# The PowerShell Conference Book


Edited by
Mike F. Robbins
Michael T. Lombardi
Jeff Hicks

THE DEVOPS COLLECTIVE
devopscollective.org

# The PowerShell Conference Book

Mike F Robbins, Michael T Lombardi and Jeff Hicks

This book is for sale at http://leanpub.com/powershell-conference-book

This version was published on 2018-09-08


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# Tweet This Book!

Please help Mike F Robbins, Michael T Lombardi and Jeff Hicks by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm supporting the future and just bought a copy of The #PowerShell Conference Book

The suggested hashtag for this book is #PSConfBook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#PSConfBook

# Also By These Authors

## Books by Mike F Robbins

PowerShell 101

## Books by Jeff Hicks

The PowerShell Scripting and Toolmaking Book

The PowerShell Practice Primer

# Contents

# Introduction

Windows PowerShell has been around for over a decade now and with each revision, more and more features have been added. PowerShell Core, which was introduced in 2017, allows you to run PowerShell not only on Windows, but also on Linux and macOS.

Learning the features that were added incrementally for those of us who started back in the PowerShell version 1.0 or 2.0 days was easy enough, but the learning curve has become steeper with each new feature for people who are just getting started in the industry.

I've attended the PowerShell + DevOps Global Summit every year since its inception and I can definitely speak from experience that it has helped my career, not only from the knowledge that I gained, but also from the connections I've made while networking with others at the conference.

In 2018, one person was awarded a scholarship to attend the PowerShell + DevOps Global Summit. After meeting the recipient, Andrew Pla[1], I couldn't believe how enthusiastic he was about learning PowerShell. His enthusiasm was contagious. A new OnRamp track and scholarship program was also announced to help bootstrap others into the PowerShell and DevOps community.

I decided to leverage the connections that I'd made and create a book where we could all pay it forward by donating a small portion of the time we'd saved over the years with PowerShell and give it back to the community. Be sure to read the *Contributors* section of this book which lists the authors because they're the ones who made this project a huge success. I would like to thank each of them for taking time away from their families to help others begin with a solid foundation of PowerShell knowledge by donating to the OnRamp scholarship program.

*Mike F. Robbins*[2], Creator of The PowerShell Conference Book

[1] https://twitter.com/AndrewPlaTech
[2] https://mikefrobbins.com/

# Contributors

## Emin Atac

Emin Atac is an IT and SecOps Pro who blogs about his Information Security (InfoSec) and Automation journeys on p0w3rsh3ll.wordpress.com[3] and shares his projects on his GitHub page[4]. Emin is a contributor and moderator at patchmanagement.org. Emin is a certified Digital Forensics and Incident Response (DFIR) professional and a Cloud and Datacenter Management (CDM) Microsoft Most Valuable Professional (MVP) award recipient.

## Graham Beer

Graham is an experienced Infrastructure engineer with a passion for PowerShell and automation. You can follow his blog at https://graham-beer.github.io/[5] or reach him on Twitter at @GKBeer[6].

## Brian Bunke

Brian is an IT Operations Lead who likes data, APIs, testing, and long lists. He enjoys helping others become more included, empowered, and efficient. He dislikes autobiographical blurbs, especially those written in third person.

brianbunke.com[7] | Twitter[8] | GitHub[9]

## Tim Curwick

Tim Curwick is an automation consultant, trainer, and speaker with a passion for PowerShell-based automation. He blogs as MadWithPowerShell[10], tweets @MadWPowerShell[11], and speaks at many venues, most frequently as a long-time leader of the Minnesota PowerShell Automation (User) Group[12].

[3] https://p0w3rsh3ll.wordpress.com
[4] https://github.com/p0w3rsh3ll
[5] https://graham-beer.github.io/
[6] https://twitter.com/GKBeer
[7] http://www.brianbunke.com
[8] https://twitter.com/brianbunke
[9] https://github.com/brianbunke
[10] https://MadWithPowerShell.com
[11] https://Twitter.com/MadWPowerShell
[12] https://MeetUp.com/Twin-Cities-PowerShell-User-Group

## Luc Dekens

Luc Dekens, co-author of the *PowerCLI Reference* books (Ed 1 and Ed 2), is a huge automation fan, and ultimately wants to achieve a 100% Software Defined Data Center (SDDC). He is a regular speaker at conferences worldwide. For his community contributions he has received multiple VMware vExpert and Microsoft MVP awards.

## Doug Finke

Doug Finke, author of the O'Reilly title *PowerShell for Developers* and Microsoft MVP since 2010, builder of advanced PowerShell tools, gluing things together, making humans faster and more agile. From release pipelines to DevOps in the Cloud, surfacing APIs around .NET DLLs like his popular PowerShell Excel module and his latest, building PowerShell Azure Functions to make them accessible as REST APIs. Catch up with Doug at his blog https://dfinke.github.io[13]

## Tore Groneng

Tore Groneng is a consultant working with Azure and PowerShell. From time to time I write and maintain my blog at https://asaconsultant.blogspot.no/[14]. You can follow me on Twitter @ToreGroneng[15].

## Patrick Gruenauer

Patrick Gruenauer was awarded the MVP on PowerShell for the first time in 2017. He is an IT-Trainer, IT-Systems Engineer and Cisco Instructor. His focus is on Windows PowerShell, Windows Server, Network Technologies and Automation. Follow Patrick on his site sid-500.com[16] or on Twitter @pewa2303[17].

## Jeff Hicks

Jeff Hicks is a long time PowerShell MVP, IT veteran and well known PowerShell teacher, author and speaker. His latest book from Leanpub is The PowerShell Practice Primer[18]. Follow Jeff on Twitter at @JeffHicks[19] or his blog[20].

[13] https://dfinke.github.io
[14] https://asaconsultant.blogspot.no/
[15] https://twitter.com/ToreGroneng
[16] https://sid-500.com/
[17] https://twitter.com/pewa2303
[18] https://leanpub.com/psprimer
[19] https://twitter.com/JeffHicks
[20] https://jdhitsolutions.com/blog

## Don Jones®

Don Jones has worked with PowerShell since before its 2006 release, and is a well-known PowerShell author, speaker, and blogger. He co-authored "Learn Windows PowerShell in a Month of Lunches" and many other PowerShell books, co-founded PowerShell.org, and founded PowerShell + DevOps Global Summit. Reach Don online at DonJones.com[21].

## Mike Kanakos

Mike Kanakos is a Windows IT Pro located in the RTP area of North Carolina. He specializes in Active Directory, Azure AD, Group Policy, and automation via PowerShell. You can follow Mike's blog at www.networkadm.in[22] or on Twitter at @MikeKanakos[23].

## Wesley Kirkland

Wesley Kirkland is a Sr. Systems Engineer with a focus on Automation and Microsoft technologies. He is a lifelong techie and enjoys copious amount of Dr. Pepper. He can be found at wesleyk.me[24], on Twitter @UnleashTheCloud[25], or on reddit /u/creamersrealm[26]

## Mark Kraus

Mark Kraus (@markekraus[27]) is a Microsoft MVP with over 2 decades of IT experience in companies of all shapes and sizes. Mark is a PowerShell Core Open Source Collaborator and primary contributor to the Web Cmdlets. Mark is also the author of the Get-PowerShellBlog[28] PowerShell blog.

## Thomas Lee

Thomas Lee is a consultant/trainer/writer from England and has been in the IT business since the late 1960's. Since 1988, he has run his own consulting and training business, with a couple of spells as an employee. Thomas holds numerous Microsoft certifications, including the original MCSE (he was one of the first in the world), has been an MCT for 22 years, and has been awarded Microsoft's

[21] https://donjones.com/
[22] https://www.networkadm.in/
[23] https://twitter.com/MikeKanakos
[24] https://wesleyk.me
[25] https://twitter.com/UnleashTheCloud
[26] https://reddit.com/u/creamersrealm
[27] https://twitter.com/markekraus
[28] https://get-powershellblog.blogspot.com/

MVP award 17 times. He is also a Fellow (retired) of the British Computer Society and Chartered IT Professional.

Thomas is semi-retired but is always happy to teach, talk about, and solve problems related to PowerShell. He has time to work with his collection of Grateful Dead and Jerry Garcia live recordings and a bit of gardening.

# Michael T. Lombardi

Mike is a sysadmin-turned-automation-engineer-turned-software-engineer at Puppet[29], passionate about documentation and restorative justice. He is the founder and co-organizer of the St. Louis PowerShell User Group[30] and host of the ChatterOps[31] podcast.

He also makes time to mentor and teach folks about agile approaches to infrastructure, infrastructure as code, source control, systems thinking, and general engineering. He has open office hours listed here[32].

Twitter[33] | GitHub[34]

# Tommy Maynard

Tommy Maynard is a Senior Systems Administrator with a focus on Microsoft technologies, Amazon Web Services, and all things automation. Tommy is a dedicated PowerShell community author. He's published at tommymaynard.com[35] since June 2014 at a rate of nearly six posts per month. He can be found on Twitter using @thetommymaynard[36].

# Jeremy Murrah

Jeremy Murrah is an old OPS guy from the dark ages of computing. He started automating NT 4.0 installations and hasn't looked back. Classically trained as an Active Directory Administrator and Windows Engineer, he is currently engrossed in all things PowerShell and is eagerly awaiting the death of the GUI. He blogs at murrahjm.github.io[37] and can be found on twitter @JeremyMurrah[38].

[29] https://puppet.com
[30] https://meetup.com/stlpsug
[31] https://chatterops.org
[32] https://appoint.ly/t/michaeltlombardi
[33] https://twitter.com/barbariankb
[34] https://github.com/michaeltlombardi
[35] https://tommymaynard.com
[36] https://twitter.com/thetommymaynard
[37] https://murrahjm.github.io
[38] https://twitter.com/JeremyMurrah

## Adam Murray

Adam Murray is a doer, an optimist, and a technologist. With 20 years of experience in corporate IT and one startup behind him, he's recently embarked on my second startup, Tikabu, which provides premium consulting services in secure DevOps. Focusing on clients with a predominantly Microsoft technology stack, he helps his customers become more agile at delivering solutions to their customers.

You can follow Adam on twitter at @muzzar78[39], on GitHub[40] or read an occasional blog at Tikabu[41].

## Anthony E. Nocentino

Meet Anthony Nocentino, Enterprise Architect, Founder and President of Centino Systems, Microsoft Data Platform MVP, Pluralsight Author, Corporate Problem Solver and a voracious student of the latest computer science technology. Anthony is only satisfied when he finds the right technology resolution for his client's business need. Business today thrives on data – from the C-Suite to the information worker. Anthony specializes in all things related to data – database systems, virtualization, system and network design and performance engineering. He designs solutions, deploys the technology and provides expertise on business system performance, architecture and security.

## Brandon Olin

Brandon is a Cloud Architect, veteran Systems Engineer, speaker, blogger, freelance writer, and open source contributor. He has a penchant for PowerShell and DevOps processes. You can follow his code at GitHub[42], his blog at devblackops.io[43], or reach him on Twitter at @devblackops[44].

## James Petty

James has been in the IT industry since graduating college (actually before that, but only part time). He currently lives in Southeast Tennessee and works as a Windows Server Administrator. When he is not at work, you can find him volunteering with the Boy Scouts and working with the DevOps Collective. James can be found on Twitter @PSJamesP[45] and blogs at https://www.scriptautomaterepeat.com[46]

[39] https://twitter.com/muzzar78
[40] https://github.com/muzzar78
[41] https://tikabu.com.au/blog/
[42] https://github.com/devblackops
[43] https://devblackops.io
[44] https://twitter.com/devblackops
[45] https://twitter.com/PSJamesP
[46] https://www.scriptautomaterepeat.com

James is also the Treasurer for the DevOps Collective Inc / PowerShell.org and helps run the PowerShell + DevOps Global Summit.

## Rob Pleau

Rob is an automation engineer and PowerShell nerd. He is a PowerShell Summit speaker and PowerShell blogger at https://ephos.github.io.

You can follow Rob on twitter at @rjpleau[47] or on GitHub[48].

## Thomas Rayner

Thomas Rayner is a Senior Security Service Engineer at Microsoft with many years of experience in IT. He is a master technologist, specializing in DevOps, systems and process automation, public, private and hybrid cloud, security and PowerShell. Thomas is a former 4x Microsoft MVP, Honorary Scripting Guy, an international speaker, best-selling author, and instructor covering a vast array of IT topics.

Follow Thomas on Twitter at @MrThomasRayner[49], and read more from him on his blog, workingsysadmin.com[50].

## Mike F. Robbins

Mike F. Robbins is a Microsoft Cloud and Datacenter Management MVP for Windows PowerShell. He is the creator of The PowerShell Conference Book, author of PowerShell 101: The No-Nonsense Beginner's Guide to PowerShell, co-author of Windows PowerShell TFM 4th Edition, and a contributing author of a chapter in the PowerShell Deep Dives book. Mike is also the leader and co-founder of the Mississippi PowerShell User Group. He blogs at mikefrobbins.com[51] and can be found on twitter @mikefrobbins[52].

## Thom Schumacher

Thom is a Microsoft PowerShell Enthusiast and User group leader based in Chandler, Arizona.

You can follow Thom on Twitter @driberif[53] or on GitHub[54]

[47] https://twitter.com/rjpleau
[48] https://github.com/ephos
[49] http://twitter.com/mrthomasrayner
[50] https://workingsysadmin.com
[51] https://mikefrobbins.com/
[52] https://twitter.com/mikefrobbins
[53] https://twitter.com/driberif
[54] https://github.com/crshnbrn66

## Rob Sewell

Rob is a SQL Server DBA with a passion for Powershell, Azure, Automation, & SQL (PaaS geddit?). He is an MVP & an officer for the PASS PowerShell VG & has spoken at & volunteered at many events. He is a member of the committee that organises SQL Saturday Exeter & also the European PowerShell Conference. He is a proud supporter of the SQL & PowerShell communities. He works as a consultant, generally with Data Platform solutions, providing DevOps, automation and training.

He relishes sharing & learning & can be found doing both via social media. He spends most of his time looking at a screen & loves to solve problems. He knows that looking at a screen so much is bad for him because his wife tells him so. Thus, you can find him on the cricket field in the summer or flying a drone in the winter.

## Mike Shepard

Mike is a Solutions Architect doing build and install development and has previously been a SysAdmin, a DBA, a developer, and in Operations. PowerShell is Mike's second superpower, after SQL. He has been using PowerShell since 2008, blogs at https://PowerShellStation.com[55] and has written two PowerShell books. He is the founder of the Southwest Missouri PowerShell User Group[56].

## Justin Sider

Justin Sider, Chief Information Officer for Belay Technologies, leads the development and implementation of a tool for his current company utilizing VMware for an automated provisioning & testing solution. Mr. Sider has 15+ years of experience as owner of his own tech business and in lead roles for various tech companies. Mr. Sider has 10+ years of experience working with VMware products and programming with PowerShell. His work can be found on GitHub and the PowerShell Gallery under the username jpsider.

You can follow Justin on twitter at @jpsider[57], at the PowerShell Gallery[58],on GitHub[59] or read an occasional blog at Invoke-Automation[60].

## Prateek Singh

Prateek Singh is a Infrastructure developer, automation engineer and technical writer, you can read his articles on his blog RidiCurious.com[61] where he fiddles with code and technologies, specially

[55] https://PowerShellStation.com
[56] https://www.meetup.com/SWMO-PowerShell-User-Group
[57] https://twitter.com/jpsider
[58] https://www.powershellgallery.com/profiles/jpsider/
[59] https://github.com/jpsider
[60] https://invoke-automation.blog/
[61] http://ridicurious.com

with PowerShell and Azure cloud. Prateek also runs a YouTube channel[62] where you can find him talking about PowerShell scripting, Azure services and Python.

All his PowerShell projects are open-sourced at GitHub[63] and you can follow Prateek on Twitter @SinghPrateik[64] to see what he is working on right now.

## Irwin Strachan

Irwin Strachan is a senior consultant at OLGreen B.V. specialized in PowerShell and Microsoft-based technologies. You can reach Irwin at his blog[65] or his Twitter at @IrwinStrachan[66].

## Tim Warner

Tim Warner is a Microsoft MVP in Cloud & Datacenter Management based in Nashville, Tennessee. Follow Tim on Twitter at @TechTrainerTim[67].

## Friedrich "Fred" Weinmann

Fred is Systems Engineer, PowerShell consultant, speaker, blogger, PowerShell enthusiast, usergroup organizer, open source contributor and writer. His enthusiasm for automation in general and PowerShell in specific often lead him down the rabbit-hole in search for yet greater convenience.

You can trace his path by reading his blog[68], checking his coding work at his PowerShell Framework Project[69] or by his occasional tweets as @FredWeinmann[70].

## Mark Wragg

Mark is a DevOps Engineer who is passionate about PowerShell, automation and all things DevOps. You can follow his blog, https://wragg.io/, or reach him on Twitter at @markwragg[71]. He can also often be found answering questions on the PowerShell[72] and Pester[73] tags of StackOverflow.com.

[62] https://www.youtube.com/channel/UCjzFiRgBGLSKQ2Uu2wUds-g
[63] https://github.com/PrateekKumarSingh
[64] https://twitter.com/SinghPrateik
[65] https://pshirwin.wordpress.com
[66] https://twitter.com/irwinstrachan
[67] https://twitter.com/TechTrainerTim
[68] https://allthingspowershell.blogspot.com
[69] https://psframework.org
[70] https://twitter.com/fredweinmann
[71] https://twitter.com/markwragg
[72] https://stackoverflow.com/questions/tagged/powershell
[73] https://stackoverflow.com/questions/tagged/pester

# How to Use This Book

Imagine attending a PowerShell conference where over thirty speakers who are subject matter experts in the industry are each presenting one forty-five minute session. All of the sessions are at different times so there's no need to worry about choosing between them. You might be wondering how much a conference like this will cost by the time you pay for the conference, hotel, airfare, and meals? Well, there's no need to worry because this conference doesn't cost a fortune because it's actually a book that's designed to be like a conference.

This book is designed to be like a conference in a book where each chapter is written by a different author who is a subject matter expert on the topic covered in their chapter. Each chapter is also independent of the others so you can read one chapter, ten chapters, or all of them. You can start with the first chapter, the last one, or somewhere in-between and not miss out on anything related to that particular topic.

## About OnRamp

OnRamp is an annual entry-level education program focused on PowerShell and DevOps. It's a conference within a conference at the PowerShell + DevOps Global Summit. The OnRamp track requires a distinct ticket type which is specifically designed for entry-level technology professionals who have completed foundational certifications such as CompTIA A+ and Cisco IT Essentials. No prior PowerShell experience is required, although some basic knowledge of server administration is useful. You'll network with other Summit attendees who are attending the expert level sessions during keynotes, meals, and evening events.

Through fundraising and corporate sponsorships, The DevOps Collective, Inc.[74] will be offering a number of full-ride scholarships to the OnRamp track at the PowerShell + DevOps Global Summit.

***All (100%) of the royalties from this book are donated to the OnRamp scholarship program.***

More information about the OnRamp track[75] at the PowerShell + DevOps Global Summit and their scholarship program[76] can be found on the PowerShell.org[77] website.

See the DevOps Collective Scholarships cause[78] on Leanpub.com[79] for more books that support the OnRamp scholarship program.

[74] https://devopscollective.org/
[75] https://powershell.org/summit/summit-onramp/
[76] https://powershell.org/summit/summit-onramp/onramp-scholarship/
[77] https://powershell.org/
[78] https://leanpub.com/causes/devopscollective
[79] https://leanpub.com/

# Prerequisites

Prior experience with PowerShell is highly recommended. This book is written for the intermediate to advanced audience and each chapter assumes that you've completed the OnRamp track at the PowerShell + DevOps Global Summit or have equivalent experience with PowerShell.

# A Note on Code Listings

If you've read other PowerShell books from LeanPub, you probably have seen some variation on this code sample disclaimer. The code formatting in this book only allows for about 75 characters per line before things start automatically wrapping. We've tried to keep the code samples within that limit, although sometimes you may see some awkward formatting as a result.

For example:

```
Get-CimInstance -ComputerName $computer -Classname Win32_logicalDisk -Filter "drivetype=3" -pr\
operty DeviceID,Size,FreeSpace
```

Here, you can see the default action for a too-long line - it gets word-wrapped, and a backslash inserted at the wrap point to let you know. We try to avoid those situations, but they may sometimes be unavoidable. When we *do* avoid them in this book, it may be with awkward formatting, such as using backticks (`)

```
Get-CimInstance -ComputerName $computer `
               -Classname Win32_logicalDisk `
               -Filter "drivetype=3" `
               -property 'DeviceID','Size','FreeSpace'
```

This code is formatted purely for reading purposes, you would never write code in this manner.

If you are reading this book on a Kindle, tablet or other e-reader, then all code formatting bets are off the table. There's no telling what the formatting will look like due to how each reader might format the page.

When *you* write PowerShell expressions, you should not be limited by these constraints. And all of our downloaded code samples do not have these limitations.

# Feedback

Have a question, comment, or feedback about this book? Please share it via the Leanpub forum dedicated to this book. Once you've purchased this book, login to Leanpub, and click on the "Join the Forum" link in the Feedback section of this book's webpage[80].

[80] https://leanpub.com/powershell-conference-book

# Acknowledgements

## About the Cover Image

We would like to thank Will Anderson[81] for providing us with the cover image which is a photograph of David Wilson[82] presenting at the PowerShell + DevOps Global Summit[83]. We would also like to thank David Wilson for allowing us to use this photo of him as the cover image of this book.

## Trademarks



[81] https://twitter.com/GamerLivingWill
[82] https://twitter.com/daviwil
[83] https://powershell.org/summit/

# Disclaimer

All code examples shown in this book have been tested by each individual chapter author and every effort has been made to ensure that they are error free, but since every environment is different, they should not be run in a production environment without thoroughly testing them first. It is recommended that you use a non-production or lab environment to thoroughly test code examples used throughout this book.

All data and information provided in this book is for educational purposes only. The editors make no representations as to accuracy, completeness, currentness, suitability, or validity of any information in this book and will not be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its display or use. All information is provided on an as-is basis.

This disclaimer is provided simply because someone, somewhere will ignore this disclaimer and in the event that they do experience problems or a “resume generating event”, they have no one to blame but themselves. Don’t be that person!

# Part 1 - PowerShell Scripting

Without a doubt many people come to PowerShell with scripting in mind. And why not? PowerShell has an easy to learn scripting language and once you master running PowerShell from a console prompt, creating reusable scripts and tools is the next logical step. Any PowerShell conference worth your time will have plenty of scripting related content and this book is no different.

# Writing Secure and Sterile Code


## by Jeff Hicks


I've been teaching and writing about PowerShell since its very beginning and in addition to my own scripting, I've seen the efforts of countless students and conference attendees. People *want* to write good PowerShell code but often either don't have the necessary skill set yet or fall into bad habits, primarily because they are easy. But at this point in PowerShell's evolution, creating good PowerShell also means creating secure and safe PowerShell.

## Don't Be Foolish

You would think that after more than 5 versions of PowerShell that IT Pros (or even developer-types) would know better than to include hard-coded credentials into their scripts and modules. Yet, people still do this. If you want to create secure code, you **absolutely** cannot include credentials and especially plain text passwords. If you are doing this today – stop. Really. Stop. It. Now.

If you are using Visual Studio Code, the integrated PowerShell Script Analyzer should detect these types of violations, including passing passwords as clear text. If you need to support credentials, then pass it as a parameter.

```
Parameter(
    [Parameter(Position = 0, Mandatory)]
    [string]$Computername,
    [PSCredential]$Credential
)
```

Don't pass a username or password. Use a PSCredential object such as what you get when calling `Get-Credential`. What I hope I'd never see in a script is code like this:

```
$p = "S3c>+P@ssw0rd"
$pass = ConvertTo-SecureString -String $p -AsPlainText -force
$cred = New-Object PSCredential administrator,$pass
```

I often see requests for help in obfuscating or encrypting source code of a PowerShell script. I'd wager that the reason most people want to do this is because they want to hide credentials. But I've already stressed that this is a major flaw. In fact, there should be *nothing* in your script that is sensitive, confidential, or would compromise security. If your code is clean and sterile, there is no need to hide it.

# The Typical Problem

Let's take a look at a sample PowerShell function that retrieves information from Active Directory for accounts in a given organizational unit. The function itself is not that compelling but look at the highlighted points.

```
#requires -module ActiveDirectory

# Get-DomainUser.ps1

Function Get-DomainUser {
    [cmdletbinding()]
    Param(
        [Parameter(ValueFromPipeline)]
        [ValidateNotNullOrEmpty()]
        [string]$OU = "OU=Employees,DC=Company,DC=pri",  #hard coded maybe OK
        [string]$Department
    )
    Begin {
        Write-Verbose "[$((Get-Date).TimeofDay) BEGIN  ] Starting `
        $($myinvocation.mycommand)"

        $domain = "company.pri"    # <--- hard coded values
        $dc = "dom1"               # <--- hard coded values

        $properties = "Name","SamAccountName","UserPrincipalName",
        "Description","Enabled"

        if ($Department) {
            $filter = "Department -eq '$Department'"
            $properties += "Title","Department"
        }
        else {
            $Filter = "*"
        }
        $paramhash = @{
            SearchBase = ""
            Server = "$dc.$domain"
            Filter = $filter
        }
    } #begin

    Process {
        Write-Verbose "[$((Get-Date).TimeofDay) PROCESS] Getting user accounts`
         from $OU"
        $paramhash.SearchBase = $OU
```

```
        Write-Verbose "[$((Get-Date).TimeofDay) PROCESS] Connecting to domain `
        controller $($dc.toupper())"

        Get-ADUser @paramhash
    } #process

    End {
        Write-Verbose "[$((Get-Date).TimeofDay) END    ] Ending `
        $($myinvocation.mycommand)"

    } #end

} #close Get-DomainUser
```

This function relies on hard-code values that identify your domain and domain controllers. More than once I've had people tell me they can't share their code because it contains sensitive or identifiable information. Why? If you would have to sanitize your code before showing it to me, you are doing something wrong. *You need to start thinking about separating the data from the PowerShell commands that use it.*

Your PowerShell script or function should be generic, sterile and completely agnostic. With the exceptions of in-house or vendor specific applications, your code should be general enough that anyone could view or execute it without compromising your security.

## Leverage Parameters

One of the best options to achieve secure and sterile code is to pass the necessary data as parameter values. Here's a revised version of my domain user function.

```
#requires -module ActiveDirectory

# Get-DomainUser2.ps1

# Instead of relying on hard coded values, set as parameter values

Function Get-DomainUser {
    [cmdletbinding()]
    Param(
        [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
        [ValidateNotNullOrEmpty()]
        [string]$OU,
        [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
        [string]$Domain,
        [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
```

```
        [ValidateNotNullOrEmpty()]
        [string]$DC,
        [Parameter(ValueFromPipelineByPropertyName)]
        [string]$Department
    )
    Begin {
        Write-Verbose "[$((Get-Date).TimeofDay) BEGIN  ] Starting `
$($myinvocation.mycommand)"
        $properties = "Name", "SamAccountName", "UserPrincipalName",
        "Description", "Enabled"
        if ($Department) {
            $filter = "Department -eq '$Department'"
            $properties += "Title", "Department"
        }
        else {
            $Filter = "*"
        }

    } #begin

    Process {
        #moved to process block
        $paramhash = @{
            SearchBase = ""
            Server     = "$($dc).$($domain)"
            Filter     = $filter
        }
        Write-Verbose "[$((Get-Date).TimeofDay) PROCESS] Getting user `
        accounts from $OU"
        $paramhash.SearchBase = $OU

        Write-Verbose "[$((Get-Date).TimeofDay) PROCESS] Connecting to `
        domain controller $($dc.toupper())"
        Get-ADUser @paramhash
    } #process

    End {
        Write-Verbose "[$((Get-Date).TimeofDay) END    ] Ending $($myinvocation.mycommand)"

    } #end

} #close Get-DomainUser
```

In this version, the previously hard coded values are now passed as parameter values. Notice that I've made some of the parameters mandatory. You certainly have an option to set a default parameter value. Realize that is a trade-off between ease of use and security. But the default value may not be that critical, so passing it as a default value is might be OK. You have to make that call.

Notice that I also configured a number of parameters to accept value from the pipeline by property value. With this approach I can pass any object to the function.

```
$data = @"
"OU","Domain","DC"
"OU=Employees,DC=company,DC=pri","company.pri","dom1"
"@

$data | ConvertFrom-Csv | get-domainuser -Verbose -Department sales
```

In this example, I have some data stored in a CSV format. This could easily have been an actual CSV file. The key point is that I am converting the CSV data to an object with properties that correspond to the function parameters. The function consumes incoming objects, binds the parameter values and runs the code.

With some planning you can store commonly used data sets into any number for formats. It really doesn't matter if you need to use json, xml or csv. If you can turn it into a PowerShell object you can pipe it to your function. With this model, you focus on securing and protecting the data files. You could incorporate the use of CMS Messages (see `Protect-CmsMessage`.) If you really, really needed to persist a credential to disk you might use my PSJsonCredential[84] module which you can install from the PowerShell Gallery.

Depending on who might be using your PowerShell tool or function, you might need to create a wrapper script that imports the necessary data.

```
[cmdletbinding()]
Param(
[Parameter(Mandatory,HelpMessage="Enter a department")]
[String]$Department
)
Import-CSV -path c:\data\helpdesk.csv | Get-DomainUser -department $department
```

Clearly, this script sample is merely a proof of concept.

# Use Configuration Data

To take this idea of separating data from code to the extreme, but in a good way, you might consider the use of PowerShell configuration data files. If you have worked with Desired State Configuration (DSC) you have mostly likely used data configuration in the form of psd1 files. You can use that same approach in traditional PowerShell scripting. Here's a version of my domain user function that utilizes configuration data.

[84] https://github.com/jdhitsolutions/PSJsonCredential

```
#requires -module ActiveDirectory

# Get-DomainUser3.ps1

#Using configuration data

Function Get-DomainUser {
    [cmdletbinding()]
    Param(
        [Parameter(Mandatory)]
        [ValidateScript({Test-Path $_})]
        [string]$ConfigurationData,
        [string]$Department
    )
    Begin {
        Write-Verbose "[$((Get-Date).TimeofDay) BEGIN  ] Starting `
        $($myinvocation.mycommand)"

        #import the configuration data into the function
        $config = Import-PowerShellDataFile -Path $ConfigurationData

        $properties = "Name","SamAccountName","UserPrincipalName",
        "Description","Enabled"
        if ($Department) {
            $filter = "Department -eq '$Department'"
            $properties += "Title","Department"
        }
        else {
            $Filter = "*"
        }

        #use the configuration data values
        $paramhash = @{
            SearchBase = $config.ou
            Server = "$($config.dc).$($config.domain)"
            Filter = $filter
        }
    } #begin

    Process {
        Write-Verbose "[$((Get-Date).TimeofDay) PROCESS] Getting user `
        accounts from $($config.ou)"
        Write-Verbose "[$((Get-Date).TimeofDay) PROCESS] Connecting to `
        domain controller $(($config.dc).toupper())"

        Get-ADUser @paramhash
```

```
    } #process

    End {
        Write-Verbose "[$((Get-Date).TimeofDay) END    ] Ending `
        $($myinvocation.mycommand)"

    } #end

} #close Get-DomainUser
```

This version of the function assumes the user will specify the path to a .psd1 file, which is why my parameter is typed as a string. Here's a sample file, `domain.configdata.psd1`.

```
@{
    OU = "OU=Employees,DC=Company,DC=pri"
    Domain = "company.pri"
    DC = "dom1"
}
```

In my function, I can import this data and turn it into the expected hashtable format:

```
$config = Import-PowerShellDataFile -Path $ConfigurationData
```

Now, I can use the configuration data throughout my function. Depending on how configuration data might be generated or passed to the function, I could have written the parameter like this:

```
 Param(
        [Parameter(Mandatory)]
        [hashtable]$ConfigurationData,
        [string]$Department
    )
```

This avoids the need to persist data to a psd1 file if I have other ways of getting the necessary configuration data.

# Summary

I hope you recognize that the easiest way to write secure PowerShell code is to separate the data from the functionality that uses it. Once you get your head around PowerShell's object-centric nature, you'll discover this isn't that difficult. True, you may need to re-architect or at least re-think your script or function. You will also need to consider how you will secure the data, but I'm trusting that as an IT Pro you have some experience with that task.

The next time some one asks to look at your code, your answer should be, "Why of course. Let me show you how it works!"

# Finding Performance Bottlenecks with PowerShell

by Mike F. Robbins

## Lab Environment

A single workstation running Windows 10 version 1803 is used throughout this chapter. It's running Windows PowerShell version 5.1 which ships in the box with that operating system. PowerShell must be run elevated as an administrator and the execution policy must be set to remote signed or less restrictive for some of the examples in this chapter to complete successfully. I recommend following along on your computer and walking through the examples while reading this chapter to see the results for yourself.

### Windows PowerShell, not PowerShell Core

Some of the cmdlets used in this chapter do not exist in PowerShell Core. Contrary to popular belief, PowerShell Core version 6.0 is not an upgrade or replacement to Windows PowerShell version 5.1. It installs side by side on Windows systems.

## Introduction

You've had a great weekend after leaving work with everything operating normally on Friday only to discover chaos as you walk into the office on Monday morning. It seems as if everyone who arrived at the office prior to you is standing around waiting on their profile to finish loading so they can begin working. The first few people who arrived at the office this morning were able to log in successfully and their systems seem to be operating normally.

Through the logical progression of trying to figure out why these problems are occurring, you start thinking of recent changes that were made to the infrastructure. Maybe it's related to that extended power outage that occurred a couple of weeks ago when all of the systems went down and didn't come back up in the proper sequence. Maybe it's something related to those new domain controllers

that were recently added to the network. After all, both the old and new ones are now online. The users who are experiencing problems could be authenticating to the new domain controllers and the users who were able to login could be authenticating to the old ones. It's been at least several days since each of these changes were made so why would it have taken this long for the problems to occur if one of them were the cause? Maybe it's related to some undocumented change that you're unaware of? Maybe it's not related to a change at all? We all battle these types of problems at one time or another and they can be very stressful for everyone involved.

Creating a baseline of the performance for the systems in your environment can be extremely useful when these types of problems occur, but it's rarely something that's done in information technology (IT) because no one cares about infrastructure until there's a problem.

# Performance Counters

Even if you don't have a baseline, performance counters can still be beneficial for use when trying to determine where performance related problems reside. While you do need something to compare the current results to, generic industry standard recommendations can be found on the Internet.

The `Get-Counter` PowerShell cmdlet is used to query performance counters on Windows systems. One of the reasons I chose to write about this topic is because the `Get-Counter` cmdlet is not very intuitive and the results for it aren't what I would call a great object-oriented design.

## Finding Performance Counter Sets

First, if you don't already know what performance counters you want to query, you'll need to find them somehow. While you could search the Internet, finding performance counters with PowerShell itself is easy enough. A good place to start would be to read the help for the Get-Counter cmdlet[85].

As with any other commands in PowerShell that produce object-based output, a list of the properties for `Get-Counter` can be determined by piping it to `Get-Member`. It's a good idea to start here no matter how much you know about PowerShell because what you think are the property names as shown in the output of a command aren't always the actual property names.

[85] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.diagnostics/get-counter

```
PS C:\> Get-Counter -ListSet * |
        Get-Member -MemberType Properties

   TypeName: Microsoft.PowerShell.Commands.GetCounter.CounterSet

Name               MemberType    Definition
----               ----------    ----------
Counter            AliasProperty Counter = Paths
CounterSetName     Property      string CounterSetName {get;}
CounterSetType     Property      System.Diagnostics.PerformanceCounterCatego...
Description        Property      string Description {get;}
MachineName        Property      string MachineName {get;}
Paths              Property      System.Collections.Specialized.StringCollec...
PathsWithInstances Property      System.Collections.Specialized.StringCollec...
```

If you select all of the properties for the first result, you'll get an idea of what type of values are returned for each property. `Get-Counter` just happens to return all of its properties by default regardless of whether or not you select all of them from the pipeline.

`CounterSetName` is one of the properties returned when using the `ListSet` parameter of `Get-Counter` as shown in the following example.

```
PS C:\> Get-Counter -ListSet * |
        Select-Object -First 1 -Property *

Counter            : {\Hyper-V VM Virtual Device Pipe IO(*)\Receive Message
                     Quota Exceeded, \Hyper-V VM Virtual Device Pipe
                     IO(*)\Receive QoS - Total Message Delay Time (100ns),
                     \Hyper-V VM Virtual Device Pipe IO(*)\Receive QoS -
                     Exempt Messages/sec, \Hyper-V VM Virtual Device Pipe
                     IO(*)\Receive QoS - Non-Conformant Messages/sec...}
CounterSetName     : Hyper-V VM Virtual Device Pipe IO
MachineName        : .
CounterSetType     : SingleInstance
Description        : Worker process per-pipe statistics, for performance
                     debugging.
Paths              : {\Hyper-V VM Virtual Device Pipe IO(*)\Receive Message
                     Quota Exceeded, \Hyper-V VM Virtual Device Pipe
                     IO(*)\Receive QoS - Total Message Delay Time (100ns),
                     \Hyper-V VM Virtual Device Pipe IO(*)\Receive QoS -
                     Exempt Messages/sec, \Hyper-V VM Virtual Device Pipe
                     IO(*)\Receive QoS - Non-Conformant Messages/sec...}
PathsWithInstances : {\Hyper-V VM Virtual Device Pipe IO(*)\Receive Message
                     Quota Exceeded, \Hyper-V VM Virtual Device Pipe
                     IO(*)\Receive QoS - Total Message Delay Time (100ns),
                     \Hyper-V VM Virtual Device Pipe IO(*)\Receive QoS -
                     Exempt Messages/sec, \Hyper-V VM Virtual Device Pipe
                     IO(*)\Receive QoS - Non-Conformant Messages/sec...}
```

The `CounterSetName` property returns the category for a group of performance counters. The `Counter` property returns the list of counters which are grouped into that category.

A list of all the *CounterSetNames* can easily be determined by simply returning that single property. Only a subset of the results is shown in the following example.

```
PS C:\> (Get-Counter -ListSet *).CounterSetName

Hyper-V VM Virtual Device Pipe IO
Hyper-V Virtual Machine Health Summary
Storage QoS Filter - Flow
Storage QoS Filter - Volume
Network Virtualization
RAS
Hyper-V VM Vid Partition
WSMan Quota Statistics
BranchCache
Hyper-V VM Vid Numa Node
Hyper-V Virtual Switch
Network QoS Policy
RemoteFX Root GPU Management
AppV Client Streamed Data Percentage
SMB Client Shares
NVIDIA GPU
Hyper-V Virtual Machine Bus Provider Pipes
Windows Time Service
...
```

The results can be limited by filtering left if you have an idea of the specific set of performance counters that you're looking for. In the following example, the results are limited to the ones that are related to *disks*.

```
PS C:\> (Get-Counter -ListSet *disk*).CounterSetName

FileSystem Disk Activity
Storage Spaces Virtual Disk
LogicalDisk
PhysicalDisk
```

For those of you who aren't familiar with the term *filtering left*, that means filtering the results as early as possible in the pipeline or as far to the left as possible to make the process of filtering more efficient.

## Finding Performance Counter Names

Once you've narrowed your choice down to a specific `CounterSetName`, the performance counter names themselves can be determine by returning the `Paths` property.

```
PS C:\> (Get-Counter -ListSet PhysicalDisk).Paths

\PhysicalDisk(*)\Current Disk Queue Length
\PhysicalDisk(*)\% Disk Time
\PhysicalDisk(*)\Avg. Disk Queue Length
\PhysicalDisk(*)\% Disk Read Time
\PhysicalDisk(*)\Avg. Disk Read Queue Length
\PhysicalDisk(*)\% Disk Write Time
\PhysicalDisk(*)\Avg. Disk Write Queue Length
\PhysicalDisk(*)\Avg. Disk sec/Transfer
\PhysicalDisk(*)\Avg. Disk sec/Read
\PhysicalDisk(*)\Avg. Disk sec/Write
\PhysicalDisk(*)\Disk Transfers/sec
\PhysicalDisk(*)\Disk Reads/sec
\PhysicalDisk(*)\Disk Writes/sec
\PhysicalDisk(*)\Disk Bytes/sec
\PhysicalDisk(*)\Disk Read Bytes/sec
\PhysicalDisk(*)\Disk Write Bytes/sec
\PhysicalDisk(*)\Avg. Disk Bytes/Transfer
\PhysicalDisk(*)\Avg. Disk Bytes/Read
\PhysicalDisk(*)\Avg. Disk Bytes/Write
\PhysicalDisk(*)\% Idle Time
\PhysicalDisk(*)\Split IO/Sec
```

## The Top 10 Performance Counters

Remember what I said about finding recommended values on the Internet if you don't already have a performance baseline? There's an article on the Microsoft TechNet Blog titled the "Top 10 most important performance counters for Windows and their recommended values[86]". It provides a list of performance counters and their recommended values. The ones listed in that article are a great starting point to help narrow down the source of performance bottlenecks for a particular system.

## Querying Performance Counters

There aren't many choices when it comes to properties for querying a specific performance counter with `Get-Counter`. There's a `CounterSamples` property which returns each instance of the results for that particular performance counter as a separate result, there's a `Timestamp` property which is self-explanatory, and then there's the `Readings` property which returns all instances of a particular performance counter as a single result.

[86] https://blogs.technet.microsoft.com/bulentozkir/2014/02/14/top-10-most-important-performance-counters-for-windows-and-their-recommended-values/

```
PS C:\> Get-Counter -Counter '\PhysicalDisk(*)\% Idle Time' |
        Get-Member -MemberType Properties

   TypeName:
Microsoft.PowerShell.Commands.GetCounter.PerformanceCounterSampleSet

Name           MemberType     Definition
----           ----------     ----------
CounterSamples Property       Microsoft.PowerShell.Commands.GetCounter.Perfo...
Timestamp      Property       datetime Timestamp {get;set;}
Readings       ScriptProperty System.Object Readings {get=$strPaths = ""...
```

Clearly, `CounterSamples` is the easier of the two when choosing between the properties that return the actual results of a performance counter because each instance is returned by it as a separate result.

One of the problems with the results of the `CounterSamples` property is the computer name and the performance counter being queried are returned jumbled together.

```
PS C:\> (Get-Counter -Counter '\PhysicalDisk(*)\% Idle Time').CounterSamples

Path                                    InstanceName      CookedValue
----                                    ------------      -----------
\\PC01\physicaldisk(1 c:)\% idle time   1 c:         99.9588209059401
\\PC01\physicaldisk(0 d:)\% idle time   0 d:         99.9588209059401
\\PC01\physicaldisk(2 u:)\% idle time   2 u:         99.9588509052801
\\PC01\physicaldisk(_total)\% idle time _total       99.9588309057201
```

It can become messy when you're trying to use a regular expression (regex) to parse the individual information from the `Path` property.

```
PS C:\> (Get-Counter -Counter '\PhysicalDisk(*)\% Idle Time').CounterSamples |
         Select-Object -Property @{label='ComputerName';expression={
                              $_.Path -replace '^\\\\|\\.*$'}},
                       @{label='Object';expression={
                              $_.Path -replace "^\\\\$env:COMPUTERNAME\\|\(.*$"}},
                       @{label='Counter';expression={$_.Path -replace '^.*\\'}},
                       @{label='Instance';expression={$_.InstanceName}},
                       @{label='Value';expression={$_.CookedValue}},
                       @{label='TimeStamp';expression={$_.TimeStamp}}

ComputerName : PC01
Object       : physicaldisk
Counter      : % idle time
Instance     : 1 c:
Value        : 99.7276006677545
```

```
TimeStamp    : 5/21/2018 11:02:47 PM

ComputerName : PC01
Object       : physicaldisk
Counter      : % idle time
Instance     : 0 d:
Value        : 99.987620482975
TimeStamp    : 5/21/2018 11:02:47 PM

ComputerName : PC01
Object       : physicaldisk
Counter      : % idle time
Instance     : 2 u:
Value        : 99.987620482975
TimeStamp    : 5/21/2018 11:02:47 PM

ComputerName : PC01
Object       : physicaldisk
Counter      : % idle time
Instance     : _total
Value        : 99.9009438807218
TimeStamp    : 5/21/2018 11:02:47 PM
```

Luckily the format for the results is the same for all performance counters, or at least I haven't run into one that's different.

## Creating a Reusable Tool

In this section we'll create a reusable tool in the form of a PowerShell function to query the top 10 performance counters. The specific counters that will be queried are listed below.

- `'\PhysicalDisk(*)\% Idle Time'`
- `'\PhysicalDisk(*)\Avg. Disk sec/Read'`
- `'\PhysicalDisk(*)\Avg. Disk sec/Write'`
- `'\PhysicalDisk(*)\Current Disk Queue Length'`
- `'\Memory\Available Bytes'`
- `'\Memory\Pages/sec'`
- `'\Network Interface(*)\Bytes Total/sec'`
- `'\Network Interface(*)\Output Queue Length'`
- `'\Hyper-V Hypervisor Logical Processor(*)\% Total Run Time'`
- `'\Paging File(*)\% Usage'`

Just in case you didn't read the help for `Get-Counter`, let's take a look at the help for the `Counter` parameter. The one specific thing to notice is that more than one performance counter can be queried at the same time without having to call `Get-Counter` for each individual one.

```
PS C:\> help Get-Counter -Parameter Counter

-Counter <String[]>
    Gets data from the specified performance counters. Enter one or more
    counter paths. Wildcards are permitted only in the Instance value. You can
    also pipe counter path strings to Get-Counter.

    Each counter path has the following format:

    "[\\<ComputerName>]\<CounterSet>(<Instance>)\<CounterName>"

    For example:

    "\\Server01\Processor(2)\% User Time"

    The <ComputerName> element is optional. If you omit it, Get-Counter uses
    the value of the ComputerName parameter.

    Note:  To get correctly formatted counter paths, use the ListSet parameter
    to get a performance counter set. The Paths and PathsWithInstances
    properties of each performance counter set contain the individual counter
    paths formatted as a string. You can save the counter path strings in a
    variable or pipe the string directly to another Get-Counter command. For a
    demonstration, see the examples.

    Required?                    false
    Position?                    1
    Default value
    Accept pipeline input?       true (ByValue, ByPropertyName)
    Accept wildcard characters?  true
```

The function shown in the following figure stores the top 10 performance counters in a hash table. It queries the values for each of them while only running the `Get-Counter` cmdlet once. The ability to query these performance counters for a remote computer has also been added to this function. Querying all of the performance counters with a single call to `Get-Counter` has a huge performance impact and exponentially increases the efficiency of the function when querying them on a remote system.

```
#Requires -Version 3.0
function Get-MrTop10Counter {

<#
.SYNOPSIS
  Gets performance counter data from local and remote computers.

.DESCRIPTION
  The Get-MrTop10Counter function gets live, real-time performance counter
  data directly from the performance monitoring instrumentation in Windows.

.PARAMETER ComputerName
  Gets data from the specified computers. Type the NetBIOS name, an Internet
  Protocol (IP) address, or the fully qualified domain names of the computers.
  The default value is the local computer.

.EXAMPLE
  Get-MrTop10Counter -ComputerName Server01, Server02

.INPUTS
  None

.OUTPUTS
  PSCustomObject

.NOTES
  Author:  Mike F Robbins
  Website: http://mikefrobbins.com
  Twitter: @mikefrobbins
#>

  [CmdletBinding()]
  param (
    [ValidateNotNullOrEmpty()]
    [string]$ComputerName = $env:COMPUTERNAME
  )

  $Params = @{
    Counter = '\PhysicalDisk(*)\% Idle Time',
              '\PhysicalDisk(*)\Avg. Disk sec/Read',
              '\PhysicalDisk(*)\Avg. Disk sec/Write',
              '\PhysicalDisk(*)\Current Disk Queue Length',
              '\Memory\Available Bytes',
              '\Memory\Pages/sec',
              '\Network Interface(*)\Bytes Total/sec',
              '\Network Interface(*)\Output Queue Length',
              '\Hyper-V Hypervisor Logical Processor(*)\% Total Run Time',
```

```
            '\Paging File(*)\% Usage'

    ErrorAction = 'SilentlyContinue'
  }

  if ($PSBoundParameters.ComputerName) {
    $Params.ComputerName = $ComputerName
  }

  $Counters = (Get-Counter @Params).CounterSamples

  foreach ($Counter in $Counters){
    [pscustomobject]@{
      ComputerName = $ComputerName
      CounterSetName = $Counter.Path -replace "^\\\\$ComputerName\\|\(.*$"
      Counter = $Counter.Path -replace '^.*\\'
      Instance = $Counter.InstanceName
      Value = $Counter.CookedValue
      TimeStamp = $Counter.Timestamp
    }
  }
}
```

In the following example, the `Get-MrTop10Counter` function is being run against the local computer. Due to space restrictions in this chapter, the results are being filtered down to only the performance counters that are specific to the "C" drive.

```
PS C:\> Get-MrTop10Counter |
        Where-Object Instance -like '*c:'

ComputerName    : PC01
CounterSetName : physicaldisk
Counter         : % idle time
Instance        : 1 c:
Value           : 71.1378247123575
TimeStamp       : 5/26/2018 9:52:11 PM

ComputerName    : PC01
CounterSetName : physicaldisk
Counter         : avg. disk sec/read
Instance        : 1 c:
Value           : 0.014748914084507
TimeStamp       : 5/26/2018 9:52:11 PM

ComputerName    : PC01
CounterSetName : physicaldisk
Counter         : avg. disk sec/write
```

```
Instance       : 1 c:
Value          : 0.000612503821656051
TimeStamp      : 5/26/2018 9:52:11 PM

ComputerName   : PC01
CounterSetName : physicaldisk
Counter        : current disk queue length
Instance       : 1 c:
Value          : 40
TimeStamp      : 5/26/2018 9:52:11 PM
```

It's difficult to determine if the value returned by a specific performance counter is normal or not without having something to compare it to. This is why it's best to have a baseline as I previously mentioned, but values that are considerably outside of the normal range for a specific counter can be found on the Internet.

Based on the article[87] that I previously mentioned, the *Current Disk Queue Length* should not be greater than 2 and the current value for the "C" drive of the computer used in this chapter is 40 so that would be a specific area that needs more investigation.

# Automate the Validation of Performance Counters

While querying performance counters with PowerShell is relatively easy as shown in the previous portion of this chapter, who really wants to query each one of them and validate they're within an acceptable range manually? I'm assuming that no one does since all I'm hearing is crickets.

## Pester

In this section, we'll use Pester to automate the testing of whether or not the performance counter values returned by `Get-MrTop10Counter` are within their recommended ranges.

Pester is an open-source Behavior-Driven Development (BDD) based framework for PowerShell. While a version of Pester ships with Windows 10, it's an older and out of date version that must be updated before attempting to run the examples shown in this chapter.

I recommend installing the latest version of Pester from the PowerShell Gallery[88]. While `Update-Module` may work depending on whether or not you've previously updated Pester, the following command will work regardless of which version you currently have installed as long as the computer it's being run on is connected to the Internet.

[87] https://blogs.technet.microsoft.com/bulentozkir/2014/02/14/top-10-most-important-performance-counters-for-windows-and-their-recommended-values/

[88] https://www.powershellgallery.com/

```
PS C:\> Install-Module -Name Pester -Force -SkipPublisherCheck
```

More information about Pester can be found on its Wiki[89].

## Validation Tests

We'll start out by writing a simple validation test for the *Current Disk Queue Length* for the "C" drive on your local computer.

```
PS C:\> Describe 'Current Disk Queue Length' {
          It 'Should not be higher than 2' {
            (Get-Counter -Counter '\PhysicalDisk(* c:)\Current Disk Queue Length'
            ).CounterSamples.CookedValue |
            Should -Not -BeGreaterThan 2
          }
        }

Describing Current Disk Queue Length
  [+] Should not be higher than 2 1.02s
```

## Testing Collections

A helper function to convert the results of a command to a hash table is needed in the next section. Luckily, I had previously written a function to accomplish this exact task for similar scenario at some point in the past and saved it in my PowerShell repository on GitHub[90]. This helper function, `ConvertTo-MrHashTable`, is also included as part of the downloadable extras of this book.

As many computers do, the computer used in this chapter has multiple hard drives. We'll need to iterate through each one of them individually with our infrastructure test. While we could use a `foreach` loop to prevent writing the same redundant code for each one of them over and over again, Pester has a `TestCases` parameter which is specifically designed for this exact scenario.

[89] https://github.com/pester/Pester/wiki
[90] https://github.com/mikefrobbins/PowerShell

```
PS C:\> $Counters = Get-MrTop10Counter
PS C:\> Describe 'Physical Disk Current Disk Queue Length' {
        $Counter = 'Current Disk Queue Length'
        $Cases = $Counters.Where({
                   $_.Counter -eq $Counter -and $_.Instance -ne '_total'
                 }) |
                 Select-Object -Property Instance |
                 ConvertTo-MrHashTable

        It 'Should Not Be Greater than 2 for: <Instance>' -TestCases $Cases {
          param($Instance)
          $Counters.Where({
            $_.Instance -eq $Instance -and $_.Counter -eq $Counter
          }).Value |
          Should -Not -BeGreaterThan 2
        }
      }

Describing Current Disk Queue Length
  [+] Should Not Be Higher than 2 for: '0 c:' 29ms
  [+] Should Not Be Higher than 2 for: '1 d:' 37ms
```

The total for all instances (all hard drives) in the computer has been excluded because for this particular test, we're concerned about testing each hard drive individually and not the total for all of them combined.

## Advanced Validation Tests

Now it's time to write these same types of infrastructure validation tests for all of the top 10 performance counters that `Get-MrTop10Counter` queries. Writing validation tests for some of the counters is more complicated than others. Validating that the percent idle time for a physical disk is not less than sixty percent is simple because the results are returned as a percentage by default.

```
18 Describe "Physical Disk % Idle Time for $Computer" {
19   $Counter = '% Idle Time'
20   $Cases = $Counters.Where({
21              $_.Counter -eq $Counter -and $_.Instance -ne '_total'
22            }) |
23            Select-Object -Property Instance |
24            ConvertTo-MrHashTable
25
26   It 'Should Not Be Less than 60% for: <Instance>' -TestCases $Cases {
27     param($Instance)
28     $Counters.Where({
29       $_.Instance -eq $Instance -and $_.Counter -eq $Counter
30     }).Value |
```

```
    Should -Not -BeLessThan 60
  }
}
```

Validating the average time that disk transfers took is another story since the counters return seconds instead of milliseconds and it's not uncommon to see the results returned in scientific notation instead of a numeric datatype that can be used for normal calculations.

```
Describe "Physical Disk Avg. Disk sec/Read for $Computer" {
  $Counter = 'Avg. Disk sec/Read'
  $Cases = $Counters.Where({
             $_.Counter -eq $Counter -and $_.Instance -ne '_total'
           }) |
           Select-Object -Property Instance |
           ConvertTo-MrHashTable

  It 'Should Not Be Greater than 20ms for: <Instance>' -TestCases $Cases {
    param($Instance)
    $Counters.Where({
      $_.Instance -eq $Instance -and $_.Counter -eq $Counter
    }).Value * 1000 -as [decimal] |
    Should -Not -BeGreaterThan 20
  }
}

Describe "Physical Disk Avg. Disk sec/Write for $Computer" {
  $Counter = 'Avg. Disk sec/Write'
  $Cases = $Counters.Where({
             $_.Counter -eq $Counter -and $_.Instance -ne '_total'
           }) |
           Select-Object -Property Instance |
           ConvertTo-MrHashTable

  It 'Should Not Be Greater than 20ms for: <Instance>' -TestCases $Cases {
    param($Instance)
    $Counters.Where({
      $_.Instance -eq $Instance -and $_.Counter -eq $Counter
    }).Value * 1000 -as [decimal] |
    Should -Not -BeGreaterThan 20
  }
}
```

Determining if at least ten percent of memory is available is another tricky one because the performance counter returns the currently available bytes of memory, but you need to know how much physical memory is installed in the machine to be able to calculate the percentage. Figuring this out on a remote system only complicates matters even further.

```
Describe "Memory Available Bytes for $Computer" {
  It 'Should Not Be Less than 10% free' {
    ($Counters.Where({$_.Counter -eq 'Available Bytes'}).Value / 1MB) /
    ((Get-CimInstance @Params -ClassName Win32_PhysicalMemory -Property Capacity |
     Measure-Object -Property Capacity -Sum).Sum / 1MB) * 100 -as [int] |
    Should -Not -BeLessThan 10
  }
}
```

Verifying that a network card's available bandwidth isn't saturated is also complicated. The performance counter returns total bytes a second. You'll need to determine the current link speed of the network adapter in order to be able to calculate that it's not more than sixty-five percent utilized. If that weren't complicated enough, the performance counter returns parentheses as brackets in the network cards description.

```
Describe "Network Interface Bytes Total/sec for $Computer" {
  $Counter = 'Bytes Total/sec'
  $Cases = $Counters.Where({
              $_.Counter -eq $Counter -and $_.Instance -notmatch 'isatap'}) |
           Select-Object -Property Instance |
           ConvertTo-MrHashTable

  It 'Should Not Be Greater than 65% for: <Instance>' -TestCases $Cases {
    param($Instance)
    ($Counters.Where({
        $_.Instance -eq $Instance -and $_.Counter -eq $Counter
    }).Value) /
    ((Get-NetAdapter @Params -InterfaceDescription (
      $Instance -replace '\[', '(' -replace '\]', ')' -replace '_', '#')).Speed
    ) * 100 |
    Should -Not -BeGreaterThan 65
  }
}
```

To run these tests on a remote system, both a Common Information Model (CIM) session needs to be established to it and PowerShell needs to be run with enough privileges to query performance counters on the remote system.

The code for the tests used to validate all of the top 10 performance counters, `Performance-Counter-Validation.Tests.ps1`, is included as part of the downloadable extras of this book. Since these tests have been saved as a PowerShell script with a *.tests.ps1* extension, they can be run against the local system by simply running `Invoke-Pester` as shown in the following example.

```
PS C:\> Invoke-Pester -Script .\Performance-Counter-Validation.Tests.ps1
Executing all tests in '.\Performance-Counter-Validation.Tests.ps1'

Executing script .\Performance-Counter-Validation.Tests.ps1

  Describing Physical Disk % Idle Time for PC01
    [+] Should Not Be Less than 60% for: '0 c:' 1.07s

  Describing Physical Disk Avg. Disk sec/Read for PC01
    [+] Should Not Be Greater than 20ms for: '0 c:' 399ms

  Describing Physical Disk Avg. Disk sec/Write for PC01
    [+] Should Not Be Greater than 20ms for: '0 c:' 36ms

  Describing Physical Disk Current Disk Queue Length for PC01
    [+] Should Not Be Greater than 2 for: '0 c:' 29ms

  Describing Memory Available Bytes for PC01
    [+] Should Not Be Less than 10% free 32ms

  Describing Memory Pages/sec for PC01
    [+] Should Not Be Greater than 1000 25ms

  Describing Network Interface Bytes Total/sec for PC01
    [+] Should Not Be Greater than 65% for: 'microsoft hyper-v network adapter' 53ms

  Describing Network Interface Output Queue Length for PC01
    [+] Should Not Be Greater than 2 for: 'microsoft hyper-v network adapter' 30ms

  Describing Hyper-V Hypervisor Logical Processor % Total Run Time for PC01
    [+] Should Not Be Greater than 90% for: <Instance> 30ms

  Describing Paging File % Usage for PC01
    [+] Should Not Be Greater than 10% for: '\??\c:\pagefile.sys' 54ms

Tests completed in 1.77s
Tests Passed: 10, Failed: 0, Skipped: 0, Pending: 0, Inconclusive: 0
```

As shown in the previous example, these tests can easily be run against the local computer to get an idea of where performance bottlenecks reside. To run it against a remote system, simply create a CIM session, run the script itself, and specify the `CimSession` parameter.

```
PS C:\> $CimSession = New-CimSession -ComputerName Server01
PS C:\> .\Performance-Counter-Validation.Tests.ps1 -CimSession $CimSession
```

You're looking for anything that the tests return in red instead of green.

# Summary

In this chapter you've learned how to find performance bottlenecks of Windows based systems with PowerShell. You'll now be able to find specific performance counters for yourself, query those performance counters and validate that they're within acceptable ranges.

The tests shown in this chapter were used to diagnose the performance problems described in the scenario from the introduction to this chapter. The current disk queue length was excessively high on the Hyper-V hosts that the virtual desktop infrastructure (VDI) workstations were running on in that scenario. The root cause of the high disk queue length was determined to be due to a newly installed antivirus solution on the VDI workstations themselves.