were going to go before he started painting. For a database server, the daubs of paint might be a user, or a permission, or a piece of data, and the picture they form shows how they all relate to each other, and how they fulfill the primary goal of giving people no more and no less than the rights they need to accomplish their tasks. Much like the painting, we need to focus on where we will assign permissions, but we also need to have the big picture in mind before we start.

Quite often, people are overwhelmed at the sheer number of details to be managed when making sure that database users get the permissions they deserve, and do not get permissions they do not deserve. Let's face it – securing SQL Server is not a simple task. The process starts by trying to determine the identity of a user who wants to log in. Then SQL Server has to decide whether the user has permission to perform a very large list of activities at the server level. Finally, SQL Server has to decide whether the user can access a database, what identity he will have within that database, and what he can do with the data stored there. To add to the complexity, the user could be logging in with a Windows account instead of an account managed by SQL Server and, in SQL Server 7.0 and 2000, he could receive both server and database permissions by being a member of a Windows group. If you look at each individual piece of the process to the exclusion of the others, providing appropriate access to data does seem to be easy; but, when you put all the pieces together, the total picture can be quite intimidating.

Fortunately, you do not have to be a genius like Monet to learn to combine all those individual pieces into a coherent, understandable, and manageable security plan. Part of the learning process is to develop an understanding of which things you need to use, and which things can be left out. Just as Monet did not use every color available in a single painting, so are you not required to use every feature SQL Server offers for securing data. SQL Server is very flexible because it is used in many distinctly different environments. A technique that is appropriate for one environment will often simply not work in a different one; therefore, my goal for this book is to teach you how to evaluate the strengths and weaknesses of the different ways of securing data for your particular environment.

Even though securing a server may remind me of Monet's paintings, our tools will consist not of brushes and paints, but of accounts, passwords, and permissions. Before we move on to other chapters where we dig into the details of how SQL Server implements security, let's look at what is available to help us allow the good people in and keep the bad people out.

Authentication and Authorization

Every discussion of security concerns the twin processes of **Authentication** and **Authorization**. Authentication refers to the process of identifying a user, while authorization refers to the process of determining what that user can do. For SQL Server, authentication occurs both during initial login, and each time a user attempts to use a database for the first time. Authorization occurs every time a user attempts to perform any operation within a database. Authorization will also come into play any time a user attempts to change SQL Server's configuration, use a system stored procedure, make changes to database configurations, and so on.

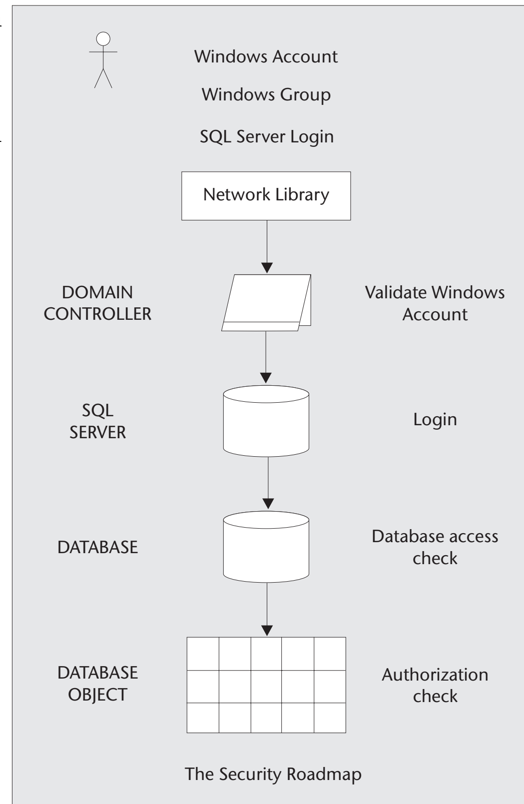
For authentication, which we will cover in Chapter 2, there are five server scenarios that are possible with SQL Server 6.5, 7.0, and 2000, running on Windows NT and 2000:

- SQL Server 6.5 on Windows NT.
- SQL Server 7.0 on Windows NT.
- SQL Server 2000 on Windows NT.
- SQL Server 7 on Windows 2000.
- SQL Server 2000 on Windows 2000.

Fortunately, all but the last scenario use basically the same mechanisms to authenticate users. It is only when we look at SQL Server 2000 running on Windows 2000 that we need to expand our discussion to encompass the new security features in Windows 2000.

Authorization is easier to cover because there is no operating system based difference in the authorization process between Windows NT and Windows 2000. However SQL Server 6.5 has important differences from SQL Server 7.0 & 2000, so we will cover them in separate chapters.

To get us started, let's create a security roadmap to help us keep track of where we must make decisions about which feature to use. This is outlined in the diagram opposite:



The Security Roadmap

This is a picture I keep in my head when I'm troubleshooting server access problems, or trying to determine what permissions a user has in a database. Each section represents a different place where you can control access. In the next few sections, we will put all the security mechanisms into the context of this picture, so that hopefully at the end of this chapter you will have a sense of where each part fits into the overall scheme of managing SQL Server security.

Options for Authentication

The place to start is authentication. When I started teaching classes on SQL Server, I discovered that most of my students did not realize that all interaction with SQL Server happens through a client application. As a service, SQL Server runs without a user interface. In fact, the only way you can change the server's settings without using a client application is by setting command line parameters and/or registry settings.

Client authentication, therefore, is a critical piece of any security plan. Administrators usually do not need to worry about authentication because they are using Windows NT accounts or SQL Server accounts that grant them complete control over the system; but users are not – and should not – be so fortunate. That means your first decision, when designing a security plan, will be how your system's users will validate their login information.

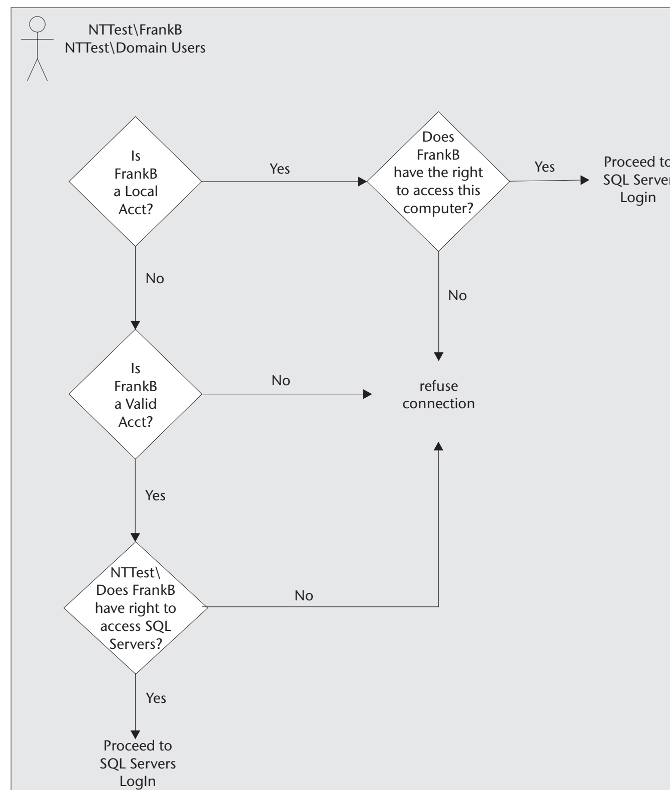
SQL Server 6.5 and 7.0 have two ways to authenticate logins: Windows NT authentication and SQL Server authentication; and SQL Server 2000 adds Kerberos and Active Directory authentication. Chapter 2 will cover the details of how authentication works; so for now, let's just concentrate on how these authentication modes fit.

Windows Authentication

To understand Windows NT/2000 authentication, you have to understand how Windows NT represents a user's permissions within the system. When a user logs in, whether they are sitting at the computer itself or are connecting to the system across the network, Windows NT creates an **access token**, which contains the user's **Security ID** (**SID**) and a list of all the groups (both local and global) of which the user is a member. Each time a user attempts to open a protected resource, Windows NT compares the SID and group memberships in the access token to the **Access Control List**, which lists approved users for that resource.

User rights play a role here as well. For a Windows local login, a user must have the 'Log on locally' user right, while network users require the 'Access this computer from the network' user right. Those rights, of course, can be granted to any group of which the user is a member, including the `Everyone` local group.

The diagram below illustrates how a client authenticates using Windows NT authentication:



In the client documentation, this kind of authentication is called a **trusted connection**, primarily because the only kinds of clients that can use it are the ones that Windows NT *trusts* – that is, other Windows clients. The main difference between Windows NT authentication and SQL Server authentication is that the Windows-based client knows how to encrypt the login credentials the way Windows expects, instead of sending the account and password across the network in clear-text, as is the case with SQL Server authentication.

The process begins with the client application's attempt to make a connection to the server. We will cover how the client finds the server a little later when we discuss the SQL Server Network Libraries in Chapter 2. The Network Library used on the client affects both the way the client finds the server, and the way the client sends data to the server. For the purposes of the discussion in this section, you can just assume that the client can find the server.

The process of sending data back and forth to the operating system is called **Interprocess Communication**, or **IPC** for short. IPC started as a way for one application to send data to another application on the same Windows NT machine, but it was expanded to allow an application to send data to another application on an entirely different computer across the network. In the process, the architects of Windows NT realized that because all clients must authenticate in Windows NT before they are allowed to do anything on the computer, IPC clients must have a way to authenticate when they attempt to connect across the network. Thus, at startup, Windows NT or 2000 creates a hidden system share named `IPC$`.

> *If you want to know more about IPC mechanisms and how Windows NT manages application and network security, you should consult* Inside Windows NT *by Helen Custer (Microsoft Press, ISBN: 155615481X),* Advanced Windows, 3rd Edition *by Jeffrey Richter (Microsoft Press, ISBN: 1572315482), and* Programming Windows, The Definitive Guide to the Win32 API *by Charles Petzold (Microsoft Press, ISBN: 157231995X). The first one is a must read for all Windows NT administrators, and the latter two are must-reads for all Windows NT programmers. Windows NT administrators who know a little about programming can benefit from reading* Advanced Windows *too.*

All clients wanting to log in to Windows NT attempt to connect to `IPC$` with a mechanism that is identical to connecting to a shared directory. Because it is a shared resource, attempting to connect to `IPC$` triggers a login process on the server. In the process, the client operating system sends its security credentials to the server so that Windows can build an access token.

The account can be either in the server's local security storage, or in the domain's security storage. If the user does not provide a domain account or if the domain provided is not one recognized by the server, Windows consults its local security storage to see if the account and password can be found there. If the account is a domain account, then Windows makes a connection to a domain controller and asks it to validate the account and password. If the domain controller approves the credentials, it sends back the list of domain global and local groups of which the account is a member. In both cases, the operating system always checks the local security storage and adds the local groups that have the login account as a member. In the case of a domain account, this check is in addition to the check of the domain security database.

After authenticating the user's Windows account, SQL Server receives a complete list of security identifiers for both the user's Windows NT/2000 account and the local and global groups of which he is a member. The net result is that a user can gain access to SQL Server through one of the following:

- His personal account.
- The local operating system's local groups (in the case of SQL Server running on a member server).
- The domain's local groups (only in certain, special cases).
- The domain global groups.

Once the operating system compiles the list of SIDs, SQL Server takes over the authentication process using a table containing login account information. Chapter 2 goes into detail on how SQL Server determines login privileges for versions 6.5, 7.0, and 2000.

Managing Server Access Using Windows NT Groups

This is a good point to mention that Windows authentication is not limited to just *user* accounts in SQL Server 7.0 and 2000; both local and global groups can be granted login permission as well. In this case, instead of storing the SID for an individual account, SQL Server stores the SID for the group.

The effect of this approach is that you can manage access to your server through Windows NT or 2000 domain global groups, or by adding Windows users to local groups on the SQL Server itself. If the Windows NT or 2000 account administrators are also the SQL Server administrators, and if it makes sense to manage your SQL Server users' access at the domain level, then this method greatly simplifies server access management. Rather than creating tens or even hundreds of login accounts, you can create several groups that represent the different groups of people who will be using the system and place members in the groups at the domain server.

If you need to deny access to a specific member of a Windows NT group, you have two options. First, you can deny access to the user's Windows NT account explicitly. Second, you can create a single domain group, deny that group access to SQL Server, and place any users who may not access the server in that group. Because having only one of their access token SID's denied means the user cannot log in, you need only one group for the entire organization.

From this point on, normal SQL Server permissions checking takes over. In Chapter 4, we will see that SQL Server 7.0 and 2000 use the contents of the access token when checking permissions at the server and database level, but other than that, Windows is out of the picture. Now, let's turn our attention to what happens when the client logs in with an account maintained by SQL Server.

SQL Server Authentication

You can think of SQL Server authentication as the lowest common denominator for authentication, because it supports logins from all clients, no matter what operating system they use. SQL Server authentication supports connections from clients that are:

- Running all versions of Windows.
- Using the TCP/IP network protocol, for example, Unix or Novell NetWare clients.
- Using the AppleTalk network protocol, for example, the iMac.
- Using the Banyan Vines network protocol.

The differences between SQL Server authentication and Windows NT authentication are:

- The request for login comes directly to SQL Server.
- SQL Server maintains the internal list of permitted logins, and the login request does not use Windows NT password encryption.

Once logged in, granting permissions is generally the same for both SQL Server and Windows authenticated logins. In SQL Server 6.5, there are no differences in the way you assign permissions to either type of login. (We will cover all the options for assigning permissions in SQL Server 6.5 in Chapter 3.) In SQL Server 7.0 and 2000, the only difference between SQL Server and Windows authenticated logins is that the SQL Server login does not carry any Windows NT group or account information with it, which means that it cannot gain any additional permissions granted to Windows groups. Instead, it will gain its permissions from system and database roles, which we will discuss in Chapter 4.

Kerberos and Active Directory Authentication

SQL Server 2000 adds its own additions to the list of authentication methods for handling Windows 2000 clients. For Windows 95/98 and Windows NT clients, and for Windows 2000 clients connecting to SQL Server 2000 running on Windows NT, authentication in SQL Server 2000 is the same as it is in SQL Server 7.0. For SQL Server 2000 running on Windows 2000, however, you will have the option of authenticating through the Active Directory and/or using the **Kerberos** authentication protocol.

Chapter 5 covers the details, but the main difference is that Windows 2000 uses Kerberos security through the Active Directory service. For the most part, login validation through Active Directory is just a variation on the theme used in Windows NT. Clients present credentials, Windows 2000 validates the credentials, and SQL Server uses the information returned by the Active Directory to match a Windows account to an entry in SQL Server 2000's list of valid logins. Once validated, all the security mechanisms built into SQL Server 7.0/2000 take over, just as they do when you have a Windows NT authenticated login.

The main difference between the authentication protocol used by Windows NT and Kerberos is that the Kerberos protocol authenticates both the client's identity and the server's identity. The client is assured they are communicating with the correct server, and the server is assured that the domain controller has authenticated the client's identity. Kerberos also verifies that the data has not been modified in transit by a third party, which can be useful in situations where the data needs to travel over the Internet.

Options for Authorization

Once the authentication process finishes, SQL Server takes control of authorizing users' access to objects and data in the system. SQL Server 6.5, 7.0, and 2000 all have similar sets of permissions; but 7.0 and 2000 differ greatly from 6.5 in the way you can assign them. SQL Server 6.5 can assign permissions to individual users and database groups, while SQL Server 7.0 and 2000 can grant permissions to Windows authenticated logins based on their individual account, or the groups of which they are members. SQL Server authenticated logins in SQL Server 7.0 and 2000 can be granted permissions based on the login ID or on membership in database roles, which function like Windows groups. Chapter 3 will cover all the ways we can assign permissions and the list of available permissions for SQL Server 6.5, and Chapter 4 will cover the same topics for SQL Server 7.0 and 2000.

Besides login authentication, all administrators need to create databases, make changes to the server configuration, perform basic maintenance on databases (like making backups and restoring corrupted databases), and manage user access to databases. In SQL Server 6.5, the only account that really had the permissions to perform all these tasks was the `sa` account, which is why so many administrators use that account as their sole server login. SQL Server 7.0 and 2000 have the concept that administrators should be able to divide responsibilities among several people, and also, therefore, logins. The `sa` account still exists, but administrators can now gain the same rights and privileges through their own accounts instead of sharing one account and password.

Server Roles in 7.0 and 2000

Version 7.0 of SQL Server is the first that treats permissions as something to be granted based on someone's role instead of on discrete rights for individual users. SQL Server 7.0 and 2000 have built-in **server roles** that exemplify this change because some permissions cannot be granted to individuals at all. In order to gain the authorizations listed, the user must be a member of one of the built-in roles, which are listed below:

Chapter 2 covers the details of which permissions go with each role, so here we'll just have a quick look at each role to get an idea of where it will fit into our security picture:

- **`sysadmin`** – This is the system administrator role, as you would expect. The primary distinction between versions 6.5 and 7.0 is that the `sa` account gets its permissions by virtue of being a member of this role rather than being a specially recognized account, as it was in version 6.5. The benefit of this change is that now users can use either a Windows NT account or group or a SQL Server login to gain the privileges of the `sa` account instead of sharing an account and password.

- **serveradmin** – This role is for administrators who will be managing SQL Server itself, but not databases or objects in it. Membership in this role gives the user permission to perform tasks such as changing memory settings, shutting down the server, and setting options on tables that affect the server – for example, DBCC PINTABLE. It does not, however, grant permission to view or modify data in any database, so this role is perfect for an administrator who should not have complete control over sensitive data.
- **setupadmin** – This is a special-purpose role for administrators who need to configure settings for remote servers or run a stored procedure at startup. It has limited capabilities and is normally used in combination with other roles, such as serveradmin or processadmin, to weave the permissions together for an administrator who should have fewer rights than those granted to sysadmin.
- **securityadmin** – This is self-explanatory. This role has permissions to manage access to the server and to databases. Permissions include managing logins, setting up login information for linked servers, and granting access to databases. Though it does not innately have rights to any database, a member of this role can grant themselves access to the database, but they cannot grant permissions to objects in the database through membership in this role. Also, in case you were worried, a member of securityadmin cannot assign themselves to the sysadmin role.
- **processadmin** – This has one permission: executing the KILL command. The only use I can think of for this role is for technical support personnel or assistant administrators who need to halt processes on a regular basis. In a normal environment, killing a process should be a rare event, so this role should get little use.
- **dbcreator** – This role does just what the name implies; it allows its members to create databases and alter them. Remember that the user who creates a database is automatically mapped to the dbo database user account and is the first member of the db_owner database role (these are discussed in detail towards the end of the chapter). Further, members in this role will need to understand the basics of how SQL Server stores data so that they can make good decisions about where to put files, file growth, and when to use file groups. This role is well designed for development staff who need full control over a database during application development.
- **diskadmin** – This role is a relic from the days of SQL Server 6.5. Its permissions allow members to manage disk devices, which SQL Server 7.0 does not use. Because the DISK INIT command no longer has any purpose in SQL Server 7.0, this role is really only useful for working with devices created in SQL Server 6.5; otherwise, it is not useful.
- **bulkadmin** – This role is new in SQL Server 2000, and its sole purpose is to grant permission to execute the BULK INSERT command. This can be useful in scenarios such as data warehousing, where data needs to be inserted into tables in large quantities. Since a scheduled script or a Data Transformation Service (DTS) package usually handle BULK INSERT operations, a likely choice for membership in this role is the account used by the service executing the script.

From these short descriptions, you can probably draw the conclusion that the sysadmin, securityadmin, and dbcreator roles will be the most useful. In terms of how you can use them to build a security plan, it makes sense to limit the scope of what an administrator can do to just the permissions he needs. Unlike Version 6.5, SQL Server 7.0 and 2000's server roles permit the creation of different levels of administrator privileges. You can:

- Reserve the sysadmin role, with its complete control over the server, for senior-level, experienced administrators.
- Combine securityadmin, setupadmin, processadmin, and serveradmin to create a set of permissions for more junior administrators.
- Grant control to databases on an individual basis through the dbcreator role.

Database User Accounts

Typically, the vast majority of users will not use server roles at all. Instead, they will gain access through permissions assigned in the database itself. In order that permissions may be handled at the level of an individual database, each database maintains its own list of database user accounts. These accounts are completely separate from SQL Server login accounts, and the process of granting access to a database is separate from the process of granting access to the server itself.

SQL Server manages user accounts with a table named sysusers. This table identifies each user with a **unique user identifier** (**UID**), and each UID has a direct mapping to a **server user identifier** (**SUID**) in the syslogins table in SQL Server 6.5, or a **security identifier** (**SID**) from the sysxlogins table in SQL Server 7.0 and 2000. SQL Server 6.5 has a concept called **aliasing**, which is simply the practice of having multiple login identifiers mapped to a single UID (see Chapter 3 for more details). SQL Server 7.0 supports aliasing for backward compatibility, but SQL Server 2000 no longer supports that function. Chapter 4 covers how SQL Server 7.0 and 2000 map login IDS to database user accounts.

Shown below is a sample output from the sysusers table:

uid	status	name	sid	password	hasdbaccess	islogin	isntuser
5	14	TimB	0x010500000000000005150...	NULL	1	1	1
12	2	FredJ	0xFD40A377E973F140AAB...	NULL	1	1	0

Essentially, the mapping of a SUID to a UID in SQL Server 6.5, and a SID to a UID in SQL Server 7.0 and 2000, is a kind of transparent authentication of the user's access to a database. Once granted, access to the database becomes a seamless part of the overall login process, and the users never need to know that another authentication process occurs each time they move from one database to another.

It may seem unusual to discuss user authentication in a section devoted to authorization; but, very simply, if a user has a UID in a database, they have access. Whether or not they have permissions depends on several other criteria, but at the least they have the *potential* to access the data in the database. Conversely, removing a UID from the database denies all access to its data. This means that user accounts are a kind of authentication that occurs each time a user attempts to access data.

Further, this process occurs even if a command references objects in databases other than the current one. Finally, because database access depends on the SUID from syslogins or the SID from sysxlogins, denying login privileges or removing a login altogether effectively eliminates access to all databases at once. This can be an effective way to seal security breaches quickly, without damaging the overall permissions structure in a database.

The discussion of database access is confused further by SQL Server 7.0's addition of Windows NT groups as a means of gaining login access to the server. Because users can log in by virtue of their membership in a group, and not their own personal accounts, their personal SID does not need to be recorded anywhere in SQL Server so long as the SID corresponding to their group is there. In this case, the user's total database permissions will be those granted to their Windows NT groups that have access to the database.

If that was not enough, even more confusion arises out of the fact that the Windows NT groups in the database do not have to be the same groups a user uses to log in to the server.

Database Roles

Like the default server roles, there are default database roles that have permissions that cannot be granted independently. Unlike server roles, some of the permissions in the sysusers table can be granted directly to a given user. Additionally, you can create your own roles and assign permissions to them if the built-in database roles do not have the proper combination of permissions. The only thing you cannot do is place a role within another role, much like a Windows NT local group cannot be a member of another local group.

The following is a list of all the built-in database roles:

- **db_owner** – This role is mostly self-explanatory. Members of this group gain all the rights and privileges of the database owner, which is to say just about complete control over everything in the database. **Database object owners** (**dboo** – yes, that is the acronym Microsoft uses) can deny database owners some types of access, but the owners can always take ownership of the object and grant whatever permissions they want. Other than that minor inconvenience, members of db_owner have no limitations on what they can do with the objects in the database.
- **public** – This is the one role to which you need never grant membership, since all users automatically have membership just by being listed in sysusers. The main security concern is that anyone granted access to the database also automatically gains the permissions granted to public. As we will see later, you will end up granting to the public role only the permissions that everyone in the database should have. With careful planning, you can minimize the risk. Alternatively, you can create a user-defined role, granting the permissions you would normally grant to public, and not granting or revoking any permissions to public itself. Besides this one concern, public can be a very useful role for decreasing the amount of work you must do to grant permissions in a database with many users.

- **db_accessadmin** – The 'access' part of db_accessadmin refers to database access, and members of this role can add and remove database user accounts.
- **db_securityadmin** – This role controls the management of user-defined roles in the database. Members can create and remove user-defined roles, as well as manage the users in those roles. Neither this role, nor db_accessadmin, can grant its members any permissions to the data itself, although db_securityadmin members can, of course, add themselves to any user-defined role and gain access through its permissions.
- **db_ddladmin** – This is the workhorse of the built-in roles. Members can execute the GRANT, DENY, and REVOKE statements as well as create foreign key references between tables. They can also perform some simple operations on objects, such as renaming them or changing their owners. Note that members of this role cannot actually add themselves to other roles; they can only grant permissions to those roles.
- **db_backupoperator** – This role has permissions that allow its members to perform database backups and the DBCC commands that check the integrity of the database before a backup starts. Interestingly, they cannot restore a backup; that privilege is reserved for members of the sysadmin role. Also, members cannot view the data they are backing up either, so you can safely grant membership in this role to just about anyone who knows how to backup the data.
- **db_datareader** – This role has SELECT permissions on all tables in the database.
- **db_datawriter** – This has INSERT, UPDATE, and DELETE permissions on all tables.
- **db_denydatareader** and **db_denydatawriter** are the reverse of their counterparts and specifically deny SELECT or INSERT, UPDATE, and DELETE permissions, respectively, to their members. The primary benefit of these roles is that they allow you to restrict access to the data to users who might otherwise have access they should not have by virtue of their other role memberships.

Though not strictly a database role, the database owner account, named dbo, has special privileges in a database much like the sa account has in the server as a whole. Like sa and the sysadmin role, dbo exists separately from the db_owner role. Unlike sa, dbo does not gain all its permissions from membership in the db_owner role; it has its own permissions outside the database role structure. Accordingly, only dbo may add members to the db_owner role, and no member of that role may remove dbo from it or grant ownership of the database to another user.

This unique status in the database makes dbo second only to members of the sysadmin role in terms of what they can do within SQL Server. The logical conclusion is that dbo in a production environment should be someone who has a high degree of both SQL Server expertise and trust within the organization. In most cases, it should be one of the senior-level server administrators because of the amount of damage a mistake made out of ignorance can cause.

Database Owner Rights in SQL Server 6.5

Database roles are not part of SQL Server 6.5. The permissions listed above belong only to the `sa` account and the `dbo` database account. The main reason SQL Server 6.5 allows multiple login accounts to share database user accounts, is to allow multiple people to share the permissions granted only to the `dbo` user account. In Chapter 3, we will look at how this many-to-one mapping works; then, in Chapter 4, we will look at how SQL Server 7.0 and 2000 database roles offer a more manageable mechanism for permitting multiple users to perform common database operations.

Summary

So far, we have just hit the highlights of SQL Server security. In general, there are three major points at which you must make significant decisions about how to authenticate users and authorize access:

- Firstly, you must decide how you will authenticate server logins, mostly because Windows authentication offers some significant advantages in overall ease of management if you have a Windows-only network.
- Secondly, you must decide how users will gain access to the databases on your server. Once again, Windows authentication presents some advantages, but it also presents some challenges due to the fact that Windows NT groups can be users in the database.
- Thirdly, you must decide how to grant permissions to database users. You can choose to grant permissions to individual users, Windows NT groups, and/or user-defined roles; or place users in the built-in database roles or a combination of both.

In many ways, deciding how to assign permissions can be the most difficult decision. In Chapters 3 & 4, we will use some examples to help us work through the rather complex interactions between database accounts and permissions and Windows authenticated logins. In Chapter 5, we will look at ways to secure different application architectures.

After we lay the foundation in the next three chapters of how SQL Server's security mechanisms work, we will finish out the book with a look at practical recommendations for securing applications, Data Transformation Services, replication, and SQL Server CE. Just remember, you do not have to use every feature SQL Server offers; you can simplify matters by being selective. If you keep things simple, you are less likely to grant access that users should not have, or deny access they should have. What you are looking for is that perfect solution in which users have exactly the permissions they need to do their jobs, and not one bit more or less. When you find it, you have succeeded in your mission to keep your data secure.