Hey, Scripting Guy!

Back Up Your Event Logs with a Windows PowerShell Script

The Microsoft Scripting Guys

Resources

- TechNet Flash Newsletter
- TechNet Technology News feed
- MSDN Magazine
- MSDN Flash Newsletter

Hot! Sweltering, sticky, breath-taking humid heat was the first thing Jason and I noticed when we landed at the airport in Kuala Lumpur, Malaysia. Before we had reached the curb, a friendly cab driver had our bags in the trunk and the door open for us. The trip was brisk through palm tree–lined roads that led to Kuala Lumpur City Center (KLCC). As soon as we entered the highway, we could see the glistening top of the Patronus Towers that mark KLCC from miles away in every direction. We were in town to teach a Microsoft Operations Framework (MOF) class to a group of Microsoft employees.

The highlight of the MOF class was an airport simulation that absolutely no one ever got right on the first attempt. But that was part of the point of the class—process improvement. We had taught the class more than twenty times, and no one ever succeeded on the first day. Some classes barely made it through the simulation on the last day of the class. Until now.

At the end of the first day, it was time for round one of the simulation. Jason and I exchanged knowing glances as we gave out the instructions. The students sat with rapt attention, and then as the clock started ticking, instead of the usual loud running around in scrambled confusion, the students quietly got together into a small huddle. They talked rapidly for about five minutes as one student opened up his laptop and began making notes. Then they calmly turned around and proceeded to win the game in the first round.

How did they do it? They simply directed all of their attention to the essential elements of the scenario. They ignored all the nonessential information and created a new process that solved the problem. Because they focused on the core problem, they were free of any complicated work rules, and they were able to focus their energy on the task at hand.

Overly complex work rules can hinder productivity. Today's script grew out of a situation one customer faced in which they were spending several hours a day backing up event logs from various network servers and copying them to a central storage location where they were later backed up to tape. After we got past their complex work rules, we were able to create a custom script that did exactly what they needed. This script saved that customer 10 hours a week and 500 hours a year in labor that they had previously spent managing event logs.

When we take time out from complicated work rules, we can focus more attention and energy on the task at hand, which is to provide IT services. Let's take a look at a script that can be used to back up, archive, and clear event logs across the network. The entire BackUpAndClearEventLogs.ps1 script is shown in **Figure 1**.

Figure 1 BackUpAndClearEventLogs.ps1

```
Param(
       $LogsArchive = "c:\logarchive",
       $List,
       $computers,
       [switch]$AD,
       [switch]$Localhost,
```

```
       [switch]$clear,
       [switch]$Help
     )
Function Get-ADComputers
{
 $ds = New-Object DirectoryServices.DirectorySearcher
 $ds.Filter = "ObjectCategory=Computer"
 $ds.FindAll() |
     ForEach-Object { $_.Properties['dnshostname']}
} #end Get-AdComputers

Function Test-ComputerConnection
{
 ForEach($Computer in $Computers)
 {
  $Result = Get-WmiObject -Class win32_pingstatus -Filter "address='$computer'"
  If($Result.Statuscode -eq 0)
   {
     if($computer.length -ge 1)
        {
         Write-Host "+ Processing $Computer"
         Get-BackUpFolder
        }
   } #end if
   else { "Skipping $computer .. not accessible" }
 } #end Foreach
} #end Test-ComputerConnection



Function Get-BackUpFolder
{
 $Folder = "{1}-Logs-{0:MMddyymm}" -f [DateTime]::now,$computer
  New-Item "$LogsArchive\$folder" -type Directory -force  | out-Null
  If(!(Test-Path "\\$computer\c$\LogFolder\$folder"))
    {
      New-Item "\\$computer\c$\LogFolder\$folder" -type Directory -force | out-
Null
    } #end if
 Backup-EventLogs($Folder)
} #end Get-BackUpFolder

Function Backup-EventLogs
{
 $Eventlogs = Get-WmiObject -Class Win32_NTEventLogFile -ComputerName $computer
 Foreach($log in $EventLogs)
        {
            $path = "\\{0}\c$\LogFolder\$folder\{1}.evt" -f
$Computer,$log.LogFileName
            $ErrBackup = ($log.BackupEventLog($path)).ReturnValue
            if($clear)
               {
                if($ErrBackup -eq 0)
                  {
                   $errClear = ($log.ClearEventLog()).ReturnValue
                  } #end if
                else
                  {
                    "Unable to clear event log because backup failed"
                    "Backup Error was " + $ErrBackup
                  } #end else
               } #end if clear
            Copy-EventLogsToArchive -path $path -Folder $Folder
        } #end foreach log
} #end Backup-EventLogs

Function Copy-EventLogsToArchive($path, $folder)
{
 Copy-Item -path $path -dest "$LogsArchive\$folder" -force
} # end Copy-EventLogsToArchive
```

```
Function Get-HelpText
{
 $helpText= `
@"
 DESCRIPTION:
 NAME: BackUpAndClearEventLogs.ps1
 This script will backup, archive, and clear the event logs on
 both local and remote computers. It will accept a computer name,
 query AD, or read a text file for the list of computers.

 PARAMETERS:
 -LogsArchive local or remote collection of all computers event logs
 -List path to a list of computer names to process
 -Computers one or more computer names typed in
 -AD switch that causes script to query AD for all computer accounts
 -Localhost switch that runs script against local computer only
 -Clear switch that causes script to empty the event log if the back succeeds
 -Help displays this help topic

 SYNTAX:
 BackUpAndClearEventLogs.ps1 -LocalHost

 Backs up all event logs on local computer. Archives them to C:\logarchive.

 BackUpAndClearEventLogs.ps1 -AD -Clear

 Searches AD for all computers. Connects to these computers, and backs up all
event
 logs. Archives all event logs to C:\logarchive. It then clears all event logs
 if the backup operation was successful.

 BackUpAndClearEventLogs.ps1 -List C:\fso\ListOfComputers.txt

 Reads the ListOfComputers.txt file to obtain a list of computer. Connects to
these
 computers, and backs up all event logs. Archives all event logs to C:\logarchive.

 BackUpAndClearEventLogs.ps1 -Computers "Berlin,Vista" -LogsArchive
"\\berlin\C$\fso\Logs"

 Connects to a remote computers named Berlin and Vista, and backs up    all event
 logs. Archives all event logs from all computers to the path c:\fso\Logs
directory on
   a remote computer named Berlin.

BackUpAndClearEventLogs.ps1 -help

Prints the help topic for the script
"@ #end helpText
  $helpText
}

# *** Entry Point To Script ***

If($AD) { $Computers = Get-ADComputers; Test-ComputerConnection; exit }
If($List) { $Computers = Get-Content -path $list; Test-ComputerConnection; exit }
If($LocalHost) { $computers = $env:computerName; Test-ComputerConnection; exit }
If($Computers)
  {
   if($Computers.Contains(",")) {$Computers = $Computers.Split(",")}
   Test-ComputerConnection; exit
  }
If($help) { Get-HelpText; exit }
"Missing parameters" ; Get-HelpText
```

The first thing we do in the BackUpAndClearEventLogs.ps1 script is use the Param statement to create some command-line parameters for the script, like so:

```
Param(
       $LogsArchive = "c:\logarchive",
       $List,
       $Computers,
       [switch]$AD,
       [switch]$Localhost,
       [switch]$Clear,
       [switch]$Help
     )
```

We use several parameters in order to give the script lots of flexibility. The -LogsArchive parameter is used to define the location of the event log archive. We set this to a default location on the C:\ drive, but by using -LogsArchive, you can choose any location that makes sense for your computing environment.

The -List parameter lets you supply a list of computers to the script via a text file. This parameter expects the complete path to a text file containing the computer names. The syntax for the text file is simple; you just place the name of each computer you want to work with on its own individual line.

The -Computers parameter allows you to supply a list of computers from the command line when you run the script. To use this parameter, you place within a set of quotation marks computer names separated by commas. Ideally, you'd use this parameter if you wanted to check just a small number of computers.

Next come four switched parameters. One of the coolest is a switched parameter called –AD, which lets you query Active Directory for a list of computers. It makes sense to use this switch if you are going to check a large number of event logs on all the computers on your network. Of course, on a large network this could take quite a long time. If you want to run the script against your local computer, use the -Localhost switch, which instructs the script to execute against the local machine. Besides backing up the event logs and archiving them to a central location, you can also empty the contents of the event logs by using the -Clear switched parameter. To get Help information, run the script using -Help. We now come to the first function in our script. The Get-ADComputers function is a query used to retrieve a listing of all of the computer accounts in Active Directory. This function does not require any input parameters. When the function is called, it uses the New-Object cmdlet to create an instance of a DirectoryServices.DirectorySearcher class from the Microsoft .NET Framework. We do not pass any information to the New-Object cmdlet, so the DirectoryServices.DirectorySearcher class is created using the default constructor. The new DirectorySearcher class is stored in the $ds variable, as shown here:

```
Function Get-ADComputers
{
$ds = New-Object DirectoryServices.DirectorySearcher
```

After we have an instance of the DirectorySearcher class, we can use the Filter property to create a search filter to reduce the number of items that are retrieved. LDAP search filters are documented in "Search Filter Syntax." The attribute we want is called ObjectCategory, and we're looking for a value of "Computer." After we have created our filter, we use the FindAll method from the DirectorySearcher object:

```
 $ds.Filter = "ObjectCategory=Computer"
$ds.FindAll() |
```

The results from the FindAll method are pipelined to the ForEach-Object cmdlet, which is used to iterate through the collection of DirectoryEntry objects that are returned by FindAll. Inside the script block, delineated by the curly brackets, we use the $_ automatic variable to refer to the current item on the pipeline. We access the properties of the DirectoryEntry object and return the dnshostname:

```
     ForEach-Object {$_.Properties['dnshostname']}
} #end Get-ADComputers
```

Now we'll create the Test-ComputerConnection function to ensure that the computer is on the network and running. This will prevent timeout issues and make the script more efficient. We begin by using the Function keyword, then specify the name of the function and open the script block:

```
Function Test-ComputerConnection
{
```

Next we need to walk through the collection of computers stored in the $Computers variable, which we'll do using the ForEach statement with the $Computer variable as the enumerator. We then open the script block using a left curly bracket:

```
 ForEach($Computer in $Computers)
{
```

We need to use the WMI class Win32_PingStatus to ping the remote computer. To do this, we use the Get-WmiObject cmdlet and specify the class Win32_PingStatus and create a filter that examines the address property to see if it matches the value stored in the $Computer variable. We store the results of this WMI query in a variable named $Result, as shown here:

```
  $Result = Get-WmiObject -Class Win32_PingStatus -Filter "address='$computer'"
```

Now we evaluate the status code returned from the WMI query. If the status code is equal to zero, there were no errors and the computer is up and running:

```
  If($Result.Statuscode -eq 0)
   {
```

For some strange reason, on my computer the query returns a phantom computer that it evaluates as present but that did not have a name. To get rid of the phantom computer, I added a line of code to ensure that the computer name was at least one character long:

```
     if($computer.length -ge 1)
        {
```

Next we provide a bit of feedback to the user by displaying a status message stating that we are processing the computer. We use the Write-Host cmdlet to provide this feedback:

```
         Write-Host "+t Processing $Computer"
```

Now we call the Get-BackupFolder function to find the folder to be used for the backup:

```
         Get-BackUpFolder
        }
   } #end if
```

If the computer is not accessible, there is no point in attempting to back up the event log because we won't be able to reach it. We display a status message indicating that we will skip the computer and exit the function:

```
    else { "Skipping $computer .. not accessible" }
  } #end Foreach
} #end Test-ComputerConnection
```

After evaluating the accessibility of the computer, it is time to create the Get-BackupFolder function:

```
Function Get-BackupFolder
{
```

The next line of code is somewhat odd-looking and therefore a bit confusing:

```
$Folder = "{1}-Logs-{0:MMddyymm}" -f [DateTime]::now,$computer
```

We are using the format operator (-f) to perform some value substitution within the string that we will use for the folder name. The string contains the number 1 in a pair of curly brackets, the word Logs surrounded by dashes, and another pair of curly brackets enclosing the number 0 followed by a bunch of letters.

Let's take this one step at a time. The –f operator performs a substitution of values contained in the string. Just as with an array, the first element begins at 0, the second at 1. The items on the right side of the –f operator are the substitute values that are placed in the appropriate slots on the left side.

Before getting back to the main script, let's take a moment to consider an example to clarify how substitution is done in the script. Notice how we substitute the word one for the {0} portion and the word two for the {1} portion in the following code:

```
PS C:\Users\edwilson> $test = "{0}-first-{1}-second" -f "one","two"
PS C:\Users\edwilson> $test
one-first-two-second
```

We probably should have rearranged our code so that the first element was in the first position. Instead, we have written it such that the second element is in the first position, and the first element is in the second position. If we had moved things around a bit, the line of code would have looked something like this:

```
PS C:\Users\edwilson> $computer = "localhost"
PS C:\Users\edwilson> $Folder = "{0}-Logs-{1:MMddyymm}" -f $computer,
[DateTime]::now
PS C:\Users\edwilson> $Folder
localhost-Logs-04070938
```

The {1:MMddyymm} in the above command is used to supply the current date and time. We don't want the normal display of the DateTime object that is shown here because it is too long and has characters that are not allowed for a folder name. The default display for the DateTime object is Tuesday, April 07, 2009 6:45:37 PM.

The letter patterns that follow the colon character in our script are used to control the way the DateTime values will be displayed. In our case, the month is followed by the day, the year, and the minute. These DateTime format strings are documented at Custom DateTime Format Strings. They were also discussed in a recent article on the

Microsoft Script Center, "How Can I Check the Size of My Event Log and Then Backup and Archive It If It Is More Than Half Full?"

Next we create the event log archive folder. To create the folder, we use the New-Item cmdlet and specify the type as Directory. We use the variable $LogsArchive and the pattern that we stored in the $folder variable to create the path to the archive. We use the -force parameter to enable creation of the entire path, if it is required. Because we are not interested in the feedback from this command, we pipeline the results to the Out-Null cmdlet, shown here:

```
New-Item "$LogsArchive\$folder" -type Directory -force | Out-Null
```

We also need to determine if the log folder exists on the computer. To do this we use the Test-Path cmdlet, like so:

```
  If(!(Test-Path "\\$computer\c$\LogFolder\$folder"))
    {
```

The Test-Path cmdlet returns a $true or $false Boolean value. It is asking, "Is the log folder there?" Placing the not operator (!) in front of the Test-Path cmdlet indicates we are interested only if the folder does not exist.

If the log folder does not exist on the remote computer, we use the New-Item cmdlet to create it. We have a hard-coded value of LogFolder, but you can change this. As in the previous New-Item cmdlet, we use the -force parameter to create the entire path and pipeline the results to the Out-Null cmdlet:

```
      New-Item "\\$computer\c$\LogFolder\$folder" -type Directory -force | out-Null
    } #end if
```

We now want to call the Backup-EventLogs function, which performs the actual backup of the event logs. We pass the path stored in the $folder variable when we call the function:

```
 Backup-EventLogs($folder)
} #end Get-BackUpFolder
```

Next, we create the Backup-EventLogs function:

```
Function Backup-EventLogs
{
```

We use the Win32_NTEventLogFile WMI class to perform the actual backup. To do this, we call the Get-WmiObject cmdlet and give it the class name of Win32_NTEventLogFile as well as the computer name contained in the $computer variable. We store the resulting WMI object in the $Eventlogs variable:

```
$Eventlogs = Get-WmiObject -Class Win32_NTEventLogFile -ComputerName $computer
```

By performing a generic, unfiltered WMI query, we return event log objects representing each event log on the computer. These are the classic event logs shown in **Figure 2**.

In order to work with these event logs, we need to use the ForEach statement to walk through the collection of WMI objects. We use the variable $log as our enumerator to

help keep our place as we walk through the collection:

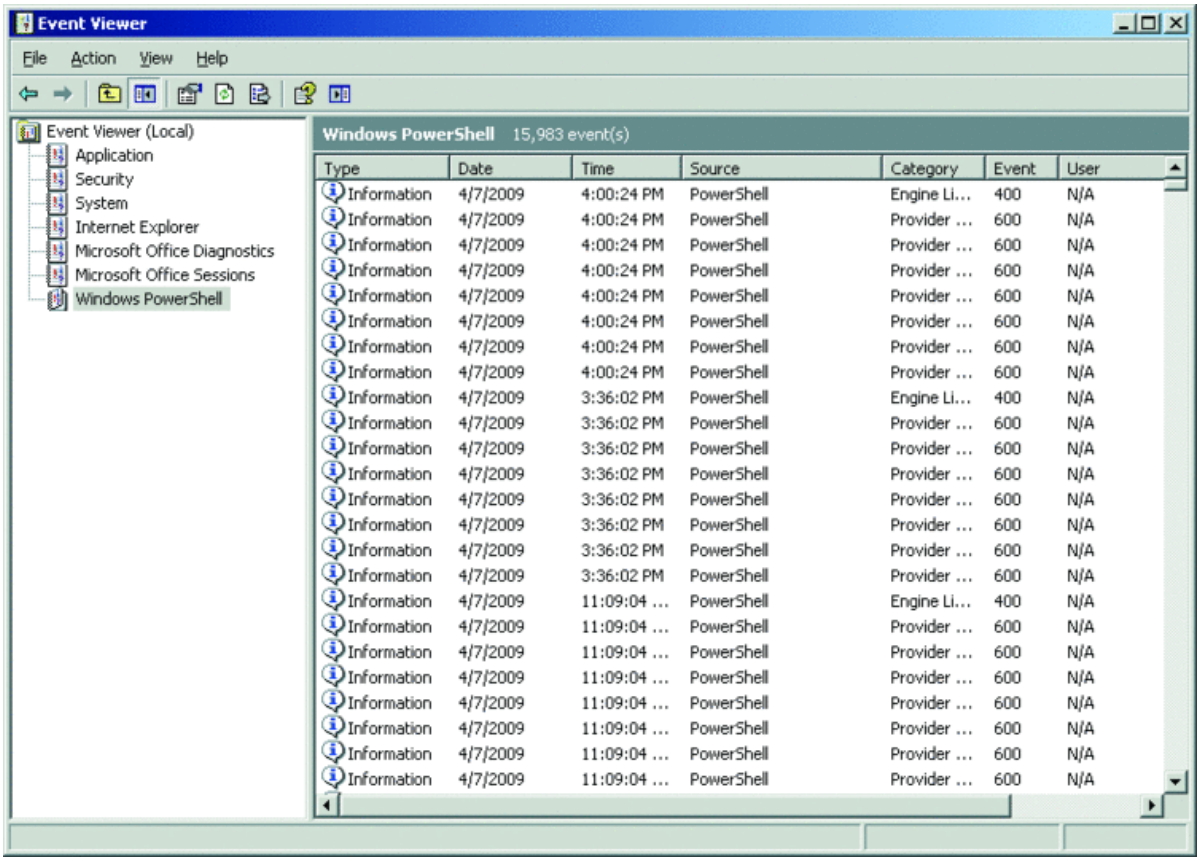

Figure 2 **Classic event logs retrieved by Win32_NTEventLogFile**

```
ForEach($log in $EventLogs)
       {
```

We now need to create the path. Once again, we use the format operator to do some pattern substitution. The {0} is a placeholder for the computer name in the path that will be used for the event logs. The {1} is a placeholder that is used to hold the log file name that will be used when backing up the event log:

```
            $path = "\\{0}\c$\LogFolder\$folder\{1}.evt" -f
$Computer,$log.LogFileName
```

Now we call the BackupEventLog method from the Win32_NTEventLogFile WMI class. We pass the path to the BackupEventLog method and retrieve the return value from the method call. We store the return value in the $ErrBackup variable as seen here:

```
            $ErrBackup = ($log.BackupEventLog($path)).ReturnValue
```

If the script were run with the –clear switch, the $clear variable would be present and, in that case, we would need to clear the event logs. Therefore, we use the if statement to determine whether the $clear variable is present:

```
            if($clear)
               {
```

Before we empty the event logs, we would like to ensure that the event log was successfully backed up. To do this, we inspect the value stored in the $ErrBackup variable. If it is equal to zero, we know there were no errors during the backup operation and that it is safe to proceed with emptying the event logs:

```
                if($ErrBackup -eq 0)
                  {
```

We call the ClearEventLog method from the Win32_NTEventLogFile WMI class and retrieve the ReturnValue from the method call. We store the ReturnValue in the $errClear variable as shown here:

```
                   $errClear = ($log.ClearEventLog()).ReturnValue
                  } #end if
```

If the value of the $ErrBackup variable is not equal to 0, we do not clear out the event logs. Instead, we display a status message stating we were unable to clear the event log because the backup operation failed. To provide additional troubleshooting information, we show the status code that was retrieved from the backup operation:

```
                else
                  {
                    "Unable to clear event log because backup failed"
                    "Backup Error was " + $ErrBackup
                  } #end else
                } #end if clear
```

Next, we copy the event logs to the archive location. To do this, we call the Copy-EventLogsToArchive function and give it the path to the event logs and the folder for the destination:

```
            Copy-EventLogsToArchive -path $path -Folder $Folder
        } #end foreach log
} #end Backup-EventLogs
```

Copy-EventLogsToArchive uses the Copy-Item cmdlet to copy the event logs to the archive location. We again use the -force parameter to create the folder if it does not exist:

```
Function Copy-EventLogsToArchive($path, $folder)
{
 Copy-Item -path $path -dest "$LogsArchive\$folder" -force
} # end Copy-EventLogsToArchive
```

Now we need to create some Help text for the script. To do this, we create a Get-HelpText function that stores the Help information in a single variable: $helpText. The Help text is written as a here-string, which lets us format the text as we want it to appear on the screen without concerning ourselves with escaping quotation marks. This makes it a lot easier for us to type a large string such as the one in **Figure 3**.

Figure 3 Get-HelpText Function

```
Function Get-HelpText
{
```

```
 $helpText= `
@"
 DESCRIPTION:
 NAME: BackUpAndClearEventLogs.ps1
 This script will backup, archive, and clear the event logs on
 both local and remote computers. It will accept a computer name,
 query AD, or read a text file for the list of computers.
 PARAMETERS:
 -LogsArchive local or remote collection of all computers event logs
 -List path to a list of computer names to process
 -Computers one or more computer names typed in
 -AD switch that causes script to query AD for all computer accounts
 -Localhost switch that runs script against local computer only
 -Clear switch that causes script to empty the event log if the back succeeds
 -Help displays this help topic
 SYNTAX:
 BackUpAndClearEventLogs.ps1 -LocalHost
Backs up all event logs on local computer. Archives them to C:\logarchive.
 BackUpAndClearEventLogs.ps1 -AD -Clear
 Searches AD for all computers. Connects to these computers, and backs up all
event
 logs. Archives all event logs to C:\logarchive. It then clears all event logs if
the
 backup operation was successful.
 BackUpAndClearEventLogs.ps1 -List C:\fso\ListOfComputers.txt
 Reads the ListOfComputers.txt file to obtain a list of computer. Connects to
these
 computers, and backs up all event logs. Archives all event logs to C:\logarchive.
 BackUpAndClearEventLogs.ps1 -Computers "Berlin,Vista" -LogsArchive
"\\berlin\C$\fso\Logs"
 Connects to a remote computers named Berlin and Vista, and backs up all event
 logs. Archives all event logs from all computers to the path c:\fso\Logs
directory on
 a remote computer named Berlin.
 BackUpAndClearEventLogs.ps1 -help
 Prints the help topic for the script
 "@ #end helpText
```

To display the Help text, we call the variable by name:

```
  $helpText
}
```

Next, we parse the command-line input shown here:

```
If($AD) { $Computers = Get-ADComputers; Test-ComputerConnection; exit }
If($List) { $Computers = Get-Content -path $list; Test-ComputerConnection; exit }
If($LocalHost) { $computers = $env:computerName; Test-ComputerConnection; exit }
```

If the $AD variable is present, the script was run with the -AD switch and we therefore populate the $Computers variable with the information obtained from the Get-ADComputers function. We then call the Test-ComputerConnection function, which will determine if the computer is online and back up the event logs. Then we exit the script. If the $List variable is present, we use the Get-Content cmdlet to read a text file and populate the $Computers variable. We then call the Test-ComputerConnection function and exit the script. If the $LocalHost variable is present, we populate the $Computers variable by reading the value directly from the computerName environment variable. We then call the Test-ComputerConnection function and exit the script.

If the $Computers variable is present, it means that the script was run and that the computer names were supplied from the command line. We will need to break these computer names into an array. To do this, we use the split method of the string object:

```
If($Computers)
  {
   if($Computers.Contains(",")) {$Computers = $Computers.Split(",")}
   Test-ComputerConnection; exit
  }
```

If the script was run with the -help switch, we call the Get-HelpText function, display the Help, and exit the script:

```
If($help) { Get-HelpText; exit }
```

If no parameters are present, we display a message that states we're missing parameters and then call the Get-Help test function:

```
"Missing parameters" ; Get-HelpText
```

The BackupAndClearEventLogs.ps1 script can be easily adapted to serve your needs on your network. For example, you could modify the Active Directory query so that it returns only servers from a particular organizational unit, or you could add an additional WMI query to further filter out the machines that are processed. We hope you enjoy the script and can use it where you work. Visit us and the community of scripters on the Script Center at where you can also catch our daily Hey, Scripting Guy! articles.

Ed Wilson, a well-known scripting expert, is the author of eight books, including *Windows PowerShell Scripting Guide Microsoft Press* (2008) and *Microsoft Windows PowerShell Step by Step* Microsoft Press (2007). Ed holds more than 20 industry certifications, including Microsoft Certified Systems Engineer (MCSE) and Certified Information Systems Security Professional (CISSP). In his spare time, he enjoys woodworking, underwater photography, and scuba diving. And tea.

Craig Liebendorfer is a wordsmith and longtime Microsoft Web editor. Craig still can't believe there's a job that pays him to work with words every day. One of his favorite things is irreverent humor, so he should fit right in here. He considers his greatest accomplishment in life to be his magnificent daughter.