6
Logon Scripts

When a user logs on to an NT machine, a *logon script* can be executed. This script enables you, as the network administrator, to dynamically customize the user's environment. Such flexibility has been a part of computer systems since the beginning. UNIX shell scripts have, for example, always provided facilities to customize the user's environment during the logon process.

The name *logon script* is a bit of a misnomer because it does not really have to be a script per se. It can be any executable program, such as an application, a batch file, a Java application, a Visual Basic script, or even a Perl script. Traditionally, logon scripts have been batch files.

Because Windows NT enables different users to log on to different machines, setting up a computer for only one user is difficult. Logon scripts provide the mechanism required to guarantee that drives can be mapped to appropriate network servers, printers can be reconnected to local printing devices, environment variables can be updated, and so on.

How Logon Scripts Work

When a user attempts to log on to a Win32 domain, her machine contacts a *domain controller (DC)*. The DC holds a copy of the domain's user database. This database contains each user account's configuration information. The DC authenticates the user and (if her password is correct), she is be permitted to log on to the domain.

A DC that services a logon request is known as the logon session's *logon server*. If a network has several DCs, a request to log on may be serviced by different DCs at different times. Which machine services your logon request is based, in part, on which controller is least busy.

Note

Logon scripts run only when a user interactively logs in to a domain (such as when logging on using the keyboard or via a terminal server client). If the user logs in over a network (as in mapping a drive letter to a server) or if the user's account is impersonated (such as when a Web server authenticates a user account), the logon script is not executed. ◆

After a user successfully logs in to a domain, the machine attempts to run the user's logon script. The logon script is always run from the logon server's NetLogon shared directory. For example, if a user account is authenticated by the DC called LogonServer, and the user's logon script is MyLogonScript.exe, Win32 attempts to run the following:

```
\\LogonServer\NetLogon\MyLogonScript.exe
```

This means that all user account logon scripts must be a path relative to a DC's NetLogon shared directory. If the file does not exist or is inaccessible (maybe because of security permissions), the attempt to execute the logon script simply fails, but no error message is displayed to the user.

Tip

If you have many DCs on your network, maintaining up-to-date copies of any logon scripts on each controller is important. This is what the Replication Service was designed for. It enables you to designate one controller, typically the primary domain controller (PDC), as the logon script repository.
Any script updates are made to scripts in the repository. Scripts that have been modified are then replicated to all the other DCs. ◆

A logon script does not really have to be a script at all. It can be an executable program, a batch file, or a script such as Perl, Java, Python, or Visual Basic. For example, a logon script can be configured to be a Perl script like this:

```
UserScripts\MyPerlLogonScript.pl
```

This example assumes, of course, that the machine the user logs on to has the .pl extension associated with Perl (refer to Chapter 1, "Perl and the Admin," for details).

In a user account, the logon script attribute is really just an arbitrary string. This means that an account can be configured to be more than just a path to a logon script. It can contain parameters to be passed in as well. For example, an account can be configured to pass in the name of the user as well as the local computer name (using environment variables):

```
UserScripts\MyLogonScriptProgram.exe %USERNAME% %COMPUTERNAME%
```

Because logon script attributes are arbitrary strings, you can use them to run Perl from the logon server and pass in the name of the logon script. For example, say that you have installed Perl in a Perl directory under the NetLogon share on all your DCs. You can configure your user accounts to have this logon script:

```
perl\bin\perl.exe %LOGONSERVER%\NetLogon\UserScripts\MyLogonScript.pl
```

Of course, when this script is run, the logon server and NetLogon share are prepended, so Win32 actually executes the following:

```
\\MyLogonServer\NetLogon\perl\bin\perl.exe
\\MyLogonServer\NetLogon\UserScripts\MyLogonScript.pl
```

In this case, the `%LOGONSERVER%` environment variable is replaced with the name of the DC (`\\MyLogonServer`) that is authenticating the user. By doing this, you don't need to install Perl on individual user machines but instead only on the DCs. Accessing Perl across the network in this manner does present some pitfalls, however. Refer to Chapter 1 for more information.

Configuring Logon Scripts

Logon scripts are configured as a part of a user account, enabling one user to have a logon script, while another user is configured with a different script. This is an ideal way to provide each user with a personalized experience. You can modify what logon script a user account is configured to use by using the User Manager program (on Windows NT) or the Computer Management MMC (on Windows 2000). Additionally, you can configure user accounts by using a scripting language such as Perl.

Example 6.1 illustrates how you can use Perl to discover what logon script is used by a user. Run the script, passing in any number of user account names. The script will display each user's logon script.

Example 6.1 *Displaying a User's Logon Script*

```
01. use Win32;
02. use Win32::AdminMisc;
03.
04. if( ! scalar @ARGV )
05. {
06.   my( $Script ) = ( $0 =~ /([^\\]*?)$/ );
07.   print << "EOT";
08.   Syntax
09.     perl $Script <User Account> [ <User Account 2> [ ... ] ]
10.
11.   Example:
12.     perl $Script administrator
```

continues ▶

Example 6.1 *continued*

```
13.     perl $Script somedomain\\administrator accounting\\joel
14. EOT
15.
16.   exit();
17. }
18.
19. foreach my $Account ( @ARGV )
20. {
21.   my %Info;
22.
23.   print "$Account logon script: ";
24.   if( Win32::AdminMisc::UserGetMiscAttributes( '', $Account, \%Info ) )
25.   {
26.     if( "" eq $Info{USER_SCRIPT_PATH} )
27.     {
28.       $Info{USER_SCRIPT_PATH} = "No logon script configured";
29.     }
30.     print "$Info{USER_SCRIPT_PATH}\n";
31.   }
32.   else
33.   {
34.     print Win32::FormatMessage( Win32::AdminMisc::GetError() );
35.   }
36. }
```

Displaying a user account's logon script is certainly useful, but so is being able to modify the account. Example 6.2 accepts the name of a user account and a logon script path. This code modifies the user account's logon script path.

Example 6.2 *Setting a User's Logon Script*

```
01. use Win32;
02. use Win32::AdminMisc;
03.
04. if( ! scalar @ARGV )
05. {
06.   my( $Script ) = ( $0 =~ /([^\\]*?)$/ );
07.   print << "EOT";
08.   Syntax:
09.     perl $Script <User Account> <Logon Script Path>
10.
11.   Example:
12.     perl $Script administrator scripts\\administrator.bat
13.     perl $Script accounting\\joel "genericlogon.pl"
14.
15.   To remove a logon script set the path to "" as in:
16.     perl $Script accounting\\joel ""
17. EOT
18.   exit();
```

```
19. }
20.
21. my( $Account, $Path ) = @ARGV;
22. if( "" eq $Path )
23. {
24.   print "Removing the logon script for the $Account account.\n";
25. }
26. else
27. {
28.   print "Configuring the $Account account with a logon script of:\n";
29.   print "$Path\n";
30. }
31. if( Win32::AdminMisc::UserSetMiscAttributes( '',
32.                                             $Account,
33.                                             USER_SCRIPT_PATH => $Path ) )
34. {
35.   print "Successful\n";
36. }
37. else
38. {
39.   print "An error occurred: ";
40.   print Win32::FormatMessage( Win32::AdminMisc::GetError() );
41. }
```

Modifying the User Environment

A logon script can be programmed to perform such a variety of activities that it is futile to consider discussing all of them. However, one of the more common activities—querying and modifying environment variables—is well worth covering.

Environment Variables

Windows NT was designed as a multiuser operating system. Therefore, it makes distinctions between user and system resources. One such resource is the list of environment variables.

A user's profile can contain environment variables that are defined by the user himself. For example, if the user JOEL adds an environment variable called INCLUDE, it will be there whenever the user logs on. If another user logs on to the same machine, she will see her own set of variables that may be different.

Additionally, if the users are configured to use roaming profiles, this set of environment variables follows each user from machine to machine. No matter what computer the user logs on to, the INCLUDE environment variable appears as he configured it.

User environment variables are separate from system environment variables. The system has its own set of variables that it uses from boot time until shutdown. This set includes such important variables as `PATH` and is set by either the system itself or a user with administrator privileges.

Typically, user variables override system variables. For example, if a system environment variable `INCLUDEPATH` is set to `c:\SystemInclude`, and a user who has a user environment variable `INCLUDEPATH` set to `c:\MyInclude` logs on, then any application that uses the `INCLUDEPATH` variable will see only the user version: `c:\MyInclude`. This can be a problem because it means that a user can use either the system variable value or the user variable value, but not both at the same time.

The workaround is to embed the system variable within the user variable. To continue the preceding example, if the user variable `INCLUDEPATH` is set to `%INCLUDEPATH%;c:\SystemInclude`, an application that uses this environment variable sees `INCLUDEPATH` as `c:\MyInclude;c:\SystemInclude`. Notice how the `%INCLUDEPATH%` expands as the system version of the variable. When you place a reference to the environment variable within the user variable, it inherits the system variable's value. This trick works only in that direction. A system variable cannot inherit from a user variable. If it did, this could result in redundant loops of variable inheritance. The moral here is that if a user variable refers to itself, it is really referring to the system environment variable by the same name.

As you already learned, a user variable will override a system variable by the same name. This is true for all but one—the `PATH` variable. This variable is special because it is required by the system. In this case, any user-defined `PATH` variable is appended to the system's version. Thus, if a user defines `PATH` to be `c:\temp` and the system's `PATH` is set to `c:\winnt;c:\winnt\system32`, the resulting `PATH` is `c:\winnt;c:\winnt\system32;c:\temp`. This allows such configurations as a user defining his personal `PATH` to point back to his home directory—for example, `\\server\home\joel\bin`. This enables the user to place tools and applications he uses often in one place so that he can seamlessly access them from anywhere on the LAN.

Perl and Environment Variables

Because a user variable can override a system variable, it is difficult for a Perl script to discover the version of the variable that the system administrator intended an application to use. Perl accesses environment variables by using the `%ENV` hash. It accesses only the set of variables exposed by the current environment. This environment, just like a DOS box, does not have a way to distinguish system versus user variables. Therefore, Perl is unable to see system variables if a user variable exists with the same name.

In Windows, each application has access to an *environment,* which is a block of memory allocated for a process that contains such information as the list of environment variables and their values. Each running process can have its very own copy of the environment. When a change to an environmental variable takes place (such as when a user uses the Control Panel to change a variable's value), all running programs are notified, and they can choose to reload the updated values into their environment. Therefore, a running Windows program should see such changes immediately. (Whether the program sees changes depends on how the application was designed. Some Windows programs may not see these changes immediately.) Console applications, on the other hand, typically do not listen for such messages. Therefore, legacy DOS applications, Perl, the DOS window, and other console-based applications do not see these changes until the console environment is terminated and re-created. This means that a Perl script must be stopped, the console window it runs in closed, a new console window opened, and the Perl script run again before the variable change is seen.

Console applications can update their environment when changes are detected, but most usually do not do so. Perl is one example of an application that does not listen for environment changes. Changes to this environment are performed only by a Perl script using the `%ENV` hash.

Querying Variables

Even though Perl cannot directly discern the difference between system and user environment variables using the `%ENV` hash, it can query system and user variables by relying on alternative methods. Example 6.3 shows how user and system environment variables can be queried independently using the `Win32::AdminMisc` extension. The script fetches the list of system and user environment variables into a hash (lines 5 and 14) and then displays them. When the script is run, the output shows which environment variables are embedded in other environment variables.

Example 6.4 is another script that makes use of a little-known function called `Win32::ExpandEnvironmentStrings()` that expands any references to environment variables in the string passed in. Any environment variables embedded within other variables are expanded. Comparing the output between these two examples is interesting.

Example 6.3 *Displaying System and User Environment Variables*

```
01. use Win32;
02. use Win32::AdminMisc;
03. $~ = VariableFormat;
04. print "System Variables:\n";
05. if( Win32::AdminMisc::GetEnvVar( \%System, ENV_SYSTEM ) )
```

continues ▶

Example 6.3 *continued*

```
{
    foreach $VarName ( sort( keys( %System ) ) )
    {
        $VarValue = $System{$VarName};
        write;
    }
}
print "\nUser Variables:\n";
if( Win32::AdminMisc::GetEnvVar( \%User, ENV_USER ) )
{
    foreach $VarName ( sort( keys( %User ) ) )
    {
        $VarValue = $User{$VarName};
        write;
    }
}

format VariableFormat =
  @<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  $VarName,               $VarValue
~                         ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                          $VarValue
~                         ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                          $VarValue
~                         ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                          $VarValue
~                         ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                          $VarValue
~                         ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                          $VarValue
~                         ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                          $VarValue
.
```

Example 6.4 *Displaying Expanded System and User Environment Variables*

```
use Win32;
use Win32::AdminMisc;
$~ = VariableFormat;
print "System Variables:\n";
if( Win32::AdminMisc::GetEnvVar( \%System, ENV_SYSTEM ) )
{
  foreach $VarName ( sort( keys( %System ) ) )
  {
    $VarValue = Win32::ExpandEnvironmentStrings( $System{$VarName} );
    write;
  }
}
print "\nUser Variables:\n";
if( Win32::AdminMisc::GetEnvVar( \%User, ENV_USER ) )
{
```

```
16.   foreach $VarName ( sort( keys( %User ) ) )
17.   {
18.     $VarValue = Win32::ExpandEnvironmentStrings( $User{$VarName} );
19.     write;
20.   }
21. }
22.
23. format VariableFormat =
24.   @<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
25.   $VarName,             $VarValue
26. ~                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
27.                         $VarValue
28. ~                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
29.                         $VarValue
30. ~                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
31.                         $VarValue
32. ~                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
33.                         $VarValue
34. ~                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
35.                         $VarValue
36. ~                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
37.                         $VarValue
38. .
```

Earlier you learned that console applications can update their environment with the latest environment variable values. Example 6.5 illustrates this point perfectly. The code defines an `UpdateEnv()` subroutine, which can be called at any time to update the Perl process' environment variable values.

The `UpdateEnv()` function works by procuring the latest list of system environment variables and setting Perl's `%ENV` hash to reflect the updated variables. Notice that the first system variables are set, followed by user variables. This order is important because user variables take precedence over system variables. Also, note that the variables are not expanded before being set.

Example 6.5 *Updating Perl's Copy of Environment Variables*

```
01. use Win32::AdminMisc;
02.
03. UpdateEnv();
04.
05. sub UpdateEnv
06. {
07.   my %VarList;
08.   if( Win32::AdminMisc::GetEnvVar( \%VarList, ENV_SYSTEM ) )
09.   {
10.     print "Updating system environment variables:\n";
11.     foreach my $Var ( keys( %VarList ) )
12.     {
13.       print " $Var\n";
```

continues ▶

Example 6.5 *continued*

```
14.       $ENV{$Var} = $VarList{$Var};
15.     }
16.   }
17.   if( Win32::AdminMisc::GetEnvVar( \%VarList, ENV_USER ) )
18.   {
19.     print "Updating user environment variables:\n";
20.     foreach my $Var ( keys( %VarList ) )
21.     {
22.       print " $Var\n";
23.       $ENV{$Var} = $VarList{$Var};
24.     }
25.   }
26. }
```

Modifying Variables

Generally speaking, a logon script does not need to modify environment variables. Typically, a script queries such a variable so that it can make logical decisions. For example, a logon script may query the `Temp` variable to determine the path of the temporary file directory, possibly to clean it out. Such querying is common, but setting environment variables from a logon script is not common. There are, however, exceptions to this rule, such as when a logon script wants to record the time and date when the user logged in.

A nonadministrative user's logon script has permissions to modify only the user set of environment variables, not the system set (unless administrative privileges have been granted). Consider a machine with a system environment variable called `INCLUDE` that consists of a list of semicolon-separated paths pointing to where C++ header files are located. This is a system variable, so it cannot accidentally be modified by a user. When a developer needs to point to her own set of include directories, she could create a user environment variable named `INCLUDE` that would override the system version, thus losing whatever values the system administrator had set. If the `INCLUDE` system variable contained a reference to `%INCLUDE%`, then the user variable would expand the reference into the system variable. This would allow the user to modify the `INCLUDE` variable's value without modifying the value directly.

You can implement another strategy to ensure that the system variable is always first in a path. For example, if the user adds his own version of the `INCLUDE` variable

```
C:\MyInclude;%INCLUDE%
```

the system INCLUDE variable is positioned after the user's version. In cases where this use is unacceptable, you can add a reference to the user variable by a different name. So, if the system variable INCLUDE consists of

```
C:\dev\include;\\server\share\include;%_INCLUDE%
```

the user _INCLUDE variable expands at the end of the system variable. The only problem here is that a user can create his own INCLUDE variable, overriding the system one. Example 6.6 comes in at this point.

This script shows how environment variables can be modified by a logon (or any other) script. This particular script searches for any user variable that has the same name as a system variable. If the system variable references the variable name prepended with an underscore, the user's version of the variable is deleted, and a new one is created prepended by an underscore.

As you learned earlier, Perl's %ENV hash reflects only the Perl script's personal copy of the environment. Therefore, changes made to this hash may affect how the Perl script functions but may not be seen by any other application. For this reason, modifications to the environment that must persist across applications should be made using Win32::AdminMisc's SetEnvVar() or DelEnvVar() functions.

Lines 4 and 5 retrieve the list of user and system variables. Notice that the script does not rely on the %ENV hash because it does not discern between user and system environment variables. Lines 9 through 18 simply iterate through the list looking for any user variable that has the same name as a system variable, *and* the system variable references an "underscored" version of the user variable.

Lines 24 and 32 contain the real meat of the script. Line 24 sets the new "underscored" user variable name, and line 32 deletes the original user variable. Note that lines 28 and 34 set the new variable and remove the original variable, respectively, from the %ENV hash. This is important if the script continues to process environment variables using %ENV. If these lines did not exist, the %ENV hash would have no idea that these changes were made.

Example 6.6 *Modifying Environment Variables*

```
01. use Win32;
02. use Win32::AdminMisc;
03.
04. Win32::AdminMisc::GetEnvVar( \%System, ENV_SYSTEM );
05. Win32::AdminMisc::GetEnvVar( \%User, ENV_USER );
06.
07. @NewUserVar = ();
08.
09. foreach $VarName ( keys( %System ) )
```

continues ▶

Example 6.6 *continued*

```
10. {
11.   if( defined $User{$VarName} )
12.   {
13.     if( $System{$VarName} =~ /_$VarName/ )
14.     {
15.       push( @NewUserVar, $VarName );
16.     }
17.   }
18. }
19.
20. # Modify the user env var space
21. foreach $VarName ( @NewUserVar )
22. {
23.   print "Setting new user var: _$VarName\n";
24.   if( Win32::AdminMisc::SetEnvVar( "_$VarName",
25.                                    $User{$VarName},
26.                                    ENV_USER ) )
27.   {
28.     $ENV{"_$VarName"} = $User{$VarName};
29.
30.     # Remove conflicting user env vars
31.     print "Deleting user var: $VarName\n";
32.     if( Win32::AdminMisc::DelEnvVar( $VarName, ENV_USER ) )
33.     {
34.       delete $ENV{$VarName};
35.     }
36.   }
37. }
```

Effectively Processing Logon Scripts

Many administrators provide users with their own, unique logon scripts. This is very useful for fine-tuning each user's experience; however, it can become an absolute nightmare to manage as the number of users increases. Say that all user scripts map a drive letter to a file server. If the name of the file server changes, all user scripts must be modified. In a network of thousands of users, modifying so many scripts is both time-consuming and tedious. One way around this problem is to leverage user groups.

Most users are members of groups that pertain to their position in an organization. For example, all accountants may be in the Accounting group and all administrators in the Administrators group. Because most of these users require resources that correspond to the groups they are in, it makes sense to create one script for each group. The accountants' script may map a drive to the accounting file server or a printer to some particular print server. The administrators' script, on the other hand, may start a Web browser that loads an administrative page indicating the current status of all the network servers.

Such an infrastructure would be ideal if it were not for having to modify each user's account to point to the correct script. An administrator's logon script may be

```
scripts\administrators.pl
```

and an accountant's logon script may be

```
scripts\accountants.pl
```

However, this solution breaks down if the organization has accountant administrators—that is, members of both the Administrators and Accountants groups. Likewise, this solution does not scale well as the number of users increases. Constantly making modifications every time someone moves from one position to another in the organization becomes quite unmanageable.

A much more effective method to manage such scalability issues is to use the same logon script for all users—a sort of unified logon script.

Unified Logon Scripts

An efficient method to manage user logon scripts is to use a *unified logon script*. Such a script enables you to configure every user to use the same logon script. When you create a new user account, you simply configure it to execute the same logon script as all the other users. This efficient technique allows the logon script itself to implement the logic to figure out what it should do for the user during the logon sequence.

Unified logon scripts can be beneficial because they are shared by all users. Consider such a script that collects the current date, time, IP address, computer name, and username and then submits that information to a centralized database. This information provides you with quick access to efficiently find what machine a user is on. In a large environment, this information can be quite advantageous for pinpointing the location of a particular user at a particular time. This is especially true if a user may have logged on to multiple machines, making it difficult to find him based on his network-registered username (because network-based usernames are unique and can be registered on only one machine at a time).

Processing User Groups

As you learned earlier in the section "Effectively Processing Logon Scripts," breaking out logon scripts based on user group membership can greatly enhance effectiveness of the logon scripts. The problem is that if a network implements a unified logon script, you cannot easily decide which groups a given user belongs to. If the unified logon script could discover which groups the user belongs to, it could execute each related group script.

Example 6.7 contains a simple example of discovering group membership using the `Win32::Lanman` extension. Lines 9-12 call `NetUserGetLocalGroups()` with the `LG_INCLUDE_INDIRECT` flag. This call discovers any local group the user is a member of in addition to any local group that the user is indirectly a member of (if the local group contains a global group the user is a member of). Lines 13-15 discover what global groups the user is a member of. The script accepts a username as the first parameter but uses the current logged-on user if nothing is passed in.

You can easily adopt this script to a logon script, as shown later in Example 6.9.

Example 6.7 *Discovering Group Membership Using* `Win32::Lanman`

```
01. use Win32;
02. use Win32::NetAdmin;
03. use Win32::Lanman;
04.
05. $Name = Win32::LoginName() unless( $Name = shift @ARGV );
06. $Domain = Win32::DomainName();
07. Win32::NetAdmin::GetAnyDomainController( '', $Domain, $Dc );
08.
09. Win32::Lanman::NetUserGetLocalGroups( $Dc,
10.                                       $Name,
11.                                       LG_INCLUDE_INDIRECT,
12.                                        \@LocalGroups );
13. Win32::Lanman::NetUserGetGroups( $Dc,
14.                                  $Name,
15.                                   \@GlobalGroups );
16.
17. foreach my $Hash ( sort( { $a->{name} cmp $b->{name} }
18.                    ( @LocalGroups, @GlobalGroups ) ) )
19. {
20.     print "\t";
21.     print "$Hash->{name}\n";
22. }
```

Machine Setup Script

One of the more mundane aspects of system administration is configuring new machines. This process can be quite slow and tedious. Each machine requires that you install the operating system, configure it, install appropriate software, and configure that as well. An easy way to automate this process is to leverage logon scripts.

By creating a special administrative account that has a special logon script, you can simply walk over to a new machine, log on, and then walk away. The logon script runs and begins the work of reconfiguring the computer. When it is finished, the logon script can reboot the computer so that it is ready for use by a user. And all it took was someone simply logging on with the special account.

You could rewrite the logon script so that it creates the appropriate directories, copies files, sets permissions, creates local accounts and groups, registers particular services, and even sets up scheduled tasks to execute on a routine schedule.

Example 6.8 is an example of such a setup script. It was designed with the expectation that an administrative user account would call it as its logon script. Of course, an administrator would have to modify this script to reflect her own requirements, server names, paths, and so on.

Currently, the script adds the Domain Admins group into the local machine's Administrators group. This gives any domain administrator access over the machine.

Next, the drive that holds the Perl tree is converted to NTFS so that the Perl directory tree can be secured from modification by users. After the conversion is complete and the Perl tree has been secured, Perl is copied from a network file server, and the system's `PATH` environment variable is modified to include the `perl\bin` directory.

Finally, the current user is removed from the Registry so that the next user to log on will not see what account was used to run this script. A log entry is made to a log file on a remote file server, and the machine is rebooted.

Example 6.8 *An Automated Machine Setup Logon Script*

```
01. use Win32;
02. use Win32::Perms;
03. use Win32::NetAdmin;
04. use Win32::Registry;
05. use Win32::AdminMisc;
06.
07. $Machine = Win32::NodeName();
08. $Domain = Win32::DomainName();
09.
10. $PerlDrive = 'c:';
11. $PerlPath = '\perl';
12.
13. $REMOTE_PERL_PATH = '\\\\server\perlshare$\perl';
14. $NTFS_CONVERSION_APP = "convert.exe";
15. $NTFS_CONVERSION_PARAM = "/FS:NTFS /V";
16.
17. ConfigGroup( $Domain );
18. if( ConfigDrive( $PerlDrive ) )
19. {
20.   SecureDir( "$PerlDrive$PerlPath" );
21.   CopyDir( $REMOTE_PERL_PATH, "$PerlDrive$PerlPath" );
22.   ConfigPath( "$Dir\\bin" );
23. }
24.
25. ConfigLastUser( "" );
```

continues ▶

Example 6.8 *continued*

```
26. Log( $LogFile );
27. RebootMachine();
28. print "Finished.\n";
29.
30. sub Log
31. {
32.  my( $LogFile ) = @_;
33.
34.  if( open( LOG, ">> $LogFile" ) )
35.  {
36.   flock( LOG, 2 );
37.   seek( LOG, 0, 2 );
38.   print LOG Win32::NodeName(), "\t", scalar( localtime() ), "\n";
39.   flock( LOG, 8 );
40.   close( LOG );
41.  }
42. }
43.
44. # Reboot the machine
45. sub RebootMachine
46. {
47.   Win32::AdminMisc::ExitWindows( EWX_REBOOT | EWX_FORCE );
48. }
49.
50. # Remove the "Last Logged On User" so nobody knows
51. sub ConfigLastUser
52. {
53.   my( $User ) = @_;
54.   my $Key;
55.   my $Path = 'SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon';
56.   if( $HKEY_LOCAL_MACHINE->Create( $Path, $Key ) )
57.   {
58.     $Key->SetValueEx( $Key, 0, REG_SZ, $User );
59.     $Key->Close();
60.   }
61. }
62.
63. sub ConfigGroup
64. {
65.   my( $Domain ) = @_;
66.
67.   $Domain .= "\\" if( '' ne $Domain );
68.
69.   # Add the Domain Admins group to the Administrators group
70.   Win32::NetAdmin::LocalGroupAddUsers( "",
71.                                        "Administrators",
72.                                        "$Domain" . "Domain Admins" );
73.
74. }
75.
76. sub ConfigPath
77. {
78.   my( $Dir ) = @_;
79.   my $RegexDir = "$Dir";
```

```
80.   my $Path;
81.
82.   # First check to see if the $Dir is already in the path...
83.   $Path = Win32::AdminMisc::GetEnvVar( 'PATH' );
84.
85.   # Prepare $Dir for a regex...
86.   $RegexDir =~ s/([.\\\$])/\\$1/g;
87.   if( $Path !~ /$Dir/i )
88.   {
89.     # Add $Dir to the system (not user) PATH
90.     Win32::AdminMisc::SetEnvVar( 'PATH', "$Path;$Dir" );
91.   }
92.   return( 1 );
93. }
94.
95. sub CopyDir
96. {
97.   my( $RemoteDir, $DestDir ) = @_;
98.
99.   print "Copying files from '$RemoteDir' to '$DestDir' ...\n";
100.
101.   # XCOPY.EXE will autocreate the destination dir
102.   `xcopy "$RemoteDir\\*.*" "$DestDir\\*.*" /s`;
103. }
104.
105. sub SecureDir
106. {
107.   my( $Dir ) = @_;
108.   my( $Perm, $Result );
109.
110.   print "Securing the $Dir directory...\n";
111.
112.   `md "$Dir"`;
113.
114.   if( $Perm = new Win32::Perms( $Dir ) )
115.   {
116.     # Remove *all* entries...
117.     $Perm->Remove( -1 );
118.
119.     $Perm->Allow( 'Administrators', FULL_CONTROL_FILE, FILE );
120.     $Perm->Allow( 'Administrators', FULL_CONTROL_DIR, DIR );
121.
122.     $Perm->Allow( 'Everyone', READ_FILE, FILE );
123.     $Perm->Allow( 'Everyone', LIST_DIR, DIR );
124.
125.     $Perm->Owner( 'Administrators' );
126.
127.     $Result = $Perm->Set();
128.   }
129.   else
130.   {
131.     print " Unable to create security descriptor. ";
132.     print "Directory is not secured.\n";
133.   }
```

continues ▶

Example 6.8 *continued*

```
134.
135.   return( 0 != $Result );
136. }
137.
138. sub ConfigDrive
139. {
140.   my( $Drive ) = @_;
141.   my %Info;
142.
143.   print "Configuring the $Drive drive ...\n";
144.
145.   # Make sure that the drive exists
146.   if( ! -e $Drive )
147.   {
148.     print " The $Drive drive does not exist.\n";
149.     return( 0 );
150.   }
151.
152.   # Do we have a fixed hard drive?
153.   if( DRIVE_FIXED != Win32::AdminMisc::GetDriveType( $Drive ) )
154.   {
155.     print " The $Drive drive is not a local fixed hard drive. ";
156.     print "Skipping drive formatting.\n";
157.     return( 0 );
158.   }
159.
160.   # If the drive is not NTFS then convert it into NTFS
161.   %Info = Win32::AdminMisc::GetVolumeInfo( $Drive );
162.   if( 'NTFS' ne $Info{FileSystemName} )
163.   {
164.     my $Cmd = "$NTFS_CONVERSION_APP $Drive $NTFS_CONVERSION_PARAMS";
165.     print " The $Drive drive is not an NTFS drive.\n";
166.     print " Converting from the $Info{FileSystemName} format...\n";
167.     @{$Log{ntfs_conversion}} = `$Cmd`;
168.   }
169.
170.   return( 1 );
171. }
```

CASE STUDY

Implementing a Unified Logon Script

A new network administrator was hired into a medium-sized company. One of her first tasks was to revamp the logon process for her users. She was in charge of several thousand users, each of whom ran one of several different logon scripts. These scripts were quite elaborate and designed to perform specific tasks, making it difficult to adapt them to groups of users.

The administrator decided to utilize a unified logon script for all her users. Because each user was a member of one of several NT groups, she decided to make the main logon script call out and run separate

group-based logon scripts. This way, she could manage large groups of users by simply changing their NT group memberships.

Her network had many subnets, some of which were located across slow frame relay links. Therefore, she decided that it would be best to replicate all the logon scripts across all the DCs on each subnet. She set up the controllers so that they replicated from the PDC at a regular interval. As a result, updating scripts was as easy as changing them in only one location. In the `NetLogon` network share, she created a subdirectory called Scripts where she placed all the group-based scripts.

She wrote a script for most of her domain's groups, such as Accounting.pl, Marketing.pl, Developers.pl, and, of course, Administrators.pl.

She also created a subdirectory called Users. In this directory, she could place logon scripts for every user who required one. For certain network administrators and upper management users who had specialized needs that required their own personal logon script, the Users directory contained Perl scripts named after the specific user account it belonged to. The unified logon script made use of the infrastructure illustrated in Example 6.9.

Lines 9 through 14 create a hash that contains logon-related information. Notice that the domain controller key (`dc`) hash has been assigned the `undef` value. This is done because `Win32::NetAdmin`'s `GetAnyDomainController()` function (lines 24 and 25) requires that *something* is assigned to the variable before it is passed into the function.

Line 28 fetches a list of all the DCs. The script then walks through this list, one machine at a time, looking for the particular shared directory. For each machine that is sharing the logon scripts, a ping test is performed to find the closest server (lines 36 through 42). (This process may not be as effective on larger networks that have many DCs and subnets because it might take awhile to ping all machines. It also increases network traffic.). Line 46 then sets the user environment variable `ScriptServer` with the server name so that scripts that run later during the logon session can discover what machine was used to run the logon scripts.

At this point, an attempt is made to connect to an ODBC database (line 50). This is done before any additional scripts are loaded so that they can take advantage of the database connection. In this particular script, the connection to the ODBC database is made using a Data

continues ▶

Source Name (DSN). It may be easier to rely instead on a DSN connection string hard-coded into the script. This would prevent any problems that would arise if the computer's DSN is removed or altered.

Next, the script discovers which groups the user belongs to by using the `Win32::Lanman` extension (lines 53 through 61) discussed earlier in the section "Processing User Groups." Lines 65 through 71 actually perform the work of loading each group logon script. Notice that it uses the `require` function. Therefore, each logon script *must* return a `TRUE` value (any nonzero return value is fine). Lines 75 through 78 simply load the user's personal logon script, if one exists.

Lines 81 through 114 modify the user's environment variables so they do not collide with system variables of the same name. This was discussed in detail in the earlier section "Modifying Variables."

The code from lines 118 through 130 removes all the files in the temporary directory that are older than two weeks. Notice that it actually does this for both the user's temporary directory as well as the system-defined temporary directory. Finally, lines 132 through 143 submit an update to the logging database with the time and date, username, user's domain, and machine name. This information allows administrators to quickly discover what machine a given user has logged on to.

After all accounts were changed to use the unified logon script, the administrator found that she was spending less time having to make changes to dozens of scripts just because a file server was taken offline or a user switched groups, and she was spending more time tending to other important network administration tasks.

Example 6.9 *Case Study of a Unified Logon Script*

```
01. use Win32;
02. use Win32::Lanman;
03. use Win32::NetAdmin;
04. use Win32::AdminMisc;
05. use Win32::ODBC;
06.
07. my $TempFileGracePeriod = 2 * $WEEK;
08. my $DSN = "LogonDatabase";
09. my %Logon = (
10.     account => Win32::LoginName(),
11.     machine => Win32::NodeName(),
12.     domain  => Win32::DomainName(),
13.     dc      => undef,
14. );
```

```
my %LogonPath = (
    share    => 'NetLogon',
    rootdir  => 'Scripts',
    userdir  => 'Users',
);
my $SERVER_TYPE = SV_TYPE_DOMAIN_CTRL | SV_TYPE_DOMAIN_BAKCTRL;
my $ScriptsUnc = "";
my @GroupList = ();

Win32::NetAdmin::GetAnyDomainController( '', $Logon{domain},
                                                    $Logon{dc} );

# Discover the local Perl server...
if( Win32::NetAdmin::GetServers( '', '', $SERVER_TYPE, \@List ) )
{
  my $Time = 0xFFFF;
  foreach my $Machine ( @List )
  {
    my $Unc = "\\\\$Machine\\$LogonPath{share}\\$LogonPath{rootdir}";
    if( -d $Unc )
    {
      my $Output = join( " ", `ping -n 1 $Machine` );
      my( $NewTime ) =~ /Reply from.*?\s+time[=<>](\d+)ms/is );
      if( ( "" ne $NewTime ) && ( $NewTime < $Time ) )
      {
        $Time = $NewTime;
        $ScriptsUnc = $Unc;
      }
    }
  }
}
Win32::AdminMisc::SetEnvVar( 'ScriptServer', $ScriptsUnc, ENV_USER, 10 );

# Before we call any other scripts first open the database
# so that each script can use it.
$db = new Win32::ODBC( $DSN );

# Discover what local groups the user is a member of...
Win32::Lanman::NetUserGetLocalGroups( $Logon{dc},
                                      $Logon{account},
                                      LG_INCLUDE_INDIRECT,
                                      \@LGroups );

# Discover what global groups the user is a member of...
Win32::Lanman::NetUserGetGroups( $Logon{dc},
                                 $Logon{account},
                                 \@GGroups );

print "Starting the unified logon process...\n";

foreach my $Group ( @LGroups, @GGroups )
{
```

continues ▶

Example 6.9 *continued*

```
67.   my $GroupName = $Group->{name};
68.   my $Script = "$ScriptsUnc\\$GroupName.pl";
69.   print " Calling $Script\n";
70.   require $Script if( -f $Script );
71. }
72.
73.
74. # If the user has his own logon script run it...
75. if( -f "$ScriptsUnc\\$LogonPath{userdir}\\$Logon{account}.pl" )
76. {
77.   require "$ScriptsUnc\\$LogonPath{userdir}\\$Logon{account}.pl";
78. }
79.
80. # Fix any user overriding of system variables
81. if( ( Win32::AdminMisc::GetEnvVar( \%System, ENV_SYSTEM ) )
82.   && ( Win32::AdminMisc::GetEnvVar( \%User, ENV_USER ) ) )
83. {
84.   my @NewUserVar = ();
85.   foreach $VarName ( keys( %System ) )
86.   {
87.     if( defined $User{$VarName} )
88.     {
89.       if( $System{$VarName} =~ /_$VarName/ )
90.       {
91.         push( @NewUserVar, $VarName );
92.       }
93.     }
94.   }
95.
96.   # Modify the user env var space
97.   foreach $VarName ( @NewUserVar )
98.   {
99.     print "Setting new user var: _$VarName\n";
100.     if( Win32::AdminMisc::SetEnvVar( "_$VarName",
101.                                      $User{$VarName},
102.                                      ENV_USER ) )
103.     {
104.       $ENV{"_$VarName"} = $User{$VarName};
105.
106.       # Remove conflicting user env vars
107.       print "Deleting user var: $VarName\n";
108.       if( Win32::AdminMisc::DelEnvVar( $VarName, ENV_USER ) )
109.       {
110.         delete $ENV{$VarName};
111.       }
112.     }
113.   }
114. }
115.
116. # Clean up the temp directory...
117. # First start with the system temp directory...
118. push( @TempDirs, Win32::AdminMisc::GetEnvVar( 'Temp', ENV_SYSTEM ) );
```

```
119. push( @TempDirs, Win32::AdminMisc::GetEnvVar( 'Temp', ENV_USER ) );
120. foreach my $TempDir ( @TempDirs )
121. {
122.   foreach my $Path ( glob( "$TempDir\\*.*" ) )
123.   {
124.     next if( ! -f $Path );
125.     if( $TempFileGracePeriod < time() - (stat( $Path ))[10] )
126.     {
127.       unlink( $Path);
128.     }
129.   }
130. }
131.
132. # Update the logon database...
133. if( $db )
134. {
135.     my $Query = "INSERT INTO UserLogon
136.                  (Domain, Account, Machine, Date)
137.                  VALUES ('$Logon{domain}',
138.                          '$Logon{account}',
139.                          '$Logon{machine}',
140.                          {fn Now()})";
141.     $db->Sql( $Query );
142.     $db->Close() if( $db );
143. }
144.
145. BEGIN
146. {
147.     $SECOND = 1;
148.     $MINUTE = 60 * $SECOND;
149.     $HOUR = 60 * $MINUTE;
150.     $DAY = 24 * $HOUR;
151.     $WEEK = 7 * $DAY;
152. }
```

Conclusion

Seeing a radical increase in productivity is not uncommon when you simply revamp the way logon scripts are implemented. Implementing them differently can free up a system administrator's time and provide the user with a personalized logon anywhere on the network. Such scripts are powerful indeed whether they are used to log on a user or act as a launching point to automate the configuration of a machine. And Perl is, of course, a powerful ally in working with logon scripts.