Using PowerShell to discover information about your Microsoft SQL Servers

0

Posted by Mike Robbins on July 21, 2014 | Rate it

(9 votes)

| 36,487 views

I'm an infrastructure guy who supports many different products at multiple datacenters in an enterprise environment. One of those products is Microsoft SQL Server which I've been supporting since version 6.5 back in the 1990s. In this article, I'll be discussing how PowerShell can be used to retrieve just about any information that you would want to know about modern versions of SQL Server that you currently have running in your environment. This article isn't meant to be a deep dive, it's meant to get you started thinking about how you could write your own PowerShell code to retrieve the specific information that you're looking for from your SQL Servers.

In this scenario, you have two servers running the core installation (no-GUI) version of Windows Server 2012 R2. One has SQL Server 2008 R2 installed and the other one has SQL Server 2014 installed, each one has multiple instances of SQL Server installed. All of the examples shown are being performed on a Windows 8.1 Enterprise edition workstation with the SQL Server 2014 management tools installed and the RSAT (Remote Server Administration Tools) installed.

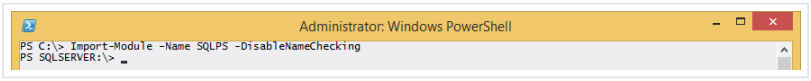
Note: SQL Server 2008 R2 and prior versions of SQL Server that supported PowerShell use a snap-in and beginning with SQL Server 2012 a module is used instead.

Working with SQL Server as if it were a file system

One way of working with SQL Server in PowerShell is through the SQLServer PSDrive that's created when the SQL PowerShell snap-in or module is imported. This allows you to work with SQL Server as if it were a file system.

Since the SQL Server 2014 client tools are installed on our workstation, there is a PowerShell module named *SQLPS* installed and you'll need to start out by importing that module:

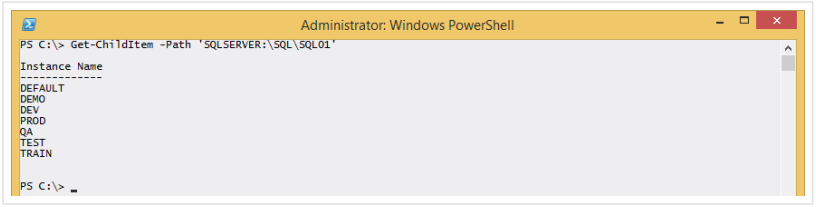
```
1 Import-Module -Name SQLPS -DisableNameChecking
```



Notice that the *–DisableNameChecking* parameter was specified in the previous example. There are a couple of cmdlets in the SQLPS module that use unapproved verbs (*Encode-SqlName* and *Decode-SqlName*) that will cause warnings to be generated if this optional parameter isn't specified. The warnings wouldn't hurt anything, but I prefer not to see them. You can also see in the previous example that when the SQLPS module is imported, it automatically changes your current location to the SQLSERVER PSDrive.

Determining the instances for the server named SQL01 is simple as shown in the following example:

```
1 Get-ChildItem -Path 'SQLSERVER:\SQL\SQL01'
```



POPULAR LATEST COMMENTS

Posh-SSH: Open Source SSH PowerShell Module
1

Find an unused drive letter
31

Accidental Sabotage: Beware of CredSSP
20

Downloads

Windows Management Framework – PowerShell 2.0 (All Platforms)

Windows Management Framework 3.0

Windows PowerShell 3.0 and Server Manager Quick Reference Guides

Windows Management Framework 4.0

Windows Management Framework 5.0 Preview

Windows PowerShell 4.0 and Other Quick Reference Guides

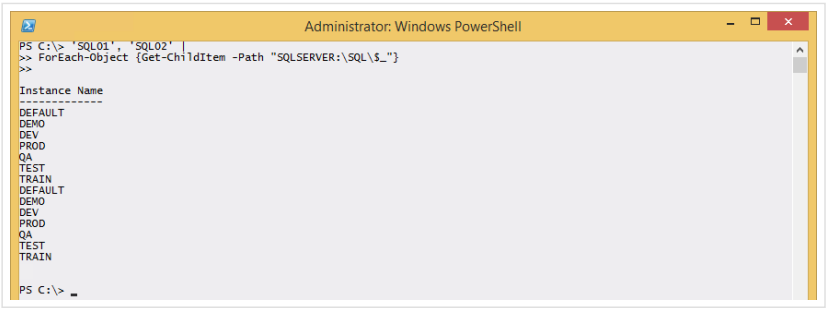
Post Tags

"PowerShell Security Special" Active Directory Azure Book Brainteaser conference DeepDive DevOps DSC eBook exchange getting started git Hyper-V infosec interview interviews ise linux Module NET news OMI Pester PowerShell PowerShell 2.0 PowerShell 3.0 PowerShell 4.0 PSConfAsia PSConfEU pscx PSDSC PSTip security SMO SQL TEC2011 TechEd Tips and Tricks Video videos Windows 8 WMI XML XPath

One of the things that makes PowerShell so powerful is that once you figure out how to perform a task for one item (one computer in this scenario), it's easy to perform that same task for multiple items.

The *ForEach-Object* cmdlet can be used to return the results for multiple SQL Servers:

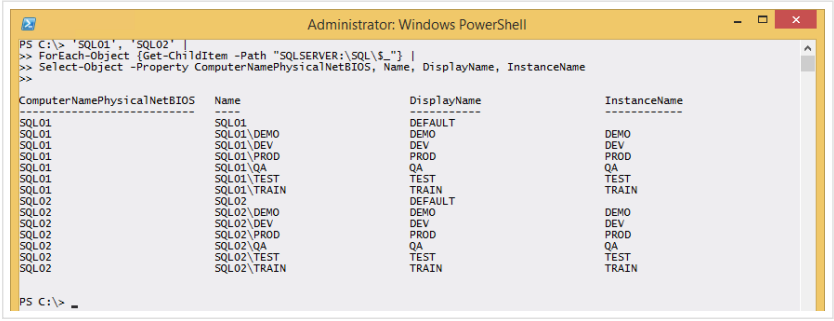
```
'SQL01', 'SQL02' |
ForEach-Object {Get-ChildItem -Path "SQLSERVER:\SQL\$_"}
```



By default, only the *InstanceName* property is returned, so you have no idea which instances belong to which servers.

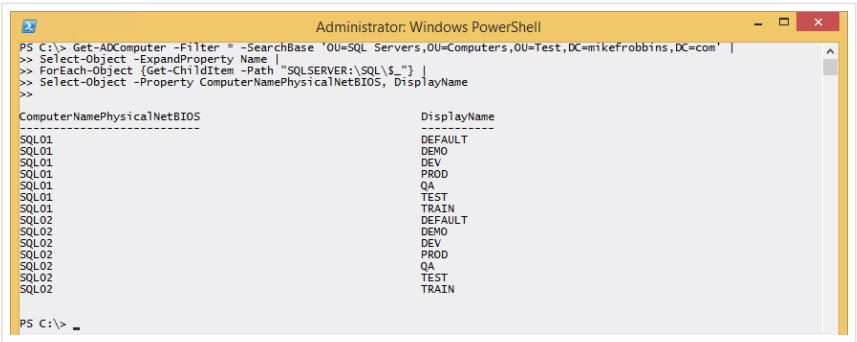
Just like any other cmdlet that produces output, you can pipe the previous command to *Get-Member* to see all of the available properties or *Format-List –Properties ** to see all of the available properties and their values. Be prepared to be overwhelmed though because there's a lot more information about SQL Server that can be obtained. For the sake of simplicity, I chose to focus on the Instance Name so here are a few helpful properties:

```
'SQL01', 'SQL02' |
ForEach-Object {Get-ChildItem -Path "SQLSERVER:\SQL\$_"} |
Select-Object -Property ComputerNamePhysicalNetBIOS, Name, DisplayName, InstanceName
```



You may have too many SQL Servers to manually type in the name for, but that's not a problem if they're stored in Active Directory where you can query the names from such as in their own Active Directory OU (Organizational Unit):

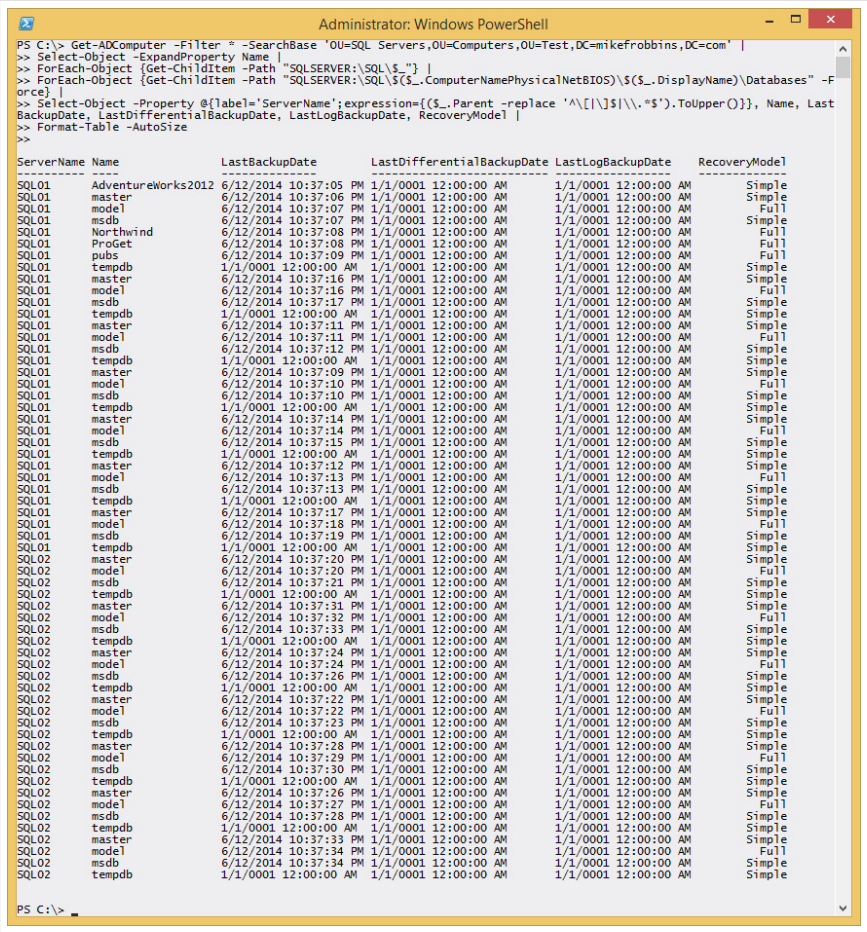
```
Get-ADComputer -Filter * -SearchBase 'OU=SQL Servers,OU=Computers,OU=Test,DC=mikefrc
Select-Object -ExpandProperty Name |
ForEach-Object {Get-ChildItem -Path "SQLSERVER:\SQL\$_"} |
Select-Object -Property ComputerNamePhysicalNetBIOS, DisplayName
```



You could also return information such as when the latest backups were taken and what the recovery model is for every database on every instance of every SQL Server all with a PowerShell one-liner:

```
Get-ADComputer -Filter * -SearchBase 'OU=SQL Servers,OU=Computers,OU=Test,DC=mikefrc
Select-Object -ExpandProperty Name |
ForEach-Object {Get-ChildItem -Path "SQLSERVER:\SQL\$_"} |
ForEach-Object {Get-ChildItem -Path "SQLSERVER:\SQL\$($_.ComputerNamePhysicalNetBIOS
```

```
Select-Object -Property @{label='ServerName';expression={($_.Parent -replace '^\[|\]
Format-Table -AutoSize
```



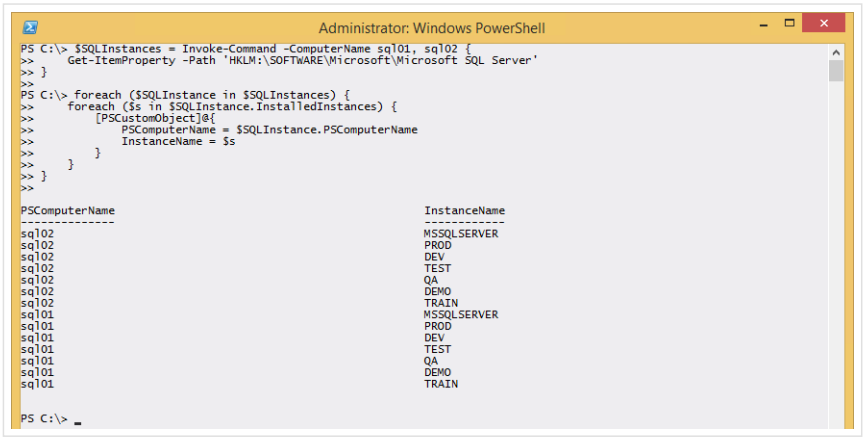
Although the command is on more than one physical line, it's still considered to be a PowerShell one-liner because it's one continuous pipeline.

All of the previous examples run against one server at a time and if a server isn't responding, you would have to wait for it to time out before the next one would begin which could be a slow process depending on how many SQL Servers are in your environment. Beginning with Windows Server 2012, PowerShell remoting is enabled by default so we could simply wrap the previous examples inside the *Invoke-Command* cmdlet which would run the commands against up to 32 servers in parallel by default. The number of servers that *Invoke-Command* runs against in parallel at a time is also configurable via the ThrottleLimit parameter. The only modification that would need to be made is the commands inside of the *Invoke-Command* script block would need to target the local computer since it would effectively be running locally on the remote SQL Servers and the results would be returned as deserialized objects.

Retrieving SQL Instance Names from the Registry

The name of each instance could also be obtained from the registry of the SQL Servers:

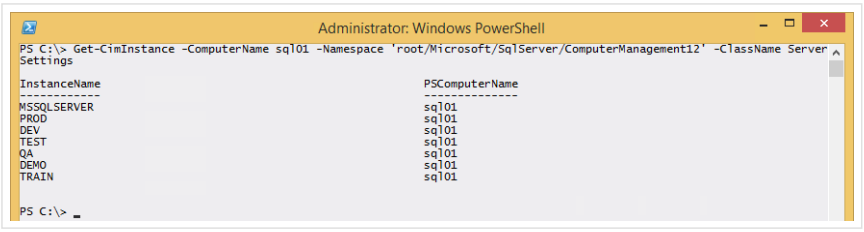
```
$SQLInstances = Invoke-Command -ComputerName sql01, sql02 {
Get-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Microsoft SQL Server'
}
foreach ($SQLInstance in $SQLInstances) {
    foreach ($s in $SQLInstance.InstalledInstances) {
       [PSCustomObject]@{
            PSComputerName = $SQLInstance.PSComputerName
            InstanceName = $s
       }
   }
}
```



Retrieving SQL Instance Names with WMI (Windows Management Instrumentation)

You could also obtain the SQL instance information with WMI. Be aware that different versions of SQL Server use different WMI namespaces.

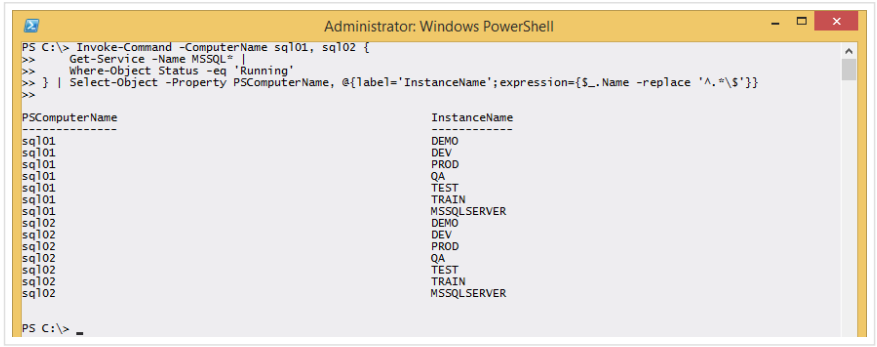
```
1 Get-CimInstance -ComputerName sql01 -Namespace 'root/Microsoft/SqlServer/ComputerMar
```



Retrieving SQL Instance Names from Services

If you've worked with PowerShell for more than 10 minutes, then you've probably been introduced to the *Get-Service* cmdlet. Believe it or not, the *Get-Service* cmdlet is one of the simplest ways to get a list of the instances on your SQL Servers since every instance that's installed creates its own service:

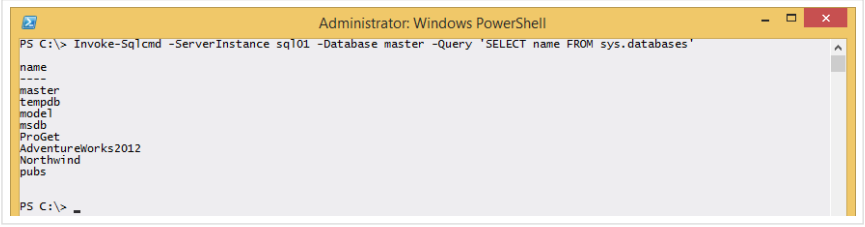
```
1 Invoke-Command -ComputerName sql01, sql02 {
2     Get-Service -Name MSSQL* |
3     Where-Object Status -eq 'Running'
4 } | Select-Object -Property PSComputerName, @{label='InstanceName';expression={$_.Na
```



Running existing T-SQL from PowerShell

If you have existing T-SQL statements, you can simply wrap them inside of the *Invoke-Sqlcmd* cmdlet:

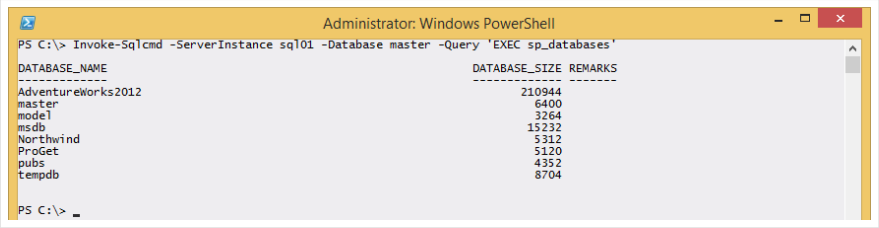
```
1 Invoke-Sqlcmd -ServerInstance sql01 -Database master -Query 'SELECT name FROM sys.da
```



Running Stored Procedures from PowerShell

You can also run stored procedures from PowerShell using the *Invoke-SQLCmd* cmdlet:

```
Invoke-Sqlcmd -ServerInstance sql01 -Database master -Query 'EXEC sp_databases'
```



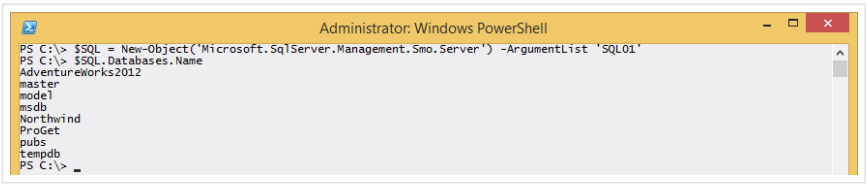
Working with SQL Server through the use of SMO (SQL Server Management Objects)

SMO in my opinion is a little more complicated, but it's also one of the more popular methods for accessing information about your SQL Servers with PowerShell.

You may have seen other articles where DLLs had to be imported in order to use SMO and that's still the case if you're running a version of the SQL Server management tools that uses a PowerShell snap-in, but the good news is that beginning with SQL Server 2012 where a PowerShell module is used, the necessary DLLs are automatically imported when the module is imported.

In this example, I'll use SMO to return a list of database names for the default instance of SQL Server on sql01:

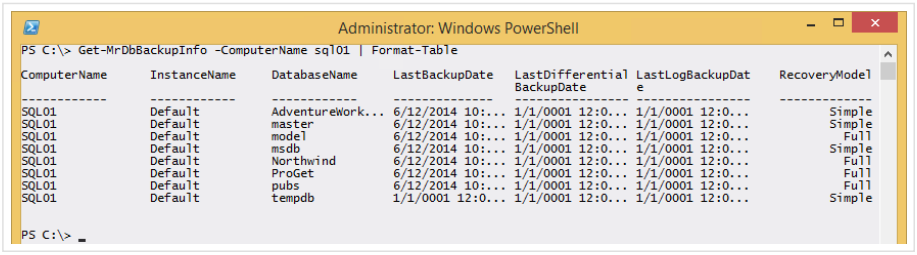
```
$SQL = New-Object('Microsoft.SqlServer.Management.Smo.Server') -ArgumentList 'SQL01'
$SQL.Databases.Name
```



You could create reusable PowerShell functions that leverage SMO to accomplish your common tasks such as this one that retrieves the backup information for one or more SQL Servers and instances:

```
#Requires -Version 3.0
function Get-MrDbBackupInfo {
<#
.SYNOPSIS
Returns database backup information for a Microsoft SQL Server database.

.DESCRIPTION
Get-DbBackupInfo is a function that returns database backup information for
one or more Microsoft SQL Server databases.

.PARAMETER ComputerName
The computer that is running Microsoft SQL Server that you're targeting to
query database backup information for.

.PARAMETER InstanceName
The instance name of SQL Server to return database backup information for.
The default is the default SQL Server instance.

.PARAMETER DatabaseName
The database(s) to return backup information for. The default is all databases.

.EXAMPLE
Get-DbBackupInfo -ComputerName sql01

.EXAMPLE
Get-DbBackupInfo -ComputerName sql01 -DatabaseName master, msdb, model

.EXAMPLE
Get-DbBackupInfo -ComputerName sql01 -InstanceName MrSQL -DatabaseName master,msdb

.EXAMPLE
'master', 'msdb', 'model' | Get-DbBackupInfo -ComputerName sql01

.INPUTS
String

.OUTPUTS
PSCustomObject
#>

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
        ValueFromPipelineByPropertyName)]
        [Alias('ServerName','PSComputerName')]
        [string[]]$ComputerName,
```

```
47
48          [Parameter(ValueFromPipelineByPropertyName)]
49          [ValidateNotNullOrEmpty()]
50          [string[]]$InstanceName = 'Default',
51
52          [Parameter(ValueFromPipelineByPropertyName)]
53          [ValidateNotNullOrEmpty()]
54          [string[]]$DatabaseName = '*'
55
56      )
57
58      BEGIN {
59         $problem = $false
60         Write-Verbose -Message "Attempting to load SQL Module if it's not already lo
61         if (-not (Get-Module -Name SQLPS)) {
62             try {
63                 Import-Module -Name SQLPS -DisableNameChecking -ErrorAction Stop
64             }
65             catch {
66                 $problem = $true
67                 Write-Warning -Message "An error has occurred.  Error details: $_.Ex
68             }
69         }
70      }
71
72      PROCESS {
73          foreach ($Computer in $ComputerName) {
74               foreach ($Instance in $InstanceName) {
75                   Write-Verbose -Message 'Checking for default or named SQL instance
76                   If (-not ($problem)) {
77                       If (($Instance -eq 'Default') -or ($Instance -eq 'MSSQLSERVER'
78                          $SQLInstance = $Computer
79                       }
80                       else {
81                          $SQLInstance = "$Computer\$Instance"
82                       }
83                       $SQL = New-Object('Microsoft.SqlServer.Management.Smo.Server')
84                   }
85
86                   if (-not $problem) {
87                        foreach ($db in $DatabaseName) {
88                            Write-Verbose -Message "Verifying a database named: $db e
89                            try {
90                                if ($db -match '\*') {
91                                    $databases = $SQL.Databases | Where-Object Name
92                                }
93                                else {
94                                    $databases = $SQL.Databases | Where-Object Name
95                                }
96                            }
97                            catch {
98                                $problem = $true
99                                Write-Warning -Message "An error has occurred.  Error
100                           }
101                           if (-not $problem) {
102                               foreach ($database in $databases) {
103                                   Write-Verbose -Message "Retrieving information f
104                                   [PSCustomObject]@{
105                                       ComputerName = $SQL.Information.ComputerName
106                                       InstanceName = $Instance
107                                       DatabaseName = $database.Name
108                                       LastBackupDate = $database.LastBackupDate
109                                       LastDifferentialBackupDate = $database.LastD
110                                       LastLogBackupDate = $database.LastLogBackupD
111                                       RecoveryModel = $database.RecoveryModel
112                               }
113                            }
114                       }
115                   }
116               }
117           }
118       }
119   }
120
121
122   Get-MrDbBackupInfo -ComputerName sql01 | Format-Table
```



Each time you find a task that you commonly need to perform for your SQL Servers, write a PowerShell function for it, combine those functions into a script module and you'll have your own custom SQL Server PowerShell Toolkit. Last but not least, share your toolkit with the PowerShell and SQL Server communities.

Share this:

Filed in: Articles, Online Only Tags: PowerShell, SQL

About Mike Robbins

Mike F Robbins is a Microsoft MVP on Windows PowerShell and a SAPIEN Technologies MVP. He is a co-author of Windows PowerShell TFM 4th Edition and is a contributing author of a chapter in the PowerShell Deep Dives book. Mike has written guest blog articles for the Hey, Scripting Guy! Blog, PowerShell Magazine, and PowerShell.org. He is the winner of the advanced category in the 2013 PowerShell Scripting Games. Mike is also the leader and co-founder of the Mississippi PowerShell User Group. He blogs at mikefrobbins.com and can be found on twitter @mikefrobbins.

View all posts by Mike Robbins →

Search

Search this site...

Categories

Categories

Select Category

Archives

Meta

Log in

Entries RSS

Comments RSS

Archives

Select Month

WordPress.org

Proudly designed by Theme Junkie.