

#PSBlogWeek

Logging Techniques





Table of Contents

Intro	i
Jason Wasser	
Building Readable Text Log Files	1
Thom Schumacher	
Slicing and Dicing Log Files	11
Jaap Brasser	
PowerShell logging in the Windows Event log	18
Adam Platt	
Reading the Event Log with Windows PowerShell.....	23
Adam Bertram	
Building Logs for CMtrace with PowerShell	31
#PSBlogWeek Wrap-up	37



#PSBlogWeek

Intro

#PSBlogWeek is a regular, week-long event where people in the PowerShell community coordinate blogging on a particular topic around Windows PowerShell. The purpose is to pool our collective PowerShell knowledge together over a 5-day period and write about a topic that anyone using PowerShell may benefit from. #PSBlogWeek is a Twitter hashtag, so feel free to stay up-to-date on the topic on Twitter at the #PSBlogWeek hashtag. For more information on #PSBlogWeek or if you'd like to volunteer for future sessions, contact [@adbertram](https://twitter.com/adbertram) on Twitter.





Building Readable Text Log Files

Jason Wasser
Follow on Twitter @wasserja



The skilled PowerShellers who authored the previous PowerShell [blog week](#) introduced and developed the practice of advanced functions.

As you mature in your PowerShell skills from one-liners to scripts, advanced functions, and eventually modules, you may realize the need to log the activities you are performing with your tools. Logging your PowerShell scripts can be extremely important for tools that change settings or data so that you can audit who made changes and when.

Surprisingly, logging functionality isn't a built-in PowerShell feature (yet) so we're left with building our own logging tools. Hit up your favorite search engine, the [TechNet Gallery](#), or the [PowerShell Gallery](#), and you'll find quite a few people who have built their own logging functions that may meet your needs.

I'm going to walk you through an advanced function that you can use to add logging to your scripts, but first, I want to show you how I got there.

When I first realized the need for logging in my scripts I started by using the [Add-Content](#) cmdlet, which I would sprinkle here and there in my scripts to capture key points in my operations.

Example:

```
1 # Get Operating System Name
2 Add-Content -Value 'Querying system for operating system information' -Path c:\Logs\script.log
3 $OS = Get-WmiObject -ClassName win32_operatingsystem
4 Add-Content -Value "$($OS.Caption)" -Path c:\Logs\script.log
5 $OS.Caption
```

When I realized that I was repeating too much code, I decided to create a helper function in my scripts to make it simpler for me.



```
1 Function Write-Log {
2     Param(
3         $Message,
4         $Logfile = 'c:\logs\script.log'
5     )
6
7     Add-Content -Value "$Message" -Path $Logfile
8 }
9
10 # Get Operating System Name
11 Write-Log -Message 'Querying system for operating system information'
12 $OS = Get-WmiObject -ClassName win32_operatingsystem
13 Write-Log -Message "$($OS.Caption)"
14 $OS.Caption
```

(Very) Basic Write-Log Helper Function

Sample Log File:

```
Querying system for operating system information
Microsoft Windows 8.1 Enterprise
```

It saved me a little work and it looks a little cleaner. However, the log file is only lines of text without any specific information about when each action happened or how much time elapsed between actions. Let's add a time stamp to our helper function.

```
1 Function Write-Log {
2     Param(
3         $Message,
4         $Logfile = 'c:\logs\script.log'
5     )
6
7     Add-Content -Value "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') $Message" -Path $Logfile
8 }
9
10 # Get Operating System Name
11 Write-Log -Message 'Querying system for operating system information'
12 $OS = Get-WmiObject -ClassName win32_operatingsystem
13 Write-Log -Message "$($OS.Caption)"
14 $OS.Caption
```

Added formatted timestamp.

Here's the log:

```
2015-11-22 22:17:21 Querying system for operating system information
2015-11-22 22:17:21 Microsoft Windows 8.1 Enterprise
```

Better right? We can do better. We should offer a way to differentiate the type of message we are writing to a log file. Sometimes we just have informational messages, but we also might have warning or error messages we would like to log. With our current log format, we can't tell what is an error or simply informational without reading the log file message line by line. Let's add a Level parameter.



```
1 Function Write-Log {  
2     Param(  
3         $Message,  
4         $LogFile = 'c:\logs\script.log',  
5         $Level  
6     )  
7  
8     Add-Content -Value "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') $Level $Message" -Path $LogFile  
9 }  
10  
11 # Get Operating System Name  
12 Write-Log -Message 'Querying system for operating system information' -Level Info  
13 $OS = Get-WmiObject -ClassName win32_operatingsystem  
14 Write-Log -Message "$($OS.Caption)" -Level Info  
15 $OS.Caption
```

Level Parameter added.

Log File:

```
2015-11-23 10:02:23 Info Querying system for operating system information  
2015-11-23 10:02:23 Info Microsoft Windows 8.1 Enterprise
```

Now that's starting to look like a decent log file that we can search through using `grep` or `Select-String`.

Let's promote this helper function to an advanced function, add some logic, and sanity checks so that we can have a versatile logging tool for our PowerShell toolbox.



<#

.Synopsis

Write-Log writes a message to a specified log file with the current time stamp.

.DESCRIPTION

The Write-Log function is designed to add logging capability to other scripts. In addition to writing output and/or verbose you can write to a log file for later debugging.

.NOTES

Created by: Jason Wasser @wasserja

Modified: 11/24/2015 09:30:19 AM

Changelog:

- * Code simplification and clarification - thanks to @juneb_get_help
- * Added documentation.
- * Renamed LogPath parameter to Path to keep it standard - thanks to @JeffHicks
- * Revised the Force switch to work as it should - thanks to @JeffHicks

To Do:

- * Add error handling if trying to create a log file in a inaccessible location.
- * Add ability to write \$Message to \$Verbose or \$Error pipelines to eliminate duplicates.

.PARAMETER Message

Message is the content that you wish to add to the log file.

.PARAMETER Path

The path to the log file to which you would like to write. By default the function will create the path and file if it does not exist.

.PARAMETER Level

Specify the criticality of the log information being written to the log (i.e. Error, warning, Informational)

.PARAMETER NoClobber

Use NoClobber if you do not wish to overwrite an existing file.

.EXAMPLE

Write-Log -Message 'Log message'

Writes the message to c:\Logs\PowerShellLog.log.



.EXAMPLE

```
Write-Log -Message 'Restarting Server.' -Path c:\Logs\Scriptoutput.log
```

Writes the content to the specified log file and creates the path and file specified.

.EXAMPLE

```
Write-Log -Message 'Folder does not exist.' -Path c:\Logs\Script.log -Level Error
```

Writes the message to the specified log file as an error message, and writes the message to the error pipeline.

.LINK

<https://gallery.technet.microsoft.com/scriptcenter/Write-Log-PowerShell-999c32d0>

#>

```
function Write-Log
```

```
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,
                    ValueFromPipelineByPropertyName=$true)]
        [ValidateNotNullOrEmpty()]
        [Alias("LogContent")]
        [string] $Message,

        [Parameter(Mandatory=$false)]
        [Alias('LogPath')]
        [string] $Path='C:\Logs\PowerShellLog.log',

        [Parameter(Mandatory=$false)]
        [ValidateSet("Error", "Warn", "Info")]
        [string] $Level="Info",

        [Parameter(Mandatory=$false)]
        [switch] $NoClobber
    )
}
```

```
Begin
```



```
{
    # Set VerbosePreference to Continue so that verbose messages are displayed.
    $VerbosePreference = 'Continue'
}

Process
{

    # If the file already exists and NoClobber was specified, do not write to the log.
    if ((Test-Path $Path) -AND $NoClobber) {
        Write-Error "Log file $Path already exists, and you specified NoClobber. Either
delete the file or specify a different name."
        Return
    }

    # If attempting to write to a log file in a folder/path that doesn't exist create
the file including the path.
    elseif (!(Test-Path $Path)) {
        Write-Verbose "Creating $Path."
        $NewLogFile = New-Item $Path -Force -ItemType File
    }

    else {
        # Nothing to see here yet.
    }

    # Format Date for our Log File
    $FormattedDate = Get-Date -Format "yyyy-MM-dd HH:mm:ss"

    # Write message to error, warning, or verbose pipeline and specify $LevelText
    switch ($Level) {
        'Error' {
            Write-Error $Message
            $LevelText = 'ERROR:'
        }
    }
}
```



```
'Warn' {  
    Write-Warning $Message  
    $LevelText = 'WARNING:'  
}  
  
'Info' {  
    Write-Verbose $Message  
    $LevelText = 'INFO:'  
}  
  
}  
  
# Write log entry to $Path  
"$FormattedDate $LevelText $Message" | Out-File -FilePath $Path -Append  
}  
End  
{  
}  
}
```

Stepping through the code, we see our comment-based help for the "next guy" (including myself who tends to forget after a few weeks how my own code works). Then, I added the magic keyword `CmdletBinding` to transform our lowly function into an advanced function with cmdlet superpowers. I attempt to use standard parameter names where applicable and use the `ValidateSet` attribute on the `Level` parameter to ensure the user enters a proper level.

The `Begin` block sets the `VerbosePreference` to `Continue` so that our `Write-Verbose` statements are sent to the Verbose pipeline.

The `Process` block starts by creating a new log file, including requisite path, ensuring that we don't clobber (overwrite) the file when the `NoClobber` switch is present.

Based upon the selected `Level`, with `Info` being the default, we then write the message to our log file as well as send the supplied message to the Verbose, Warning, or Error pipeline where applicable.

That's great, but how do I use this in my scripts? I'm glad you asked. After the function is loaded, either via dot-sourcing or added to a module, you can begin adding the `Write-Log` function to your scripts.



Example:

```
1 $LogPath = c:\logs\w32timescript.log
2 Write-Log -Message 'Beginning w32time Restart Script' -Path $LogPath
3 Write-Log -Message 'Restarting Windows Time Service' -Path $LogPath
4 Restart-Service -Name w32time
5 Write-Log -Message 'End of Script' -Path $LogPath
```

Sample Log:

```
2015-11-23 10:16:48 INFO: Beginning w32time Restart Script
2015-11-23 10:16:48 INFO: Restarting Windows Time Service
2015-11-23 10:16:48 INFO: End of Script
```

Although this is just a basic example of how you could use Write-Log in your scripts, you can incorporate it in a way that works best for you. I found it helpful to include a header, footer, and log rotation as part of my template script.

```
29 Begin
30 {
31     # Begin Logging
32     Write-Log "-----" -Path $LogFileName
33     Write-Log "Beginning $($MyInvocation.InvocationName) on $(Senv:COMPUTERNAME) by Senv:USERDOMAIN\Senv:USERNAME" -Path $LogFileName
34 }
35 Process
36 {
37 }
38 End
39 {
40     # Clean up
41     Write-Log "$($MyInvocation.InvocationName) complete." -Path $LogFileName -Level Info
42     Write-Log "-----" -Path $LogFileName -Level Info
43     # Rotate Log file
44     if (Test-Path $LogFileName) {
45         $TimeStamp = Get-Date -Format "yyyyMMddhhmmss"
46         $LogFilePath = Get-ChildItem -Path $LogFileName
47         Rename-Item $LogFilePath -NewName "$($LogFilePath.BaseName)-$TimeStamp.log"
48     }
49 }
```

PowerTip: Read more about setting default parameter values [\\$PSDefaultParameterValues](#) so that you don't have to specify the Path every time you call the Write-Log function (#Requires - Version 3.0).

Here's an example of a log generated from my [Set-IPAddress](#) function which shows the name of the script, where it was run, and who ran it (names were changed to protect the innocent).

Example Log Output:

```
2015-04-29 12:23:18 INFO: -----
2015-04-29 12:23:18 INFO: Beginning Set-IPAddress on WORKSTATION02 by
DOMAIN\username
2015-04-29 12:23:21 INFO: Current IP Address(es): 10.146.2.160
2015-04-29 12:23:21 INFO: Current Subnet(s): 255.255.255.0
2015-04-29 12:23:21 INFO: Setting IP Address(es) for Local Area Connection
2: 10.146.2.160 10.146.2.159
2015-04-29 12:23:21 INFO: Subnet Mask(s): 255.255.255.0 255.255.255.0
2015-04-29 12:23:24 INFO: No gateway specified.
2015-04-29 12:23:24 INFO: No DNS specified.
2015-04-29 12:23:24 INFO: New IP Configuration: 10.146.2.159 10.146.2.160
2015-04-29 12:23:24 INFO: New Subnet Configuration: 255.255.255.0
255.255.255.0
2015-04-29 12:23:24 INFO: New Gateway Configuration: 10.146.2.1
2015-04-29 12:23:24 INFO: New DNS Configuration: 10.146.2.68 10.146.2.69
2015-04-29 12:23:25 INFO: Current IP Address(es): 10.146.2.159 10.146.2.160
```



```
2015-04-29 12:23:25 INFO: Current Subnet(s): 255.255.255.0 255.255.255.0
2015-04-29 12:23:25 INFO: Setting IP Address(es) for Local Area Connection
2: 10.146.2.159 10.146.2.160 10.146.2.160
2015-04-29 12:23:25 INFO: Subnet Mask(s): 255.255.255.0 255.255.255.0
255.255.255.0
2015-04-29 12:23:25 ERROR: Error setting IP: 70
2015-04-29 12:23:25 INFO: No gateway specified.
2015-04-29 12:23:25 INFO: No DNS specified.
2015-04-29 12:23:25 INFO: New IP Configuration: 10.146.2.159 10.146.2.160
2015-04-29 12:23:25 INFO: New Subnet Configuration: 255.255.255.0
255.255.255.0
2015-04-29 12:23:25 INFO: New Gateway Configuration: 10.146.2.1
2015-04-29 12:23:25 INFO: New DNS Configuration: 10.146.2.68 10.146.2.69
2015-04-29 12:23:25 INFO: Pausing to allow the network card to apply the new
settings.
2015-04-29 12:23:30 INFO: Set-IPAddress complete.
2015-04-29 12:23:30 INFO: -----
```

If you think you can use this Write-Log function, feel free to [download](#) it and add it to your tool belt. If you see ways you could improve the script, feel free to send along your suggestions via [Twitter](#) or improve it yourself on [github](#). If you don't like the script, feel free to [download](#) it anyway, and then you can have the pleasure of putting it in your Recycle Bin.

In summary, we find, at times, that our scripts, functions, and modules need some basic text logging, especially if the code is going to modify a system configuration or data. We walked through Write-Log, an advanced function that we can use to create readable text logs.



Slicing and Dicing Log Files

Thom Schumacher

Follow on Twitter @driberif



To begin to slice or dice a log file, first, we need to get it into a variable.

To open a log file or text file, open a PowerShell prompt and run a [Get-Content](#) command. For example, this command uses Get-Content to get the content of the License.txt file in the local path.

```
PS Ps:\> get-content .\LICENSE.txt
Copyright (c) 2010-2013 Keith Dahlby and contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Now that we know how to use Get-content, we can save the results in whatever variable we choose. In my case, I chose \$file.

```
PS Ps:\> $file = get-content .\LICENSE.txt
```

Now that we have the contents of our file in the **\$file** variable, we can begin to chop this file up.

To find out how many lines are in our file, type in **\$file.Count**. This expression uses the Count property of the string array in \$file to get the count of array elements. By default, Get-Content reads the contents of the file into a System.Array type. Each line is placed in a separate array element. To see the first line of our file, type:

```
PS Ps:\> $file[0]
Copyright (c) 2010-2013 Keith Dahlby and contributors
```

As you can see from the first image, this is the first line of our License.txt file.

Now, if we want to search through this line for matches on the word 'and', we can use the [Match operator](#).



```
PS Ps:\> $file[0] -match 'and'
True

PS Ps:\> $Matches

Name                                Value
----                                -
0                                    and
```

The Match operator automatically populates the \$matches variable with the first match of the item specified after the [-Match operator](#). If we had used the [-Match operator](#) on the entire file our results would have shown us the number of matches in a result set. For Brevity the count of the mached value and is shown.

```
PS Ps:\> ($file -match 'and').count
4

PS Ps:\> ($file).count
10
```

As you can see our total count for the \$file is 10 and the matched number is 4.

If we wanted to replace the word and, instead of using a match, we could use the [-Replace operator](#).

```
PS Ps:\> $file[0] -replace 'and','replaced And'
Copyright (c) 2010-2013 Keith Dahlby replaced And contributors
```

As you can see, the 'and' in the string is now replaced with 'replaced And'.

To do the same operation on the entire file use the -Replace operator

```
PS ps:\> $file -replace 'and', 'replaced And'
Copyright (c) 2010-2013 Keith Dahlby replaced And contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software replaced And associated documentation files (the "Software
"), to deal in the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, r
eplaced And/or sell copies of the Software, replaced And to permit persons to whom the
Software is furnished to do so, subject to the following conditio
ns:

The above copyright notice replaced And this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITN
ESS FOR A PARTICULAR PURPOSE replaced And NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
```



LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

To save the file, pipe the results to the [Set-Content](#) cmdlet with the path of where you want it saved.

To read the file in one string so there is no array, specify the Raw dynamic parameter. It will save the contents of the file as a single string in the variable you specify.

```
$file = get-content -raw .\LICENSE.txt
```

To separate this file into two chunks, we'll use the [Substring method](#) of string.

```
$file.substring(0,542) | out-file .\LICENSE1.txt
```

The first value in [Substring method](#) tells Substring the position of the character in our **\$file** string we want to start with (beginning at 0) and the second number is the position of the character where we want to end. So now we end up with half of our original License1.txt file.

It's even easier to get the second half of the file:

```
$file.substring(542) | out-file .\LICENSE542.txt
```

Substring starts from position 542 and goes to the end of the string.

Using a license file is not very practical. However, we did figure out how to read the file into a variable, match items in the result, get a count and split the file in two, now let's move onto logs that IIS produces.

These files can be found on C:\inetpub\logs\logfiles\w3svc(x), where (x) is the number of the site that is logging data for the site.

After inspecting logs for this web server we've discovered that there are a great number of small files with not much information in them. Since this is just a local desktop we haven't worked much with IIS here so the first thing we'll do is to consolidate all these log files into a single log and then begin the process of drilling into the information in the consolidated log file.



```
PS C:\inetpub\logs\logfiles\w3svc1> dir
```

```
Directory: C:\inetpub\logs\logfiles\w3svc1
```

Mode	LastWriteTime	Length	Name
-a----	3/7/2015 10:11 AM	338	u_ex150307.log
-a----	6/14/2015 7:03 PM	5522	u_ex150615.log
-a----	6/15/2015 8:27 PM	472	u_ex150616.log
-a----	8/17/2015 9:19 PM	593	u_ex150818.log
-a----	8/19/2015 6:43 AM	2616	u_ex150819.log
-a----	8/19/2015 7:36 PM	1516	u_ex150820.log
-a----	8/20/2015 6:08 PM	761	u_ex150821.log
-a----	8/23/2015 10:36 AM	3666	u_ex150822.log
-a----	8/24/2015 5:42 PM	761	u_ex150825.log
-a----	8/26/2015 4:04 PM	2117	u_ex150826.log
-a----	8/30/2015 8:59 AM	2574	u_ex150829.log
-a----	8/30/2015 4:48 PM	1366	u_ex150830.log
-a----	9/1/2015 7:45 PM	2458	u_ex150901.log
-a----	9/2/2015 3:50 PM	769	u_ex150902.log
-a----	9/2/2015 9:26 PM	2131	u_ex150903.log
-a----	9/9/2015 7:24 AM	1266	u_ex150906.log
-a----	9/11/2015 3:57 PM	860	u_ex150910.log
-a----	9/23/2015 7:54 PM	1074	u_ex150923.log
-a----	9/24/2015 8:27 PM	763	u_ex150925.log
-a----	9/28/2015 8:53 PM	1457	u_ex150929.log
-a----	10/3/2015 8:46 AM	769	u_ex151001.log
-a----	11/9/2015 7:52 PM	1469	u_ex151110.log

As you can see, we have

```
(dir).count  
22
```

...22 files in our IIS logging folder. Get-Content takes a Path value by property name. So, you can pipe Get-ChildItem to Get-Content without the need for a foreach loop.

```
dir | get-content | set-content iislog.txt
```

Now that we have the content of all my files in iislog.txt, we want to filter out anything that isn't really a log. This can be done with a regular expression. The one we'll use is this filter:

```
"(\d+)-(\d+)-(\d+) (\d+):(\d+):(?:\d+) ::"
```

This will match on items like this one: 2015-11-10 02:36:54 ::1

```
(get-content .\iislog.txt) -match "(\d+)-(\d+)-(\d+) (\d+):(\d+):(?:\d+) ::" | Set-Content  
filtered_iislog.txt
```

Now since we have our log file trimmed and the excess items that were at the top of the log chopped we can now begin the process of discovering interesting items about our log file.



How many 404's has this web server experienced? We can find the number of 404's by reading our log file back into a variable and using the [-Match operator](#) to find our 404's

```
PS C:\inetpub\logs\LogFiles\W3SVC1> $iislog = get-content .\filteredIislog.txt
PS C:\inetpub\logs\LogFiles\W3SVC1> $404Logs = $iislog -match ' 404 '
PS C:\inetpub\logs\LogFiles\W3SVC1> $404Logs.count
16
PS C:\inetpub\logs\LogFiles\W3SVC1> ($iislog -match ' 404 ').count
16
```

As we can see from the scripting above we used the [Get-Content](#) command to get the file. Then we used the [-Match operator](#) to get us the number of matches. Since we didn't use the `-raw` switch on the [Get-Content](#) command the result set is scalar allowing matches to give us the number of matches it found in the collection. Here are the first 2 matches in the results of the scalar match:

```
PS C:\inetpub\logs\LogFiles\W3SVC1> $iislog -match ' 404 ' | select -first 2
2015-06-15 00:05:31 ::1 GET
/t/curl_http...clusterexternalnic_8090.storage.172.25.3.20.nodestat - 80 - ::1
Mozilla/5.0+(Windows+NT+6.3;+WOW64)+ApplewebK
it/537.36+(KHTML,+like+Gecko)+Chrome/43.0.2357.124+Safari/537.36 - 404 3 50 496

2015-06-15 00:05:31 ::1 GET /favicon.ico - 80 - ::1
Mozilla/5.0+(Windows+NT+6.3;+WOW64)+ApplewebKit/537.36+(KHTML,+like+Gecko)+Chrome/43.0.2357.
124+Safa
ri/537.36
http://localhost/t/curl_http...clusterexternalnic_8090.storage.172.25.3.20.nodestat 404 0 2
4
```

Now that we have the number of 404's in our log file we can see what queries resulted in a 404. To achieve this we'll use the first result set to figure out what we need to select with a Regular expression to get the query names out of our log file.

```
S C:\inetpub\logs\LogFiles\W3SVC1> $iislog -match ' 404 ' | select -first 1
2015-06-15 00:05:31 ::1 GET
/t/curl_http...clusterexternalnic_8090.storage.172.25.3.20.nodestat - 80 - ::1
Mozilla/5.0+(Windows+NT+6.3;+WOW64)+ApplewebK
it/537.36+(KHTML,+like+Gecko)+Chrome/43.0.2357.124+Safari/537.36 - 404 3 50 496
```

The starting character in log is a / and the ending characters are (a space followed by a -).

So we'll re-use our regular expression used earlier and add `::1 GET` and add it to a `$404Filtered` Variable.



```
$404filtered = ($iislog -match ' 404 ') -REPLACE "(\d+)-(\d+)-(\d+) (\d+):(\d+):(?:\d+) ::1 GET" , ''
```

Note: This works for this log file because the address written after the :: is a local loopback address. So you may have to change this based on your scenario.

Below is the 1st item in the array of 16 based on the -replace String above.

```
PS C:\inetpub\logs\LogFiles\W3SVC1> $404filtered | select -first 1
/t/curl_http...clusterexternalnic_8090.storage.172.25.3.20.nodestat - 80 - ::1
Mozilla/5.0+(Windows+NT+6.3;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Geck
o)+Chrome/43.0.2357.124+Safari/537.36 - 404 3 50 496
```

Now we need to remove the string on the end to give us our failing page. Again we can do this with a regular expression we'll use this expression: '-.*'

```
PS C:\inetpub\logs\LogFiles\W3SVC1> $404filtered -replace '-.*',''
/t/curl_http...clusterexternalnic_8090.storage.172.25.3.20.nodestat
/favicon.ico
/t/console_print.css
```

For brevity sake we've only selected the first 3 failing pages. Now we can take these files to the developers and ask them if they should be here or if some other file in the website is hitting them that shouldn't.

As you can see you can do a number of things with very few commands to split / join or search through logs and drill into the information that will help you the most. PowerShell has many other methods and commands that can be used to search through files and operate on them. In my next blog post I'll go through some of the other methods and ways that will make dealing with logs that much easier.

For more really good examples, see this blog post, which was used as a primary source for the IIS regular expressions:

<https://infracloud.wordpress.com/2015/09/28/iis-log-parsing-using-powershell/>

For many other regular expressions that can be used with PowerShell see this website.

<http://regexlib.com/Search.aspx?k=iis&c=-1&m=-1&ps=20>

Until next time...**Keep on Scripting!**

Thom



PowerShell logging in the Windows Event log

Jaap Brasser

Follow on Twitter @jaap_brasser



An important part of PowerShell scripting is error handling, one of the main differences between a script and a one-liner for me personally is error handling and dealing with exceptions that might occur. For more information on error handling please refer to the previous PSBlogWeek articles where Boe Prox dives into use Try-Catch in order to catch specific errors.

When moving your scripts from into a production environment logging becomes more important, initially using plain text files for logging might be an appropriate solution. Another option however is writing logging information to the Windows Event Log. This has the benefit of being the centralized location where most logging takes place.

Writing to the event log is relatively simple, the Write-EventLog cmdlet can be used for this purpose:

```
PS C:\Users\JaapBrasser> Write-EventLog -LogName Application -Source PowerShell -eventID 12345 -EntryType Information -Message 'Script started'
Write-EventLog : The source 'PowerShell' is not registered in log 'Application'. (It is registered in log 'Windows PowerShell'.) " The Source and Log properties must be matched, or you may set Log to the empty string, and it will automatically be matched to the Source property.
At line:1 char:1
+ Write-EventLog -LogName Application -Source PowerShell -eventID 12345 ...
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Write-EventLog], Exception
+ FullyQualifiedErrorId : The source 'PowerShell' is not registered in log 'Application'. (It is registered in log 'Windows PowerShell'.) " The Source and Log properties must be matched, or you may set Log to the empty string, and it will automatically be matched to the Source property.,Microsoft.PowerShell.Commands.WriteEventLogCommand
```

By taking a look at the error message we see that the PowerShell event source is not registered with the application log, in order to resolve this, we can write to the Windows PowerShell event log instead:

```
PS C:\Users\JaapBrasser> Write-EventLog -LogName 'Windows PowerShell' -Source PowerShell -eventID 12345 -EntryType Information -Message 'Script started'
```



Notice how the command now successfully executes without the error message, this is because it is important for the script

Alternatively, it is also possible to specify a custom event provider, and register this to the correct event log as such:

```
PS C:\Users\JaapBrasser> New-EventLog -LogName 'windows PowerShell' -Source AwesomeScript
PS C:\Users\JaapBrasser> Write-EventLog -LogName 'windows PowerShell' -Source AwesomeScript
-eventID 12345 -EntryType Information -Message 'Script started'
```

It is also possible to create an entire separate event log for your script logging, also by using the New-EventLog cmdlet:

```
PS C:\Users\JaapBrasser> New-EventLog -LogName 'Scripts' -Source 'Your Script'
Write-EventLog -LogName 'Scripts' -Source 'Your Script' -EventId 12345 -EntryType
Information -Message 'Script started'
New-EventLog -LogName 'Scripts' -Source 'Your Script2'
Write-EventLog -LogName 'Scripts' -Source 'Your Script2' -EventId 12345 -EntryType
Information -Message 'Script started'
```

To build upon this, it would be possible to create a script that logs when the script starts, executes an action, when an error occurs and when the script ends.

```
PS C:\Users\JaapBrasser> begin {
    $EventHashInformation = @{
        LogName     = 'Scripts'
        Source      = 'Your Script'
        EventId     = 30000
        EntryType   = 'Information'
    }
    $EventHashWarning = @{
        LogName     = 'Scripts'
        Source      = 'Your Script'
        EventId     = 40000
        EntryType   = 'Warning'
    }
    Write-EventLog @EventHashInformation -Message 'Script started'
}

process {
    try {
        Get-CimInstance -ClassName Win32_Bios -ErrorAction Stop
        Write-EventLog @EventHashInformation -Message 'Successfully queried Win32_Bios'
    } catch {
        Write-EventLog @EventHashWarning -Message 'Error occurred while querying Win32_Bios'
    }
}

end {
    Write-EventLog @EventHashInformation -Message 'Script finished'
}
```

Using this method of logging allows for a dynamic form of logging where information such as the timestamp are automatically added to the information. A nice feature of using the event log is that it also allows for exporting the information directory to xml, for example:



```
PS C:\Users\JaapBrasser> Get-WinEvent -FilterHashtable @{
    LogName = 'Scripts'
} | ForEach-Object {$_.ToXml()}
```

Because this type of logging can quickly fill up the event logs it is important to set the limits and the type of logging the event logs can do. In order to verify the current configuration, we can use the Get-EventLog cmdlet:

```
PS C:\Users\JaapBrasser> Get-EventLog -List | Where-Object {$_.LogDisplayName -eq 'Scripts'}
|
Select-Object -Property Log,MaximumKilobytes,MinimumRetentionDays,OverflowAction

Log      MaximumKilobytes MinimumRetentionDays OverflowAction
---      -
Scripts      512                7 OverwriteOlder
```

From this we gather that the Minimum retention of this log should be seven days and that only events older than 7 days will be discarded. If the event log is at the maximum size and there are no events older than 7 days to be discarded the latest event will be discarded and will not be written to the script. For more information on this subject please refer to the following MSDN article:

[OverflowAction](#)

Member name	Description
DoNotOverwrite	Indicates that existing entries are retained when the event log is full and new entries are discarded.
OverwriteAsNeeded	Indicates that each new entry overwrites the oldest entry when the event log is full.
OverwriteOlder	Indicates that new events overwrite events older than specified by the MinimumRetentionDays property value when the event log is full. New events are discarded if the event log is full and there are no events older than specified by the MinimumRetentionDays property value.

Based on the information in the article, we decide to increase the maximum size of log to 20 MB and to allow overwriting as needed, this is also referred to as circular logging:

```
PS C:\Users\JaapBrasser> Limit-EventLog -MaximumSize 20MB -Logname Scripts -OverflowAction
OverwriteAsNeeded
Get-EventLog -List | Where-Object {$_.LogDisplayName -eq 'Scripts'} |
Select-Object -Property Log,MaximumKilobytes,MinimumRetentionDays,OverflowAction

Log      MaximumKilobytes MinimumRetentionDays OverflowAction
---      -
Scripts      20480                0 OverwriteAsNeeded
```



If all this information is combined we have learned the following things in this article:

- How to write information to the event log
- How to register an event source to an event log
- How to create a new event log
- How to configure an existing event log

I hope this was informative and do not forget to read through the other #PSBlogWeek posts.



Reading the Event Log with Windows PowerShell

Adam Platt

Follow on Twitter @platta



Whether it's an error report, a warning, or just an informational log, one of the most common places for Windows to write logging information is to the event logs. There are tons of reasons to open up Event Viewer and peruse the event logs. Some of the things I've had to do recently include:

- Checking why a service failed to start at boot time.
- Finding the last time a user logged into the computer and who it was.
- Checking for errors after an unexpected restart.

Although Event Viewer is a handy tool, given today's IT landscape, we should always be looking for more efficient ways to consume data sources like the event logs. That's why we're going to take a look at the `Get-WinEvent` cmdlet in PowerShell. This cmdlet is a powerful and flexible way of pulling data out of the event logs, both in interactive sessions and in scripts.

Get-WinEvent: The Basics

Before we get started, don't forget to launch your PowerShell session as Administrator, since some of the logs (like Security) won't be accessible otherwise.

Like any good PowerShell cmdlet, `Get-WinEvent` has excellent help, including fourteen different examples. We're going to review a lot of the important points, but you can always use `Get-Help Get-WinEvent -Online` to bring the full help up in a browser window.

Without any parameters, `Get-WinEvent` returns information about every single event in every single event log. To get to the specific events you want, you need to pass one or more parameters to filter the output. Here are the most common parameters of `Get-WinEvent` and what they do:

- `LogName` – Filters events in the specified log (think Application, Security, System, etc.).
- `ProviderName` – Filters events created by the specified provider (this is the Source column in Event Viewer).
- `MaxEvents` – Limits the number of events returned.



- `Oldest` – Sorts the events returned so that the oldest ones are first. By default, the newest, most recent events are first.

So, by using these basic parameters, we can build commands that do things like:

Get the last 10 events from the Application log.

```
Get-WinEvent -LogName Application -MaxEvents 10
```

Get the last 5 events logged by Outlook.

```
Get-WinEvent -ProviderName Outlook -MaxEvents 5
```

Get the oldest 50 events from the Application log.

```
Get-WinEvent -LogName Application -MaxEvents 50 -Oldest
```

Discovering

The event logs have grown up quite a bit since the days when Application, Security, and System were the only logs to look through. Now there are folders full of logs. How are you going to know how to reference those logs when you're using `Get-WinEvent`? You use the `ListLog` and `ListProvider` parameters.

For example, consider the log that PowerShell Desired State Configuration (DSC) uses. In Event Viewer, that log is located under:

```
Applications and Services Logs\Microsoft\Windows\Desired State  
Configuration\Operational
```

If I pass that name to the `LogName` parameter of `Get-WinEvent`, I get an error. To find the name I need to use, I run the command:

```
Get-WinEvent -ListLog *PowerShell*
```

```
PS C:\Windows\system32> Get-WinEvent -ListLog *PowerShell*

LogMode  MaximumSizeInBytes RecordCount LogName
-----
Circular      15728640          18473 Windows PowerShell
Circular      1052672           0 Microsoft-Windows-PowerShell-DesiredStateConfiguration-FileDownloadManager/Operational
Retain       1048985600        Microsoft-Windows-PowerShell/Admin
Circular      15728640          26944 Microsoft-Windows-PowerShell/Operational
```

and see that there are a couple of options. Although there is one that mentions Desired State Configuration, it doesn't sound like what I'm looking for. Let's see if the name uses the abbreviation DSC.

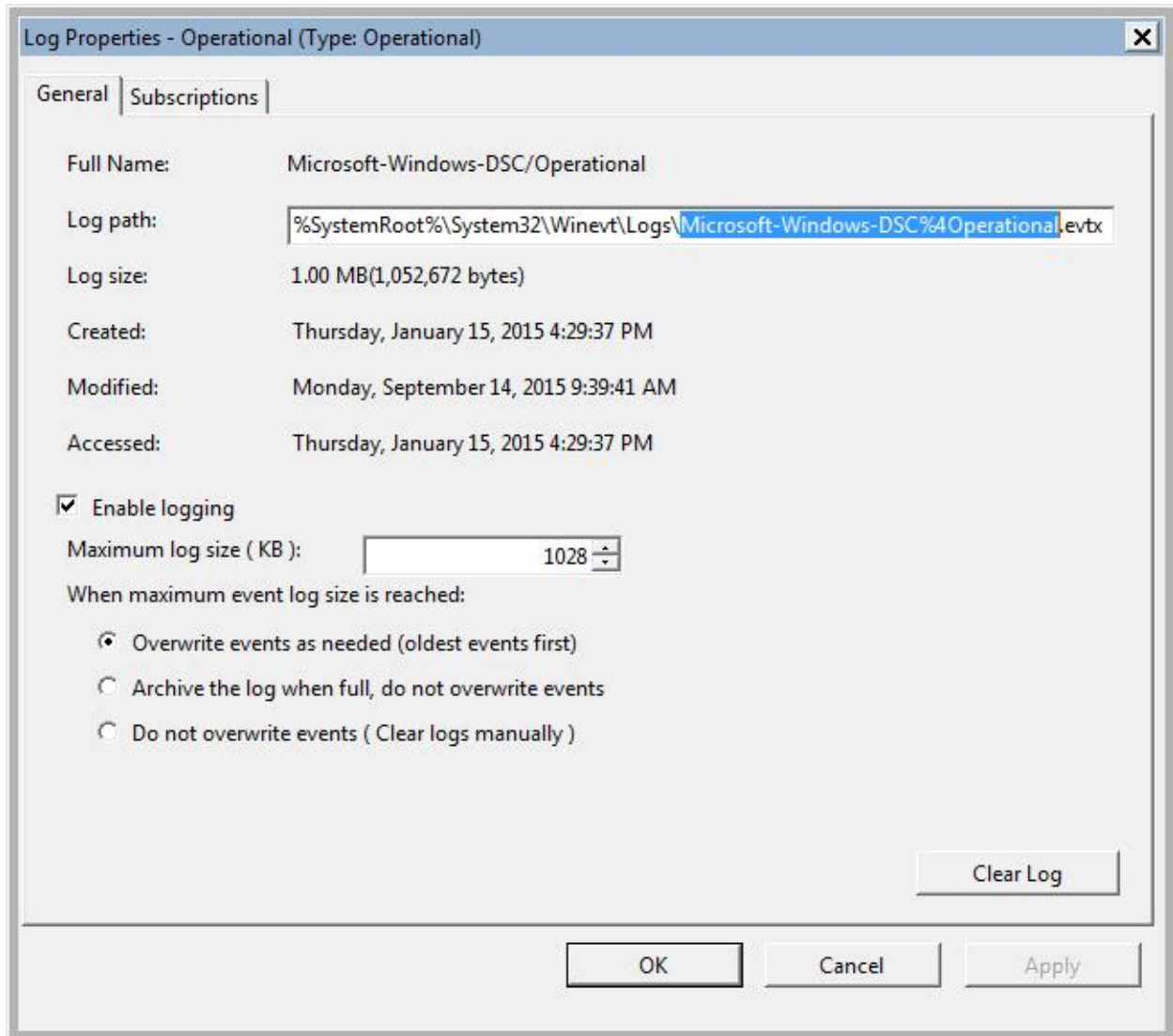
```
Get-WinEvent -ListLog *DSC*
```




```
PS C:\Windows\system32> Get-WinEvent -ListLog *DSC*

LogMode   MaximumSizeInBytes RecordCount LogName
-----
Circular   1052672           2000 Microsoft-Windows-DSC/Operational
```

That looks like a much better fit. To be sure, you can always check the properties of a log in Event Viewer and look at the file name.



Bingo.

You can pass `*` to the `ListLog` or `ListProvider` parameters to get all logs or all providers.



Filtering

The commonly used parameters are great for doing basic filtering, but if you're doing anything more complicated than a basic health check, you're going to need a more powerful search. That's where the `FilterHashtable` parameter comes in.

First, a quick review of hash table syntax. A hash table can be specified on one line or on multiple lines. Both statements below produce the same hash table.

```
$Hashtable1 = @{Key = "Value"; Foo = "bar"; Bat = "baz"}
```

```
$Hashtable2 = @{  
    Key = "Value"  
    Foo = "bar"  
    Bat = "baz"  
}
```

When passing a hash table to the `FilterHashtable` parameter, here are some of the keys you may find useful.

- `LogName` – Same as the `LogName` parameter
- `ProviderName` – Same as the `ProviderName` parameter
- `ID` – Allows you to filter on the Event ID
- `Level` – Allows you to filter on the severity of the entry (You have to pass a number: 4 = Informational, 3 = Warning, 2 = Error)
- `StartTime` – Allows you to filter out events before a certain Date/Time
- `EndTime` – Allows you to filter out events after a certain Date/Time (You can use `StartTime` and `EndTime` together)
- `UserID` – Allows you to filter on the user that created the event

If this does not provide the level of granularity that you need, don't forget that you can always pipe the results through `Where-Object` to further refine your results:

```
Get-WinEvent -FilterHashTable @{LogName = "Application"} -MaxEntries  
50 | Where-Object -Property Message -Like "*success*"
```

Extracting Meaning

Running one of the commands above might produce output that looks like this:



```
PS C:\Windows\system32> Get-WinEvent -ProviderName Outlook -MaxEvents 10

ProviderName: Outlook

TimeCreated      Id LevelDisplayName Message
-----
11/24/2015 3:23:47 PM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 3:22:45 PM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 3:22:29 PM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 3:21:23 PM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 3:20:42 PM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 3:20:23 PM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 3:19:59 PM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 3:19:44 PM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 11:23:40 AM 63 Information The Exchange web service request GetAppManifests succeeded.
11/24/2015 11:22:43 AM 63 Information The Exchange web service request GetAppManifests succeeded.
```

but what `Get-WinEvent` actually returns are objects of type

`[System.Diagnostics.Eventing.Reader.EventLogRecord]`. PowerShell is being nice and picking just four common properties to show us, but there are more properties that we can access by storing the results to a variable or using a cmdlet like `Select-Object`, `Format-Table`, `Format-List`, or even `Out-GridView`.

Here are some of the properties you might find useful:

- `Id` – Event ID
- `Level` – Numeric representation of the event level
- `LevelDisplayName` – Event level (Information, Error, Warning, etc.)
- `LogName` – Log name (Application, Security, System, etc.)
- `MachineName` – Name of the computer the event is from
- `Message` – Full message of the event
- `ProviderName` – Name of the provider (source) that wrote the event

You can also dig into the message data attached to the event. Event log messages are defined by the Event ID, so all events with the same event ID have the same basic message. Messages can also have placeholders that get filled in with specific values for each instance of that Event ID. These fill-in-the-blank values are called "insertion strings," and they can be very useful.

For example, let's say I grabbed an event out of the Application log and stored it in a variable `$x`. If I just examine `$x`, I can see the full message of the event is "The session '183c457c-733c-445d-b5d6-f04fc9623c8b' was disconnected".

```
PS C:\Windows\system32> $x

ProviderName: Citrix Desktop Service

TimeCreated      Id LevelDisplayName Message
-----
11/20/2015 4:55:25 PM 1049 Information The session '183c457c-733c-445d-b5d6-f04fc9623c8b' was disconnected.
```



This is where things get interesting. Since Windows Vista, event logs have been stored in XML format. If you run `(Get-WinEvent -ListLog Application).LogFilePath` you'll see the .evtx extension on the file. The `EventLogRecord` objects that `Get-WinEvent` returns have a `ToXml` method that I can use to get to the XML underneath the object; this is where the insertion string data is stored.

By converting the event to XML and looking at the insertion strings, I can get direct access to the value that was inserted into the message without having to parse the full message and extract it. Here is the code to do that:

```
([xml]$x.ToXml()).Event.EventData.Data
```

The `ToXml` method returns a string containing the XML, and putting the `[xml]` accelerator in front of it tells PowerShell to read that string and create an XML object from it. From there, I navigate the XML hierarchy to the place where the insertion string data is stored.

```
PS C:\Windows\system32> ([xml]$x.ToXml()).Event.EventData.Data
183c457c-733c-445d-b5d6-f04fc9623c8b
```

The messages for different Event IDs can have different numbers of insertion strings, and you may need to explore a little to figure out exactly how to pull the specific piece of data you're looking for, but all events for a given Event ID will be consistent. The example above uses a simple event on purpose, but one example of how I've used this in my automation is when pulling logon events from the Security log. Event 4624, which records a successful logon, contains insertion strings for which user is logging on, what domain they belong to, what kind of authentication they used, and more.

Remote Computers

All the cool stuff we just covered above? You can do all of that on remote targets as well. There are two parameters you can use to have `Get-WinEvent` get values from a remote computer:

- `ComputerName` – Lets you specify which computer to connect to remotely
- `Credential` – Lets you specify a credential to use when connecting, in case your current session is not running under an account with the appropriate permissions.

`Get-WinEvent` does not use PSRemoting to get remote data; just the Event Log Service. This requires TCP port 135 and a dynamically chosen port above 1024 to be open in the firewall.

Putting It All Together

There is more to discover in using the `Get-WinEvent` cmdlet and the data that it returns but, hopefully, this introduction serves as a thorough foundation for accessing the event logs from



PowerShell. These techniques for discovering, filtering, and extracting meaning from the event logs can be applied in an interactive PowerShell session or an automated script. They can also be used to read the event logs on your local machine or a remote target. It's hard to know what data you will be searching for to meet your requirements until you are faced with them, but one thing is for sure: the event logs probably have at least some of the data you need, and now you know how to get it!

References

- <https://technet.microsoft.com/en-us/library/hh849682.aspx>
- <http://www.mcbsys.com/blog/2011/04/powershell-get-winevent-vs-get-eventlog>
- <https://msdn.microsoft.com/en-us/library/system.diagnostics.eventing.reader.eventlogrecord.aspx>
- <http://blogs.technet.com/b/heyscriptingguy/archive/2014/06/03/use-filterhashtable-to-filter-event-log-with-powershell.aspx>



Building Logs for CMtrace with PowerShell

Adam Bertram

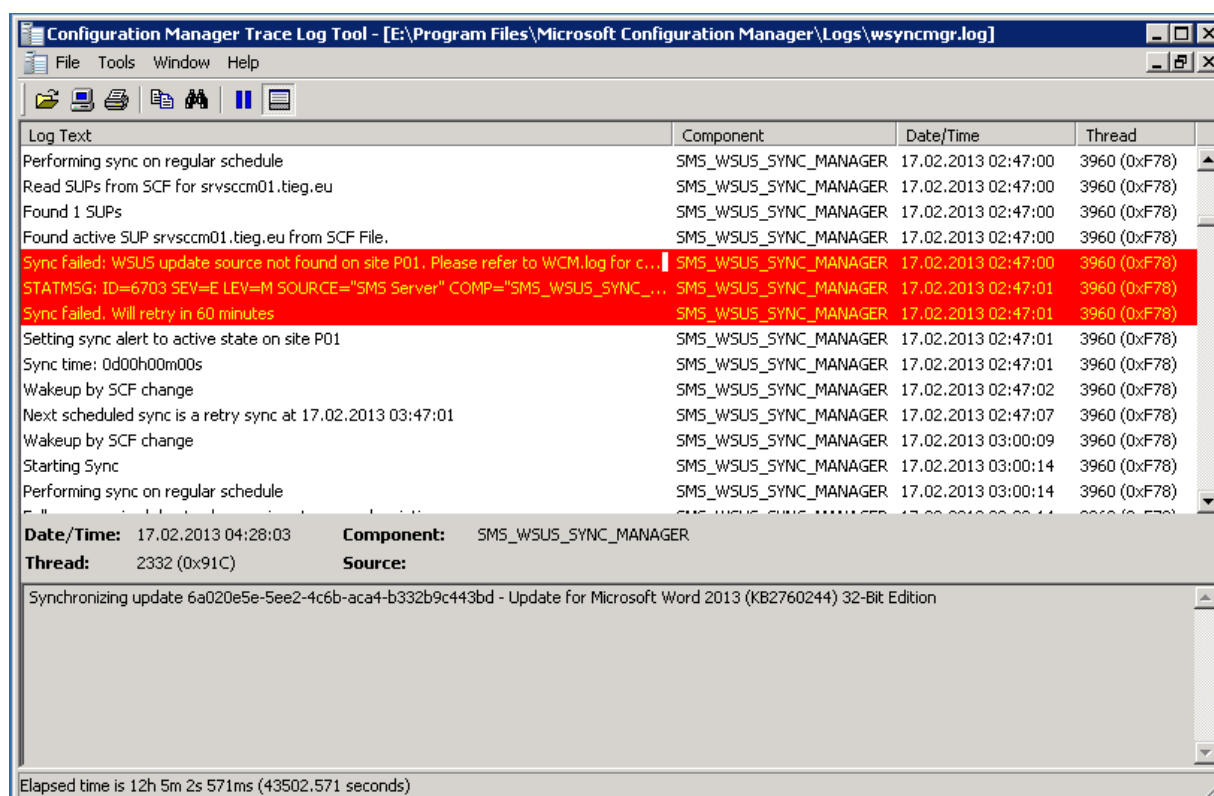
Follow on Twitter @adbertram



#PSBlogWeek

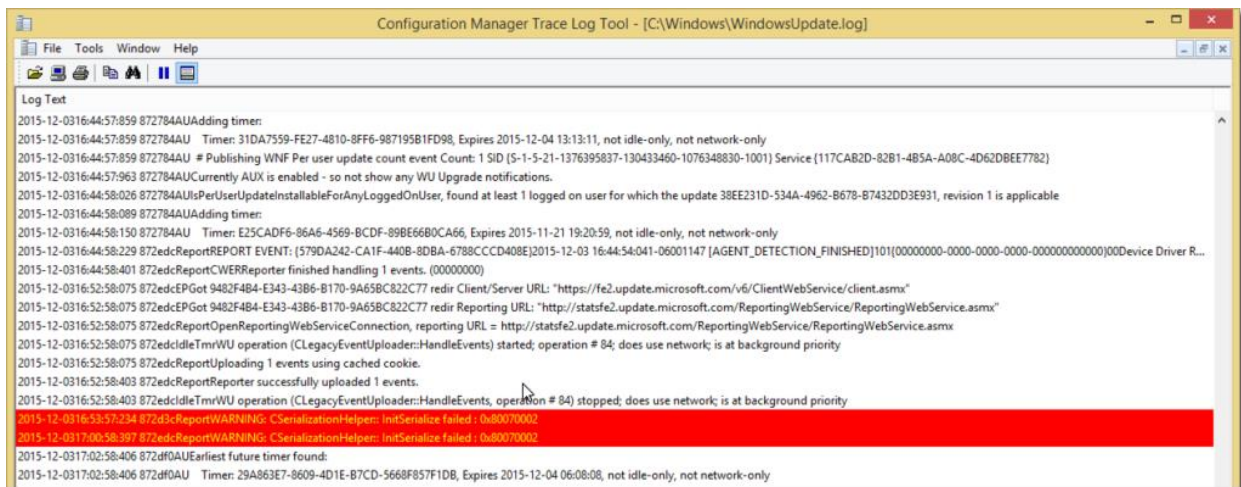
Building Logs for Cmtrace with PowerShell

In a previous life, I managed Microsoft's System Center Configuration Manager (SCCM) product. I was a SCCM ninja. One of the coolest things I got out of that was learning about the CMTrace log utility. Part of the [System Center Configuration Manager Toolkit](#), CMTrace is a log viewing utility that allows you to watch logs, in real time, as well as point out various potential problems through yellow and red highlighting, a sortable timestamp column and more.



Log Text	Component	Date/Time	Thread
Performing sync on regular schedule	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:00	3960 (0xF78)
Read SUPs from SCF for srvscm01.tieg.eu	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:00	3960 (0xF78)
Found 1 SUPs	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:00	3960 (0xF78)
Found active SUP srvscm01.tieg.eu from SCF File.	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:00	3960 (0xF78)
Sync failed: WSUS update source not found on site P01. Please refer to WCM.log for c...	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:00	3960 (0xF78)
STATMSG: ID=6703 SEV=E LEV=M SOURCE="SMS Server" COMP="SMS_WSUS_SYNC_...	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:01	3960 (0xF78)
Sync failed. Will retry in 60 minutes	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:01	3960 (0xF78)
Setting sync alert to active state on site P01	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:01	3960 (0xF78)
Sync time: 0d00h00m00s	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:01	3960 (0xF78)
Wakeup by SCF change	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:02	3960 (0xF78)
Next scheduled sync is a retry sync at 17.02.2013 03:47:01	SMS_WSUS_SYNC_MANAGER	17.02.2013 02:47:07	3960 (0xF78)
Wakeup by SCF change	SMS_WSUS_SYNC_MANAGER	17.02.2013 03:00:09	3960 (0xF78)
Starting Sync	SMS_WSUS_SYNC_MANAGER	17.02.2013 03:00:14	3960 (0xF78)
Performing sync on regular schedule	SMS_WSUS_SYNC_MANAGER	17.02.2013 03:00:14	3960 (0xF78)
Date/Time: 17.02.2013 04:28:03 Component: SMS_WSUS_SYNC_MANAGER			
Thread: 2332 (0x91C) Source:			
Synchronizing update 6a020e5e-5ee2-4c6b-aca4-b332b9c443bd - Update for Microsoft Word 2013 (KB2760244) 32-Bit Edition			
Elapsed time is 12h 5m 2s 571ms (43502.571 seconds)			

Just look at the beauty of the sortable columns and the red highlighting! At first, you might think that you can view any kind of text log in CMTrace and you'd be right. However, let's a look at the WindowsUpdate.log file in CMTrace.



Notice all the columns are gone? CMTrace will still view regular log files but you won't get some of the features that makes CMTrace great. You'll soon find that a text file has to be properly formatted in order to get all of those helpful columns to show up and to properly define which lines should be highlighted yellow vs. red. vs. nothing at all.

In today's post, I'd like to show you a couple functions called Write-Log and Start-Log . These functions were specifically built to record your script's activity to a log file which can then be read in CMtrace. By the end of this post, you will have a function that you can call in your scripts to build log files in a way for CMtrace to read them properly.

Start-Log

To prevent having to specify the same log file path over and over again I chose to create a function called Start-Log . This function is intended to be called at the top of your script. This function simply creates a text file and (the important part) sets a global variable called ScriptLogFilePath .

```
[CmdletBinding()]

param (

    [ValidateScript({ Split-Path $_ -Parent | Test-Path })]

    [string]$FilePath

)

try

{

    if (!(Test-Path $FilePath))
```



```
{  
  
    ## Create the log file  
  
    New-Item $FilePath -Type File | Out-Null  
  
}  
  
## Set the global variable to be used as the FilePath for all subsequent Write-Log  
## calls in this session  
  
$global:ScriptLogFilePath = $FilePath  
  
}  
  
catch  
  
{  
  
    Write-Error $_.Exception.Message  
  
}
```

This function is super-simple. However, is required to prevent us from having to pass -LogFile every, single time we need to call our Write-Log function in our scripts. By simply creating a global variable ahead of time, we can then simply call Write-Log and will know the log file path.

Write-Log

Once you've called Start-Log in your script, you are now able to run Write-Log to write log messages to the log file. Write-Log has two parameters; Message and LogLevel. Message is easy. That's simply what you'd like to write to the log. LogLevel requires some explaining. In order for CMTrace to highlight lines as red or yellow the line needs to be recorded a certain way. More specifically, it needs to have string like this: type="1". This type key can be 1,2 or 3. These indicate levels of severity in your script. For example, if I'd like to log a simple informational message, then that'd be a 1. If I'd like to log a more severe activity then I might use 2 which would get highlighted yellow. Finally, I might choose 3 if I'd like that line highlighted red in CMTrace.

```
param (  
  
    [Parameter(Mandatory = $true)]  
  
    [string]$Message,
```



```
[Parameter()]  
  
[ValidateSet(1, 2, 3)]  
  
[int]$LogLevel = 1  
  
)
```

Notice the `LogLevel` parameter? By default, it will set that to a 1 but you are always able to override that if necessary if you'd like to write some more severe activity that happens during your script's execution.

Next, you need that handy date/time column to show up right. To do this required a specific date/time format that is achieved by this string manipulation wizardry.

```
$TimeGenerated = "$(Get-Date -Format HH:mm:ss).$((Get-Date).Millisecond)+000"
```

Next is where I'll build a log line's template using all of the appropriate format that the line needs to have to show up correctly in CMtrace.

```
$Line = '<![LOG[{0}]LOG]><time="{1}" date="{2}" component="{3}" context="" type="{4}"  
thread="" file="">'
```

After you've got the template it's then a matter of building what's going to go in the `{}`'s. Here, I build an array which I will then pass into the `$Line` to replace all of our `{}`'s with real information.

```
$LineFormat = $Message, $TimeGenerated, (Get-Date -Format MM-dd-yyyy),  
"$($MyInvocation.ScriptName | Split-Path -Leaf):$($MyInvocation.ScriptLineNumber)", $LogLevel
```

These are in the same order as the `{}`'s above. `{0}` will get converted to `$Message`, `{1}` will get converted to `$TimeGenerated`, `{2}` will get converted to today's date and `{4}` will get converted by `$LogLevel`. Notice I skipped `{3}`? This is where I get all ninja on you. CMTrace has a component column that I never used much so I decided to make something out of it. I wanted to see the script's name and the line number in which `Write-Log` was called. This string: `"$($MyInvocation.ScriptName | Split-Path -Leaf):$($MyInvocation.ScriptLineNumber)"` is what makes that happen.

I then bring these two variables together using PowerShell's string formatting to build `$Line`.

```
$Line = $Line -f $LineFormat
```

It's then just a matter of writing `$Line` to a text file that's already been defined by `Start-Log`.

```
Add-Content -Value $Line -Path $ScriptLogFilePath
```

How it Works

Let's say I build a script that looks something like this called `LogDemo.ps1`:



```
Start-Log -FilePath C:\MyLog.log

Write-Host "Script log file path is [$ScriptLogFilePath]"

Write-Log -Message 'simple activity'

Write-Log -Message 'warning' -LogLevel 2

Write-Log -Message 'Error' -LogLevel 3
```

This script creates our log file at C:\MyLog.log and then proceeds to write 3 levels of severity to the log through using the LogLevel parameters I explained above.

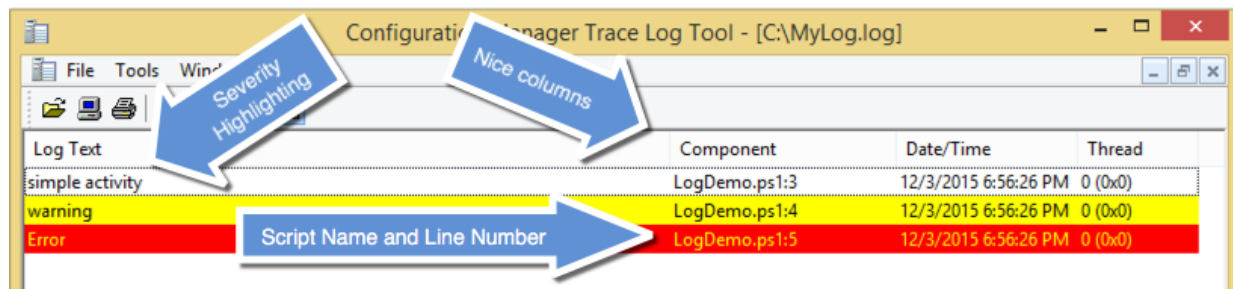
When I check out the output of this file with Get-Content it looks pretty ugly.

```
<![LOG[simple activity]LOG]!><time="18:56:26.307+000" date="12-03-2015"
component="LogDemo.ps1:3" context="" type="1" thread="" file="">

<![LOG[warning]LOG]!><time="18:56:26.307+000" date="12-03-2015" component="LogDemo.ps1:4"
context="" type="2" thread="" file="">

<![LOG[Error]LOG]!><time="18:56:26.307+000" date="12-03-2015" component="LogDemo.ps1:5"
context="" type="3" thread="" file="">
```

However, let's break this open in CMTrace and see what it looks like.



Isn't that beautiful?

Even if you're not an SCCM admin I highly recommend using CMtrace for all your log viewing needs. Once you've got the log files in the appropriate format (and you now have no excuse not to) simply open them up in CMtrace and observe the beauty of all that is CMtrace!



#PSBlogWeek Wrap-up

This week something special happened in the PowerShell community. It brought forth a second round of the popular blogging event known as #PSBlogWeek. Born from [Jeff Hicks'](#) idea nearly a year ago, #PSBlogWeek was started to get more people sharing their knowledge with the PowerShell community through blogging. Five willing volunteers took their valuable time and posted a technical, content-rich article on their blog all coordinated around a central theme.

#PSBlogWeek is special because it's simply not about one person and a blog. It's about a coordinated effort between multiple people not only to write the articles but also to promote them as well. #PSBlogWeek is a way to start giving back that knowledge you have stored in your noggin and is a great way to get some exposure to yourself and your writing. #PSBlogWeek is heavily promoted on Twitter by many of the top PowerShell personalities in the community as well as by the participants themselves. If you're looking for an excuse to start blogging on PowerShell and want to be sure people see it, #PSBlogWeek is a great opportunity.

This week's theme was logging. Big thanks to the following participants for their contributions. Be sure to check out all of the great posts if you haven't already and let the authors know via Twitter if they helped you learn a thing or two.

Monday (Jason Wasser [@wasserja](#)) – [Building Readable Text Log Files](#)

Tuesday (Thom Schumacher [@driberif](#)) – [Slicing and Dicing Text Log Files](#)

Wednesday (Jaap Brasser [@jaap_brasser](#)) – [PowerShell Logging in the Windows Event Log](#)

Thursday (Adam Platt [@platta](#)) – [Reading Events from Event Logs](#)

Friday (Adam Bertram [@adbertram](#)) – [Building Logs for CMTrace](#)

Editing by: [June Blender](#)

#PSBlogWeek events are not held on a particular schedule but if you're a blogger and want to be part of the next one, contact [Adam Bertram \(@adbertram\)](#) on Twitter. He will be gathering a list of interested participants to potentially be notified the next time the event is held.