

Master PowerShell Tricks

Volume 2

Dave Kawula - MVP
Thomas Rayner - MVP
Sean Kearney - MVP
Allan Rafuse - MVP
Will Anderson – MVP
Mick Pletcher – MVP
Ed Wilson – The Scripting Guy

PUBLISHED BY
MVPDays Publishing
<http://www.mvpdays.com>

Copyright © 2017 by MVPDays Publishing

All rights reserved. No part of this lab manual may be reproduced or transmitted in any form or by any means without the prior written permission of the publisher.

ISBN: 978-1542677967

Warning and Disclaimer

Every effort has been made to make this manual as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Feedback Information

We’d like to hear from you! If you have any comments about how we could improve the quality of this book, please don’t hesitate to contact us by visiting www.mvpdays.com or sending an email to feedback@mvpdays.com.

Foreword Ed Wilson “The Scripting Guy”

There are many cool things about Windows PowerShell. For me three of the most awesome things are the following:

1. If you don't like the way Windows PowerShell does things, you can change it.
2. If Windows PowerShell can't do something you need, you can add it.
3. The Windows PowerShell community is super dedicated, and will help you do both one and two.

Now, to be honest at times, it is necessary to write hundreds and hundreds of lines of arcane code, to dive into the deepest and darkest mysteries of programing, and even to learn about things like API's, Constructors, Events, threading, ACL's, DACL's, CACL's and maybe even Tetradactylies. Then again, most of the time it is not. In fact, it has been years and years since I wrote a hundreds and hundreds of lines of arcane code.

It is almost as if the Windows PowerShell team deliberately tried to make Windows PowerShell easy to use and easy to learn. Hmm ... I wonder if that approach would ever catch on? Anyway, there used to be an old saw: “Ease of use is directly opposed to program capability.” Or, in other words, if it is easy to use, it probably is not all that powerful. Well, PowerShell changes that ... dramatically.

And yet, Windows PowerShell is also deceptively easy to use. One can go from Get-Service and Get-Process or even Get-Date to some pretty complicated stuff in like one line of code.

This is why item number three is so important. The authors of this book: Dave, Sean, Will, Mick, Thomas and Allan are all Windows PowerShell experts, and have even been recognized by Microsoft as community leaders. So, this means not only do they know their stuff, but they are also great at sharing that knowledge with the community. Sean Kearney is even an Honorary Scripting Guy – a very elite group indeed!

One of the great way that MVP’s share their knowledge and experience is via MVP Days a traveling road show that was started by Dave and Cristal Kawula. This is a very well run event, and I have had the opportunity to speak at two of the events ... it is cool, and it is fun.

So grab this book, get it autographed, and learn how to master some awesome PowerShell tricks. It is cool.

Ed Wilson

Microsoft Scripting Guy

@ScriptingGuys

Acknowledgements

From Dave

Cristal, you are my rock and my source of inspiration. For the past 20 + years you have been there with me every step of the way. Not only are you the “BEST Wife” in the world you are my partner in crime. Christian, Trinity, Keira, Serena, Mickaila and Mackenzie, you kids are so patient with your dear old dad when he locks himself away in the office for yet another book. Taking the time to watch you grow in life, sports, and become little leaders of this new world is incredible to watch.

Thank you, Mom and Dad (Frank and Audry) and my brother Joe. You got me started in this crazy IT world when I was so young. Brother, you mentored me along the way both coaching me in hockey and helping me learn what you knew about PC's and Servers. I'll never forget us as teenage kids working the IT Support contract for the local municipal government. Remember dad had to drive us to site because you weren't old enough to drive ourselves yet. A great career starts with the support of your family and I'm so lucky because I have all the support one could ever want.

A book like this filled with amazing Canadian MVP's would not be possible without the support from the #1 Microsoft Community Program Manager – Simran Chaudry. You have guided us along the path and helped us to get better at what we do every day. Your job is tireless and your passion and commitment make us want to do what we do even more.

Last but not least, the MVPDays volunteers, you have donated your time and expertise and helped us run the event in over 20 cities across North America. Our latest journey has us expanding the conference worldwide as a virtual conference. For those of you that will read this book your potential is limitless just expand your horizons and you never know where life will take you.

About the Authors

Dave Kawula - MVP

Dave is a Microsoft Most Valuable Professional (MVP) with over 20 years of experience in the IT industry. His background includes data communications networks within multi-server environments, and he has led architecture teams for virtualization, System Center, Exchange, Active Directory, and Internet gateways. Very active within the Microsoft technical and consulting teams, Dave has provided deep-dive technical knowledge and subject matter expertise on various System Center and operating system topics.

Dave is well-known in the community as an evangelist for Microsoft, IE, and Veeam technologies. Locating Dave is easy as he speaks at several conferences and sessions each year, including TechEd, Ignite, MVP Days Community Roadshow, and VeeamOn.

As the founder and Managing Principal Consultant at TriCon Elite Consulting, Dave is a leading technology expert for both local customers and large international enterprises, providing optimal guidance and methodologies to achieve and maintain an efficient infrastructure.

BLOG: www.checkyourlogs.net

Twitter: @DaveKawula



Sean Kearney - MVP

`$PowerShellMVPBio=@'`

"A long time ago in a Cmdlet far far away... there was this guy, who sang about PowerShell"

Here is a person who genuinely loves his job and smiles he gets paid to do what he loves, It's PowerShell MVP Sean Kearney

Presently working for a Microsoft Gold Partner in Ottawa as a Senior Solutions Architect, he lives each and every day for an opportunity to show someone and easier and more consistent way to do their job with Windows PowerShell.

BLOG: <http://www.energizedtech.com/>

Twitter: [@energizedtech](https://twitter.com/energizedtech)



Thomas Rayner - MVP

Thomas Rayner is an information technology, entrepreneurship and leadership enthusiast with a penchant for Microsoft tools and products. Thomas is a proud graduate of several programs at NAIT, an institution that he remains actively connected to. He works on the DevOps and Automation team at PCL Construction.

BLOG: <http://workingsysadmin.com>

Twitter: [@mrthomasrayner](https://twitter.com/mrthomasrayner)



Allan Rafuse – MVP

Allan has worked as a senior member of the Windows and VMWare Platform Department at Swedbank. He took part in the architecture and implementation of multiple datacenters in several countries. He is responsible for the roadmap and lifecycle of the Windows Server Environment, including the development of ITIL processes of global server OSD, configuration, and performance.

He is an expert at scripting solutions and has an uncanny ability to reduce complexity and maximize the functionality of PowerShell. Allan has recently rejoined the TriCon Elite Consulting team again as a Principal Consultant.

BLOG: <http://www.checkyourlogs.net>

Twitter: @allanrafuse

Will Anderson – MVP

Will Anderson is a fifteen-year infrastructure veteran with a specialization in Patch Management and Compliance and System Center Configuration Manager. Working in environments ranging from 80 users to over 150,000, Will has acquired a knowledge of a broad range of products and service lines ranging from Exchange, Active Directory and GPO, to the operating system platform and a variety of applications.

In recent years, Will has become quite the nerd about PowerShell, and blogs about the latest, new, cool things he finds or creates to make his life as an admin and engineer easier. You can find him on PowerShell.org as a moderator, webmaster, and occasional writer for the PowerShell TechLetter. He is also a co-founder of the Toronto PowerShell Users' Group (PowerShellTO), founder of the Metro Detroit PowerShell User Group, and a member of the Association for Windows PowerShell Professionals.

Will is a second year recipient of the Microsoft MVP award in Cloud and Datacenter Management, and was awarded the moniker of 2015 Honorary Scripting Guy, by Ed Wilson – The Scripting Guy, in January 2016. In October of 2016, he joined the DevOps Collective Board of Directors.

Will also nerds out on Video Games, Cars, Photography, and Board Games. You can find him at various places on the internet including PowerShellTO, PowerShell.org, Twitter, his personal blog – Last Word in Nerd, and occasionally as a guest blogger on 'Hey, Scripting Guy!'.

BLOG: <http://lastwordinnerd.com>

Twitter: @GamerLivingWill



Mick Pletcher – MVP

I started out as a help desk tech at J.C. Bradford in 1999. From that point on, I moved to third level help desk. It was this position in which I got my first taste of scripting and automation. Most in IT know what it is like having tickets escalated to them and not knowing exactly what was done to resolve an issue. It was that very thing that prompted me to learn scripting and write automation scripts so that when a ticket was escalated, I knew exactly what had taken place. In 2005, I left that company and moved on to a management IT position. I did everything at that point. It was a one man job. I have to say that I learned a lesson on being the only IT person. It is stressful to say the least. Much of that position was not automated. I really got into automation big time there. By the time I was finished writing so many scripts, I had maybe an hour of work a day. At this point, I would be so bored in that position that I took up learning new skills, which led me to building a computer build lab for system builds. It was this very thing that got me where I am today. I learned all about packaging files, creating a system build, scripting, automation, and VBScript. Soon after learning all of this, the music industry was taking enormous hits and I knew I had to leave before my job got cut, which it did 6 months later. My new job was what I do today....an SCCM administrator. The skills I had learned in the previous job of packaging and scripting lead right into software deployments as a next step. In this position I was still scripting with VBScript, but began slowly transitioning in 2011 to PowerShell. Finally, I was offered my current position also as an SCCM administrator. It has been very rewarding.

Blog: <http://mickitblog.blogspot.ca>

Twitter: @mick_pletcher



Ed Wilson – The Scripting Guy

Ed Wilson is a well-known scripting expert. He is a Microsoft Certified Trainer who has delivered a popular Windows PowerShell workshop to Microsoft Premier customers worldwide. Ed has written six books on Microsoft Windows scripting for Microsoft Press. His three Windows PowerShell books are Windows PowerShell 2.0 Best Practices, Windows PowerShell Scripting Guide, and Microsoft Windows PowerShell Step by Step. He has also written or contributed to almost a dozen other books. Ed holds more than 20 industry certifications, including Microsoft Certified Systems Engineer (MCSE) and Certified Information Systems Security Professional (CISSP). Before coming to work for Microsoft, Ed was a senior consultant for a Microsoft Gold Certified Partner where he specialized in Active Directory design and Exchange Server implementation. Ed and the “Scripting Wife” Teresa live in South Carolina. In his spare time, he enjoys woodworking, underwater photography, and scuba diving. And tea.

BLOG: <https://blogs.technet.microsoft.com/heyscriptingguy/>

Twitter: @scriptingguys

Technical Editors

Cristal Kawula – MVP

Cristal Kawula is the co-founder of MVPDays Community Roadshow and #MVPHour live Twitter Chat. She was also a member of the Gridstore Technical Advisory board and is the President of TriCon Elite Consulting. Cristal is also only the 2nd Woman in the world to receive the prestigious Veeam Vanguard award.

BLOG: <http://www.checkyourlogs.net>

Twitter: @supercrystal1

Emile Cabot - MVP

Emile started in the industry during the mid-90s working at an ISP and designing celebrity web sites. He has a strong operational background specializing in Systems Management and collaboration solutions, and has spent many years performing infrastructure analyses and solution implementations for organizations ranging from 20 to over 200,000 employees. Coupling his wealth of experience with a small partner network, Emile works very closely with TriCon Elite, 1E, and Veeam to deliver low-cost solutions with minimal infrastructure requirements.

He actively volunteers as a member of the Canadian Ski Patrol, providing over 250 hours each year for first aid services and public education at Castle Mountain Resort and in the community.

BLOG: <http://www.checkyourlogs.net>

Twitter: @ecabot



Kai Poynting

Kai Poynting has over ten years of experience in the technical writing field. as a freelance technical writer, senior technical writer for one of the larger energy software companies in Calgary Alberta, and experience writing about solutions for IT.

In addition to writing about solutions for IT, those ten years were also spent testing, building and deploying some of the same solutions. As part of a small group of consultants, Kai was provided with a great many opportunities to obtain hands on experience in installing, configuring, building, and managing various server solutions, including Server 2008 R2, Exchange 2007 and 2010, Hyper-V, SCVMM, and more. He also provided customer support and technical support for the company's classroom environment, including requirements management, tech support, deployment, and server configuration.

He also holds a BA with an English major, and writes creatively whenever he can. Communication is the cornerstone of his profession and he prides himself on being able to provide the clearest possible message in all documents he provides.

Contents

Foreword Ed Wilson “The Scripting Guy”	iii
Acknowledgements.....	v
From Dave	v
About the Authors	vi
Dave Kawula - MVP	vi
Sean Kearney - MVP	vii
Thomas Rayner - MVP	vii
Allan Rafuse – MVP.....	viii
Will Anderson – MVP	ix
Mick Pletcher – MVP.....	x
Ed Wilson – The Scripting Guy	xi
Technical Editors	xii
Cristal Kawula – MVP.....	xii
Emile Cabot - MVP.....	xii
Kai Poynting.....	xiii
Contents.....	xiv
Introduction	1
North American MVPDays Community Roadshow.....	1
Structure of the Book	2
Sample Files.....	3
Additional Resources	3
Chapter 1.....	5
PowerShell Regex to Get Value Between Quotation Marks.....	5
Chapter 2.....	7
How to Send an Email Whenever a File Gets Changed	7
Chapter 3.....	8

Using PowerShell to Add Groups to Accept messages only Distribution list members	8
The Code	9
Chapter 4.....	10
Installing and Finding Modules	10
Why use Install-Modules	10
Module Dependencies	10
Repositories.....	11
Managing Repositories	11
Finding Modules in the Repositories	11
Installing a Module	13
Proxy Configuration	14
Summary	14
Chapter 5.....	15
Use PowerShell to Validate Parameters for you.....	15
Common PowerShell Functions – NO Validation	16
PowerShell Features for Parameters	16
Problem 1.....	16
Problem 2.....	17
Problem 3.....	17
Solution	18
Other Parameter Validation Features	18
Chapter 6.....	19
PowerShell – My Way to Prepare a SQL Server VM	19
Template File.....	20
A look at library-PrepareSQLServer.ps1	21
Running the script.....	23
Chapter 7.....	25
Building Gold VHDx images with PowerShell	25
Define the Variables.....	26
Build \$unattendSource.....	26
Get-UnattendChunk.....	27
New-Unattendfile	28
New-BaselImage	29

Chapter 8.....	32
Using Try Catch to help with Decision Making in a Script	32
Chapter 9.....	36
Invoking your SCCM Client Remotely with PowerShell.....	36
Chapter 10.....	38
Use PoweShell to Parse RSS Feeds.....	38
Chapter 11.....	44
How to alter the public IP Address of an Azure Virtual Machine using PowerShell	44
The Challenge	44
Data formatting or using an existing example.....	44
Make the change	46
Chapter 12.....	48
Get the Public IP of an Azure VM with PowerShell	48
Chapter 13.....	49
SCCM Local Administrator Reporting with PowerShell.....	49
LocalAdmins.ps1	53
Chapter 14.....	58
Configuring Power Settings using PowerShell	58
Set-PowerScheme.ps1	58
Chapter 15.....	73
Automated SCCM Endpoint Full System Scan upon Infection with Email Notification.....	73
ApplicationVirusDetectionMethodEmail.ps1	75
AntiVirusScanEmail.ps1	76
Chapter 16.....	79
Laptop Mandatory Reboot Management with SCCM and PowerShell	79
MandatoryReboot.ps1.....	80

MandatoryRebootCustomDetection.ps1.....	82
Chapter 17.....	84
Set Windows Features with Verification	84
WindowsFeatures.ps1	85
Chapter 18.....	90
Easily Restore a Deleted Active Directory User with PowerShell	90
Chapter 19.....	91
Getting Large Exchange Mailbox Folders with PowerShell.....	91
Chapter 20.....	95
Getting your Organizations Largest Exchange Mailboxes with PowerShell	95
Chapter 21.....	99
Just Enough Administration (JEA) First Look.....	99
So how do you get started?	99
Chapter 22.....	105
Does a String Start or End in a Certain Character?	105
Chapter 23.....	107
Using PowerShell to List All the Fonts in a Word Document.....	107
Chapter 24.....	109
Find PowerShell Scripts in the GUI.....	109
Chapter 25.....	116
How to use PowerShell as an Administrative Console.....	116
Chapter 26.....	118
How to Erase Files based on Date using PowerShell	118
Chapter 27.....	121

Join us at MVPDays and meet great MVP's like this in person.....	121
Live Presentations	121
Video Training.....	121
Live Instructor-led Classes.....	121
Consulting Services	122
Twitter.....	123

Introduction

North American MVPDays Community Roadshow

The purpose of this book is to showcase the amazing expertise of our guest speakers at the North American MVPDays Community Roadshow. They have so much passion, expertise, and expert knowledge that it only seemed fitting to write it down in a book.

MVPDays was founded by Cristal and Dave Kawula back in 2013. It started as a simple idea; “There’s got to be a good way for Microsoft MVPs to reach the IT community and share their vast knowledge and experience in a fun and engaging way” I mean, what is the point in recognizing these bright and inspiring individuals, and not leveraging them to inspire the community that they are a part of.

We often get asked the question “Who should attend MVPDays”?

Anyone that has an interest in technology, is eager to learn, and wants to meet other like-minded individuals. This Roadshow is not just for Microsoft MVP’s it is for anyone in the IT Community.

Make sure you check out the MVPDays website at: www.mvppdays.com. You never know maybe the roadshow will be coming to a city near you.

The goal of this particular book is to give you some amazing Master PowerShell tips from the experts you come to see in person at the MVPDays Roadshow. Each chapter is broken down into a unique tip and we really hope you find some immense value in what we have written.

Structure of the Book

Chapter 1 in this chapter Thomas Rayner shows use Regex to a values between quotation marks.

Chapter 2 Thomas Rayner shows how to send an Email whenever a file gets changed.

Chapter 3 Thomas Rayner demonstrates add groups to accept messages only from distribution list members.

Chapters 4 in this chapter Allan Rafuse shows us how to install and find PowerShell modules.

Chapter 5 Allan Rafuse will show you how to use PowerShell to validate parameters.

Chapter 6 Allan Rafuse demonstrates how to prepare a SQL Server installation using PowerShell

Chapter 7 in this chapter Dave Kawula shows how to provision Gold VHDx images using PowerShell from an ISO

Chapter 8 Will Anderson uses Try-Catch to help make decisions inside of a PowerShell script

Chapter 9 Will Anderson shows a cool PowerShell technique to remotely invoke the SCCM Client using PowerShell

Chapter 10 Ed Wilson and Will Anderson show us how to use PowerShell to parse RSS Feeds

Chapter 11 Will Anderson shows us how to alter the Public IP Address of an Azure Virtual Machine using PowerShell

Chapter 12 In this quick master tip Will Anderson show us how to get the Public IP Address of an Azure Virtual Machine using PowerShell

Chapter 13 Mick Pletcher shows us to get reports on the Local Administrators of workstations using SCCM and PowerShell

Chapter 14 Mick Pletcher demonstrates how to configure Power Settings using his custom PowerShell scripts.

Chapter 15 in this chapter Mick Pletcher shows how to configure SCCM Endpoint full System Scans upon infection with Email Alerting

Chapter 16 Mick Pletcher shows how to for mandatory reboots using SCCM and PowerShell

Chapter 17 Mick Pletcher shows how to install Windows Features with validation with his custom script

Chapter 18 Thomas Rayner is back to show you how to easily restore a deleted Active Directory user with PowerShell

Chapter 19 Thomas Rayner shows how to report on Large Exchange mailbox folders with PowerShell

Chapter 20 Thomas Rayner walks you through a trick to report on the largest Exchange mailboxes using PowerShell.

Chapter 21 Thomas Rayner introduces you to Just Enough Administration (JEA)

Chapter 22 Thomas Rayner helps us figure out if a string starts or ends with a certain character

Chapter 23 in this chapter Thomas helps us list all of the fonts in a word document using PowerShell

Chapter 24 Sean Kearney shows how to find PowerShell scripts inside of the Windows GUI

Chapter 25 Sean Kearney demonstrates how we can use PowerShell as an Administrative Console

Chapter 26 Sean Kearney shows how to erase files based on date using PowerShell

Chapter 27 Did you like what you read? This is how you find our experts

Sample Files

All sample files for this book can be downloaded from <http://www.checkyourlogs.net>

Additional Resources

In addition to all tips and tricks provided in this book, you can find extra resources like articles and video recordings on our blog <http://www.checkyourlogs.net>.

Chapter 1

PowerShell Regex to Get Value Between Quotation Marks

By: Thomas Rayner – MVP

If you've got a value like the following...

```
$s = @"
Here is: "Some data"
Here's "some other data"
this is "important" data
"@
```

... that maybe came from the body of a file, was returned by some other part of a script, etc., and you just want the portions that are actually between the quotes, the quickest and easiest way to get it is through a regular expression match.

That's right, forget splitting or trimming or doing other weird string manipulation stuff. Just use the `[regex]::matches()` feature of PowerShell to get your values.

```
[regex]::matches($s, '(?<=\\").+?(?=\\")').value
```

Matches takes two parameters. 1. The value to look for matches in, in this case the here-string in my `$s` variable, and 2. The regular expression to be used for matching. Since Matches returns a few items, we are making sure to just select the value for each match.

So what is that regex doing? Let's break it down into it's parts.

- `(?<=\\")` this part is a look behind as specified by the `?<=` part. In this case, whatever we are matching will come right after a quote. Doing the look behind prevents the quotation mark itself from actually being part of the matched value. Notice I have to escape the quotation mark character.
- `.+?` this part basically matches as many characters as it takes to get to whatever the next part of the regex is. Look into regex lazy mode vs greedy mode.
- `(?=\\")` this part is a look ahead as specified by the `?=` part. We're looking ahead for a quotation mark because whatever comes after our match is done will be a quotation mark.

So basically, what we've got is "whatever comes after a quotation mark, and as much of that as you need until you get to another quotation mark". Easy, right? Don't you love regex?

Chapter 2

How to Send an Email Whenever a File Gets Changed

By: Thomas Rayner – MVP

A little while ago, I fielded a question in the PowerShell Slack channel which was “How do I send an email automatically whenever a change is made to a specific file?”

Turns out it’s not too hard. You just need to set up a file watcher.

```
$watcher = New-Object System.IO.FileSystemWatcher
$watcher.Path = 'C:\temp\'
$watcher.Filter = 'test1.txt'
$watcher.EnableRaisingEvents = $true

$changed = Register-ObjectEvent $watcher 'Changed' -Action {
    write-output "Changed: $($eventArgs.FullPath)"
}
```

First, we create the watcher, which is just a FileSystemWatcher object. Technically the watcher watches the whole directory for changes (the path), which is why we add a filter.

Then we register an ObjectEvent, so that whenever the watcher sees a change event, it performs an action. In this case, I just have it writing output but it could easily be sending an email or performing some other task.

To get rid of the ObjectEvent, just run the following.

```
Unregister-Event $changed.Id
```

Chapter 3

Using PowerShell to Add Groups to Accept messages only Distribution list members

By: Thomas Rayner - MVP

Identifying users with large Exchange mailboxes is a task undertaken by most system administrators who are in need of freeing up space on their mail servers. While most search for mailboxes approaching a certain size, I wanted to take this a step further and identify the large folders within user mailboxes. An example of this would be to find all the users who have a large Deleted Items folder or Sent Items or Calendar that would be eligible to be cleaned out. It's made to be run from a Remote Exchange Management Shell connection instead of by logging into an Exchange server via remote desktop and running such a shell.



Figure 1 – Large Mailbox Folders

The Code

```
function Add-AcceptMessagesOnlyFromDLMembers
{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [string]$AppendTo,
        [Parameter(Mandatory)]
        [string]$DLName
    )

    $arr = $(Get-MailContact $AppendTo | Select-Object
AcceptMessagesOnlyFromDLMembers).AcceptMessagesOnlyFromDLMembers
    $arr += ($(Get-ADGroup $NameOfGroup -Properties
CanonicalName).CanonicalName)
    set-mailContact $AppendTo -AcceptMessagesOnlyFromDLMembers:"${$arr}"
}
```

First things first, I declare the function named `Add-AcceptMessagesOnlyFromDLMembers` which is a bit more verbose than I'd usually like to make it, but I'm also a fan of descriptive function and cmdlet names.

Second, I need some parameters. The mail recipient whose `AcceptMessagesOnlyFromDLMembers` value we're appending something to, and the DL that we're appending.

Line 11 is where we begin doing the real work. I've got to get the mail contact and select just the value currently in `AcceptMessagesOnlyFromDLMembers` so I can append something to it. I store that data in `$arr`.

On line 12, I'm retrieving the `CanonicalName` attribute for the DL I want to append to the list of DLs that can send mail to this contact. The `AcceptMessagesOnlyFromDLMembers` attribute is a bit weird in that it only appears to take Canonical Names, not Distinguished names, etc.. I'm appending that value to the end of `$arr`.

Line 13 is pretty straight forward. I'm setting the `AcceptMessagesOnlyFromDLMembers` attribute to the value of `$arr` determined in line 12.

That's it! If this is a task you perform regularly, please take this script and apply it. If you make it more robust, I'd love to see what your modifications are.

Chapter 4

Installing and Finding Modules

By: Allan Rafuse – Future MVP

Install-Module is a wonderful new cmdlet that comes with PowerShell v5 and can be found in Windows Management Framework (WMF) 5.0. This allows us to skip the whole search the Internet to find modules and pull them from pre-configured repositories. By default, your machines should be configured to look at <https://www.powershellgallery.com/api/v2/>.

Why use Install-Modules

It makes your life easier! Instead of going to the Internet, searching, downloading the files and then putting them into the appropriate directories, you can use Install-Module. Doesn't that just make your life easier?

But again Why? What modules are out there that I should or could be using? I'll give two primary examples where I use this all the time, Microsoft Azure and Desired State Configuration (DSC). There are so many Microsoft teams that provide supported PowerShell modules for you to download and use, similarly there are many 3rd party and individuals who also author their own modules to help out the community.

Did someone in the community just release an update? Now we can just Update-Module and we're done!

Module Dependencies

You would think that running the new command would be easy and it would just work. In most cases it does, but in my experience you'll be prompted a few times to install things here and there. Underneath the covers Install-Module uses the module, PowerShellGet. PowerShellGet uses a Package Provider provided by Microsoft call NuGet. I hope you're not lost yet! That should be as deep as we need to go to see what is actually going on when we use Install-Module.

By default, the NuGet Package Management provider is not installed by default. You can install it by running:

```
Install-PackageProvider NuGet
```

Once we install the NuGet Package Provider, we're ready to go ahead and use the commands provided by the PowerShellGet module.

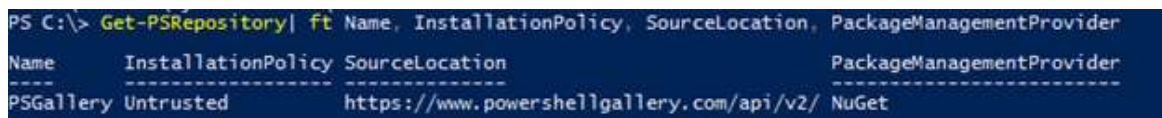
Repositories

Before running and installing modules, let's just look at where we, or I should say the system looks for modules. These modules are stored in repositories outside of your organization or home.

Managing Repositories

Listing the configured repositories, run the Get-PSRepository cmdlet. To demonstrate a little more, let's run:

```
Get-PSRepository | ft Name, InstallationPolicy, SourceLocation, PackageManagementProvider
```



```
PS C:\> Get-PSRepository | ft Name, InstallationPolicy, SourceLocation, PackageManagementProvider
Name      InstallationPolicy SourceLocation      PackageManagementProvider
-----
PSGallery Untrusted          https://www.powershellgallery.com/api/v2/ NuGet
```

Figure 2 –Get-PSRepository

Notice that we are using NuGet behind the scenes and connecting to www.PowershellGallery.com.

Jump over to this blog to see how you could create your own NuGet Respository. <https://blogs.msdn.microsoft.com/powershell/2014/05/20/setting-up-an-internal-powershellget-repository/>

Finding Modules in the Repositories

Now that we know which repositories are configured and where the system will look, how do we know what's available at each the repositories.

We can surf around on the PowerShellGallery.com website, but let's use PowerShell. Introducing the Find-Module cmdlet. What a tricky name!

This can take a wildcard characters in the name that are similar to the -like search (Ex. “*”). Here is an example of finding all the Desired State Configuration (DSC) modules for System Center.

Find-Module xSC*

```
PS C:\> Find-Module xSC*
-----
Version      Name              Repository        Description
-----
1.2.4.0      xSCVMM            PSGallery         Module with DSC Resources for deployment and con...
1.5.0.0      xSCSMA            PSGallery         Module with DSC Resources for deployment and con...
1.2.0.0      xSCDPM            PSGallery         Module with DSC Resources for deployment and con...
1.3.1.0      xSCSPF            PSGallery         Module with DSC Resources for deployment and con...
1.3.3.0      xSCOM             PSGallery         Module with DSC Resources for deployment and con...
1.3.0.0      xSCSR             PSGallery         Module with DSC Resources for deployment and con...
1.0.0.6      xScheduledTask    PSGallery         resource for ScheduledTask
```

Figure 2 –Find-Module

There are the -RequiredVersion or the -MaximumVersion and -MinimumVersion parameters. Using these parameters will help the Find-Module cmdlet install the correct one you’re looking for. These are actually a couple good options to think about sometimes, especially when you’re working in an environment with strict change controls and you need to keep a very specific version of a module.

Before we go ahead and use these Maximum and MinimumVersion parameters, let’s use another parameter to show us all the version that are available to us. Unfortunately, all 3 of these parameters only work on a single module name. So let’s take a look at Azure module.

Find-Module -Name Azure -AllVersions

```
PS C:\> Find-Module -Name Azure -AllVersions
-----
Version      Name              Repository        Description
-----
3.3.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
3.1.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
3.0.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
2.1.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
2.0.1        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.6.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.5.1        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.5.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.4.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.3.2        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.3.1        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.3.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.2.3        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.0.4.2      Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.0.4.1      Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.0.4        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.0.3        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.0.2        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.0.1        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
1.0.0        Azure             PSGallery         Microsoft Azure PowerShell - Service Management
0.9.11      Azure             PSGallery         Microsoft Azure PowerShell - Service Management
0.9.10      Azure             PSGallery         Microsoft Azure PowerShell - Service Management
```

Figure 3 – Finding Azure Modules

Now that we know what version are available, let’s try the MaximumVersion and MinimumVersion parameters.

```
Find-Module -Name Azure -MinimumVersion 1.3.0 -MaximumVersion 2.0.0
```

```
PS C:\> Find-Module -Name Azure -MinimumVersion 1.3.0 -MaximumVersion 2.0.0
Version      Name      Repository      Description
-----
1.6.0        Azure     PSGallery       Microsoft Azure PowerShell - Service Management
```

Figure 3 – Finding Azure Modules with minimum and maximum versions

Now let's try the RequiredVersion Parameter. This requires an explicit match to be found, otherwise an object is not returned.

```
PS C:\> Find-Module -Name Azure -RequiredVersion 1.0.4.2
Version      Name      Repository      Description
-----
1.0.4.2      Azure     PSGallery       Microsoft Azure PowerShell - Service Management
```

Figure 4 – Finding Azure Modules using the required parameter

Installing a Module

Now that you've found a module to install, let's install it. Of course, we already know the cmdlet is called Install-Module and yes it's pretty simple.

```
Install-Module -Name Azure -RequiredVersion 1.0.4.2
```

Let's see the installation outcome.

```
Get-Module -Name Azure -ListAvailable
```

```
PS C:\> Get-Module -Name Azure -ListAvailable
Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version      Name      ExportedCommands
-----
Script      1.0.4.2     Azure     {Disable-AzureServiceProjectRemoteDesktop, Enable-AzureSer...
```

Figure 5 – Getting the newly installed Azure Module

Updating the Module

Now that we have a version installed, let's pretend that someone just upgraded the Azure module from 1.0.4.2 to something else. Let's go ahead and update it.

```
Update-Module -Name Azure
```

Let's check the result.

```
Get-Module -Name Azure -ListAvailable
```

```
PS C:\> Get-Module -Name Azure -ListAvailable

Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version      Name                               ExportedCommands
-----
Script      3.3.0        Azure                               {Get-AzureAutomationCertificate, Get-AzureAutomationConnec...
Script      1.0.4.2      Azure                               {Disable-AzureServiceProjectRemoteDesktop, Enable-AzureSer...
```

Figure 6 – Updating Modules

Did you notice that we now have two version of the module? So what happens now? Well the Import-Module will use the latest version. If you want to use a specific version, you'll have to use the -Version parameter along with the Import-Module cmdlet.

Proxy Configuration

One of the nice things that I really love, is that these cmdlets support using a proxy. Have to log working in security conscious environments. Use the -Proxy and -ProxyCredential to get out to the Internet world!

Summary

Well I hope I've given you a little glimpse and ideas into managing your PowerShell Modules. Until next time!

Chapter 5

Use PowerShell to Validate Parameters for you

By: Allan Rafuse – Future MVP

When writing PowerShell scripts, one of the most time-consuming tasks is validating input and handling invalid input. PowerShell does have built in mechanisms to deal with this and ease these tasks so that we can focus on creating a production level script.

Even if you're new to PowerShell, you'll have used some basic PowerShell cmdlets to probably get information. Sometimes you need to pass in parameters such as the ComputerName or a filter parameter. The scripts that you create or edit can also have parameters.

See the official Microsoft documentation on the About Functions at https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_functions

And of course, this blog post covers most of what is in the Advanced Functions at https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_functions_advanced_parameters

Common PowerShell Functions – NO Validation

When creating your script, think of it essentially as a giant function. The function name in this case would of course be the script name itself. Now that we can think of it as a function, we can also take advantage of passing in parameters. Also, when we pass in the parameters we can have PowerShell validate them for us.

Stepping back a step, let's just look at a function with three parameters. Pretty basic and simple function, but there is a lot we can do with it to clean it up to make it more robust.

```
function Write-TextToScreen {
    param (
        $text,
        $ForegroundColor,
        $TimesToRepeat,
        $NoNewline
    )
    1..$TimesToRepeat | % { Write-Host -ForegroundColor $ForegroundColor -
NoNewline:$NoNewline -Object $text }
}
```

```
Write-TextToScreen "Hello world" yellow 3 $false
Write-Host "Testing Newline parameter"
```

The problem as I see it is there are some best practices lacking here. Yes, the parameters have some good names to signify what they are, but that's it. Do I pass in numbers, strings, arrays? Also, what values am I allowed to pass in? For some reason, I want to print the text out in the color of money. Is that possible? Silly as it may be, but yes, we can according to our function definition, it'll accept it!

PowerShell Features for Parameters

Problem 1

Now it's time to learn and "lock" this code down. Let's see what features PowerShell Provides. First off, we should always tell PowerShell what type of data the variables we're defining. Let's redefine the param section to include the type of data we will be storing. This ensures that whenever people call our script or function that they'll pass in the correct type of data.

```
param (  
    [string]$text,  
    [string]$ForegroundColor,  
    [int]$TimesToRepeat,  
    [boolean]$NoNewline  
)
```

Problem 2

If you look at the code a little closer, you'll see that most of the parameters are passed to the Write-Host cmdlet. If a person called our function without passing in parameters, I'll personally guarantee it will not print what you expected. What we can do here is add some default values to the function calls.

```
param (  
    [string]$text = "",  
    [string]$ForegroundColor = "white",  
    [int]$TimesToRepeat = 1,  
    [boolean]$NoNewline = $false  
)
```

Problem 3

We have some defaults set so now we can just call the function and even put them in any order if we are passing them using the parameter names. The next problem is I really don't know what color values I can use for the ForegroundColor. This problem comes up quite often! Read the documentation on what values you can pass, or you can self-document it a little and have PowerShell error check it for you at the same time! We do this by using the PowerShell feature called ValidateSet.

Let's redefine the Foregroundcolor parameter to allow only 3 colors. Here I want to highlight that you can allow anything into the validation set, even spelling errors, so be careful! Why be careful with your validation? Well I figured I would allow and define a string value called MoneyGreen. Well this is all fine and dandy and it will work on a call to the function, but when the Write-Host cmdlet tries to print to the screen using the foreground color of MoneyGreen, it will then complain! Usually too far too late in your code. Let's catch these errors before the function is called.

```
[ValidateSet("white", "Blue", "MoneyGreen")] [string]$ForegroundColor = "white",
```

Solution

After tackling the 3 problems above the script looks like:

```
function Write-TextToScreen {
    param (
        [string]$text = "",
        [ValidateSet("White", "Blue", "MoneyGreen")] [string]$ForegroundColor =
"White",
        [int]$TimesToRepeat = 1,
        [boolean]$NoNewline = $false
    )
    1..$TimesToRepeat | % { Write-Host -ForegroundColor $ForegroundColor -
NoNewline:$NoNewline -Object $text }
}

Write-TextToScreen -TimesToRepeat 1 "Test" -ForegroundColor Blue
Write-Host "Testing Newline parameter"
```

Other Parameter Validation Features

There are a few others, but I think you've gotten the idea of the blog post. Let's validate the information at the start of the script or a function call before it starts. This allows us to properly define and call functions. It allows us to focus on the actual purpose of the script, not all the details of parameter checking and error handling. Let PowerShell do that for you!

Here is a list of advanced function attributes I commonly use throughout my scripts:

```
[HelpMessage="Parameter help information on how to use it"]
[AllowNull()]
[AllowEmptyString()]
[AllowEmptyCollection]
[ValidateNotNullOrEmpty]
[ValidateNotNull]
[Mandatory=$true|$false]
[ValueFromPipeline=$true|$false]
[ParameterSetName="StringValue"]
[Alias=("alias1", "alias2")]
[ValidateLength(1,15)]
[ValidatePattern("SERVER\d{1,9}$")]
[ValidateRange(0,10)]
[ValidateLength]
[ValidateCount]
[ValidateScript]
```

Chapter 6

PowerShell – My Way to Prepare a SQL Server VM

By: Allan Rafuse – Future MVP

As I get called on a lot of to do SQL Server installations, I've come up with what I've found works best for me. Every location has different infrastructure, security, networks and their way of doing things. Since I'm the one doing the installation, and I know I'll get called back in the future at some point to upgrade, troubleshoot or manage the SQL Server environment, I like to have a set of standards. Documentation, I actually do enjoy writing it (yes I may be sick), but having a self-documenting PowerShell script is even better!

I have written a PowerShell script with a few functions to help prepare the SQL Server VM or SQL Server cluster node VMs. I'll copy this script to the management server, edit a template file that contains the variables. This template file is the only file I have to change from install to install. Microsoft does have desired state configurations (DSC) that are wonderful. The problem I run in when I try to use them is that I spend more time tweaking them to get them to work in each environment. This Blog is to highlight some quick and easy ways to configure the virtual hardware settings and drives for the installation of SQL Server.

Here is the process I use to build a SQL Server Hyper-V Virtual Machine (VM):

Hopefully I'm lucky enough to get one of the customers standard builds. I just want a C:\ and a domain joined Windows Server.

Prepare my template file

Run a few of my PowerShell functions against using the template file

Voila, we will now have a VM with the

C:\ = OS

Q:\ = Quorum

E:\ = SQLInstanceName_DB

F:\ = SQLInstanceName_LOG

G:\ = SQLInstanceName_TEMPDB

H:\ = SQLInstanceName_TEMPLOG

Template File

Below is my template file. I copy the contents to the customers local management server or a Hyper-V server that hosts the SQL Server VM and save it as SQLInstance_Application01.ps1. I then edit the variables to fit the customers environment. User accounts, passwords, IPs etc.

It has variables that I use to do the following:

Create VHDX files for the SQL Data/Log/TempDB files and a possible cluster Quorum disk

Attach the disks to the VM(s) that will make up the standalone or clustered SQL Server

Prepare the disks on the first VM (Init, Format, Assign Drive Letter)

```
# Input Parameters
$HyperVClusterServer = "Hyperv01"
$VHDPATH = "F:\vhds\"
$ClusterName = "CLS01"
$ClusterCap = "CAS01"
$instanceName = "MSSQLSERVER"
$ClusterIP = "192.168.1.50"
$nodes = @("SQL01")

$SQLSource = "\\corp\fileserver\ISO\MSSQL\SQLServer.iso"
$svcSQLUser = "CORP\svc-sqldb"
$svcSQLUserPassword = "MyP@$w0rd"
$svcSQLAgent = "CORP\svc-sqlagent"
$svcSQLAgentPassword = "AnotherP$assw0rd"
$instancePort = "1433"
$sqlIP = "192.168.1.51"
$clusterSubnetMask = "255.255.255.0"
$clusterNetwork = "Cluster Network 1"
$SQLAdmins = "CORP\SQL ADMIN"
$instanceDir = "C:\Program Files\Microsoft SQL Server"
$userDbDir = "E:\MSSQL.$instanceName\Data"
$userDbLogDir = "F:\MSSQL.$instanceName\Log"
$tempDbDir = "G:\MSSQL.$instanceName\Data"
$tempDbLogDir = "H:\MSSQL.$instanceName\Log"

$Drives = [ordered]@{
    "$VHDPATH$instanceName\_q.vhdx" = 1GB;
    "$VHDPATH$instanceName\_e.vhdx" = 120GB;
    "$VHDPATH$instanceName\_f.vhdx" = 20GB;
    "$VHDPATH$instanceName\_g.vhdx" = 30GB;
    "$VHDPATH$instanceName\_h.vhdx" = 30GB;
}
$DiskLabels = @{
    "Q" = "Quorum";
    "E" = "$instanceName\_DB";
    "F" = "$instanceName\_LOG";
    "G" = "$instanceName\_TEMPDB";
    "H" = "$instanceName\_TEMPLOG";
}

#The functions below will be imported from library-PrepareSQLServer.ps1
& . C:\SQLInstall\library-PrepareSQLServer.ps1
$result1 = Create-SQLDisks -ComputerName $HyperVClusterServer
$result2 = AttachDisks -Nodes $nodes -HyperVCluster $HyperVClusterServer
$result3 = PrepareDisks -Nodes $nodes -FirstDiskIsQuorum $true
```

Notice at the bottom of the template file that I include library-PrepareSQLServer.ps1. This is the workhorse that holds all the generic code to act on the variables above.

A look at library-PrepareSQLServer.ps1

The following code I copy into a file and also save this on the customer's management server in C:\SQLInstall\library-PrepareSQLServer.ps1. This is then imported into each template file I use.

```
Function Create-SQLDisks {
    # Create VHD Files
    param (
        [string]$ComputerName = "localhost"
    )

    foreach ($drive in $drives.keys) {
        write-host "Creating $($drives[$drive]) file: $drive"
        $rc = New-VHD -ComputerName $ComputerName -Path $drive -Dynamic -
SizeBytes $drives[$drive]
    }
}

Function AttachDisks {
    param (
        [Array]$nodes,
        [String]$HyperVCluster,
        [Boolean]$EnableSharedVhd = $true
    )

    # Attach the VHDX Files
    # This requires us to run it on the Hyper-V Host where the VM is running
    $clusterNodes = Get-ClusterNode -Cluster $HyperVCluster -ErrorAction
silentlyContinue
    if (($clusterNodes.Count -le 0) -or ($EnableSharedVhd -eq $false)) {
        $SupportPersistentReservations = $false
    }
    foreach ($VMName in $nodes) {
        $ControllerLocation = 1
        if ($clusterNodes) {
            $ComputerName = (Get-VM -ComputerName $clusternodes -VMName $VMName
-ErrorAction SilentlyContinue).ComputerName
            If ($ComputerName) {
                Write-Host -ForegroundColor Yellow "[$VMName] Located on Host
$ComputerName"
            } else {
                Write-Host -ForegroundColor Red "Could not find $VMName on any
Host"
                Return $false
            }
        } else {
            $ComputerName = $HyperVCluster
        }
        foreach ($drive in $drives.keys) {
            write-host "[$VMName] Connecting Disk on $ControllerLocation. Shared
VHDX: $SupportPersistentReservations"
            write-host "`t$drive"
            Add-VMHardDiskDrive -ComputerName $ComputerName -VMName $VMName -
ControllerType SCSI -ControllerNumber 0 -ControllerLocation $ControllerLocation
-Path $drive -SupportPersistentReservations:$SupportPersistentReservations

```

```

        $ControllerLocation++
    }
    Start-VM -ComputerName $ComputerName -VMName $VMName -ErrorAction
    SilentlyContinue -WarningAction SilentlyContinue
    }
    return $true
}

Function PrepareDisks {
    param (
        [Array]$nodes,
        [boolean]$FirstDiskIsQuorum = $true
    )
    # Prepare the disks and format them
    $node1 = $nodes[0]

    # wait for Machine 1 to come online so that we can prep the disks
    if (!(Test-WSMan -ComputerName $node1 -ErrorAction SilentlyContinue)) {
        Write-Host "$node1 Could not connect using WinRM. Please check
    configuration on $node1."
        return $false
    }

    Invoke-Command -ComputerName $node1 -ArgumentList @($FirstDiskIsQuorum,
    $DiskLabels) -ScriptBlock {
        param (
            $FirstDiskIsQuorum,
            $DiskLabels
        )

        Write-Host "Setting disks to an online state"
        Get-Disk | where {$_.OperationalStatus -eq "Offline"} | Set-Disk -
    IsOffline $False
        Write-Host "Setting disks to a read/write state"
        Get-Disk | where {$_.IsReadOnly -eq $true} | Set-Disk -IsReadOnly $False
        Write-Host "Clearing disks and data"
        Get-Disk | where {$_.Number -gt 0} | Clear-Disk -RemoveData -
    Confirm:$false -ErrorAction SilentlyContinue
        Write-Host "Initializing disks"
        Get-Disk | where {$_.Number -gt 0} | Initialize-Disk -PartitionStyle GPT
        -ErrorAction SilentlyContinue

        $DiskNumber = 1
        $disks = Get-Disk | where {$_.NumberOfPartitions -eq 1}
        $disks | % {
            if ($FirstDiskIsQuorum) {
                if ($DiskNumber -eq 1) {
                    $DriveLetter = "Q"
                    $AllocationUnitSize = 4096
                } else {
                    $DriveLetter = ([char](67 + $_.Number)).ToString()
                    $AllocationUnitSize = 65536
                }
            } else {
                $DriveLetter = ([char](68 + $_.Number)).ToString()
                $AllocationUnitSize = 65536
            }
        }
        Write-Host "[ $($DriveLetter):] Creating
    $($DiskLabels[$DriveLetter])"
        $Volume = New-Partition -DiskNumber $_.Number -DriveLetter
    $DriveLetter -UseMaximumSize -ErrorAction SilentlyContinue | Format-Volume -
    FileSystem NTFS -NewFileSystemLabel $DiskLabels[$DriveLetter] -
    AllocationUnitSize $AllocationUnitSize -Confirm:$false
    }
}

```



```

        For ($counter=1; $counter -lt $nodes.count) {
            Write-Host "`t[$($DriveLetter):] Setting drive letter on machine
            $($nodes[$counter])"
            Invoke-Command -ComputerName $nodes[$counter] Set-Partition -
            NewDriveLetter $DriveLetter -DiskNumber $DiskNumber
        }
        $DiskNumber++
    }
}

```

Running the script

```
PS C:\WINDOWS\system32> Create-SQLDisks -ComputerName $HyperVClusterServer
```

```
Creating 1073741824 file: F:\vhds\MSSQLSERVER_q.vhdx
```

```
Creating 128849018880 file: F:\vhds\MSSQLSERVER_e.vhdx
```

```
Creating 21474836480 file: F:\vhds\MSSQLSERVER_f.vhdx
```

```
Creating 32212254720 file: F:\vhds\MSSQLSERVER_g.vhdx
```

```
Creating 32212254720 file: F:\vhds\MSSQLSERVER_h.vhdx
```

```
PS C:\WINDOWS\system32> $result2 = AttachDisks -Nodes $nodes -HyperVCluster
$HyperVClusterServer
```

```
[SQL01] Connecting Disk on 1. Shared VHDX: False
```

```
    F:\vhds\MSSQLSERVER_q.vhdx
```

```
[SQL01] Connecting Disk on 2. Shared VHDX: False
```

```
    F:\vhds\MSSQLSERVER_e.vhdx
```

```
[SQL01] Connecting Disk on 3. Shared VHDX: False
```

```
    F:\vhds\MSSQLSERVER_f.vhdx
```

```
[SQL01] Connecting Disk on 4. Shared VHDX: False
```

```
    F:\vhds\MSSQLSERVER_g.vhdx
```

```
[SQL01] Connecting Disk on 5. Shared VHDX: False
```

```
    F:\vhds\MSSQLSERVER_h.vhdx
```

```
PS C:\WINDOWS\system32> $result3 = PrepareDisks -Nodes $nodes -FirstDiskIsQuorum $true
```

Setting disks to an online state

Setting disks to a read/write state

Clearing disks and data

Initializing disks

[Q:] Creating Quorum

[E:] Creating MSSQLSERVER_DB

[F:] Creating MSSQLSERVER_LOG

[G:] Creating MSSQLSERVER_TEMPDB

[H:] Creating MSSQLSERVER_TEMPLOG uu

Chapter 7

Building Gold VHDx images with PowerShell

By: Dave Kawula– MVP

I have recent been asked by one of my customers to help break down the functions of my Big_Demo.PS1 PowerShell Script.

It can be downloaded from www.github.com/dkawula

I figured that we would start with the basics.

What is the purpose of the Big_Demo script? I use it to build out lab environments for a variety of purposes. As we all know one of the key elements of building a lab is having a nice Gold Base VHDx. I often work with Beta Releases of Operating systems from Microsoft and often have to change the labs based on the current release of an operating system.

Big_Demo has some functions in it that will help you build out Gold Base VHDx files from an ISO file.

Before we get started we need to setup a few basic things in the Folder Structure:

- Download a copy of the Windows Server 2016 ISO and place it in a working folder.
 - In our example, we will use: c:\clusterstorage\volume1\dcbuild1

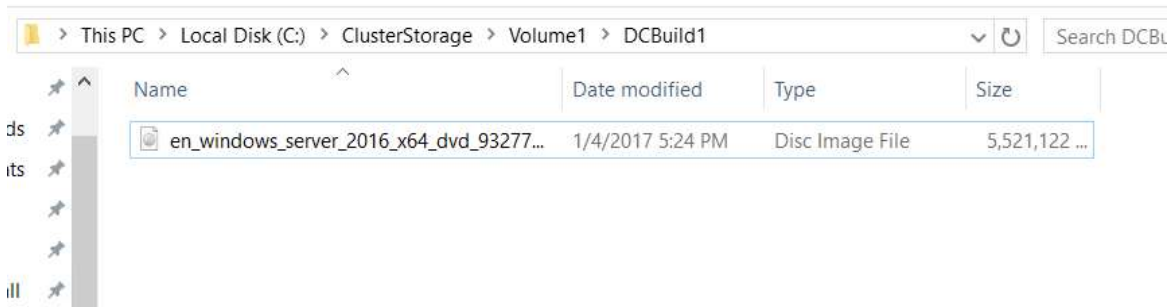


Figure 7 – Downloaded Windows Server 2016 ISO in a working directory

Define the Variables

We start off by defining a few variables that will be used to build our Gold Base VHDx's.

```
$BaseVHDPath = 'C:\ClusterStorage\Volume1\DCBuild1'
$ServerISO =
'C:\ClusterStorage\Volume1\DCBuild1\en_windows_server_2016_x64_dvd_9327751.iso'
$workingdir = 'C:\ClusterStorage\Volume1\DCBuild1'
```

Build \$unattendSource

Next, we want to Build an Unattend.xml file that will be used injected into the Base Images during the creation process.

You will want to change the <ProductKey> for use in your own lab.

```
$unattendSource = [xml]@"
<?xml version="1.0" encoding="utf-8"?>
<unattend xmlns="urn:schemas-microsoft-com:unattend">
  <servicing></servicing>
  <settings pass="specialize">
    <component name="Microsoft-Windows-Shell-Setup"
processorArchitecture="amd64" publicKeyToken="31bf3856ad364e35"
language="neutral" versionScope="nonSxS"
xmlns:wcm="http://schemas.microsoft.com/WMIConfig/2002/State"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <ComputerName>*</ComputerName>
      <ProductKey>xxxxx-xxxxx-xxxxx-xxxxx</ProductKey>
      <RegisteredOrganization>Organization</RegisteredOrganization>
      <RegisteredOwner>Owner</RegisteredOwner>
      <TimeZone>TZ</TimeZone>
    </component>
  </settings>
  <settings pass="oobeSystem">
```

```

    <component name="Microsoft-windows-Shell-Setup"
processorArchitecture="amd64" publicKeyToken="31bf3856ad364e35"
language="neutral" versionScope="nonSxS"
xmlns:wcm="http://schemas.microsoft.com/WMIConfig/2002/State"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <OOBE>
        <HideEULAPage>true</HideEULAPage>
        <HideLocalAccountScreen>true</HideLocalAccountScreen>
        <HideWirelessSetupInOOBE>true</HideWirelessSetupInOOBE>
        <NetworkLocation>Work</NetworkLocation>
        <ProtectYourPC>1</ProtectYourPC>
    </OOBE>
    <UserAccounts>
        <AdministratorPassword>
            <Value>password</Value>
            <PlainText>True</PlainText>
        </AdministratorPassword>
    </UserAccounts>
</component>
<component name="Microsoft-windows-International-Core"
processorArchitecture="amd64" publicKeyToken="31bf3856ad364e35"
language="neutral" versionScope="nonSxS"
xmlns:wcm="http://schemas.microsoft.com/WMIConfig/2002/State"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <InputLocale>en-us</InputLocale>
    <SystemLocale>en-us</SystemLocale>
    <UILanguage>en-us</UILanguage>
    <UILanguageFallback>en-us</UILanguageFallback>
    <UserLocale>en-us</UserLocale>
</component>
</settings>
</unattend>
"@

```

Get-UnattendChunk

The next thing we want to be able to do is modify the values of our base unattend.xml file on the fly based on parameters that we define during a script execution. To do this we use a function called Get-UnattendChunk.

```

function Get-UnattendChunk
{
    param
    (
        [string] $pass,
        [string] $component,
        [xml] $unattend
    )

    return $unattend.unattend.settings |
where-Object -Property pass -EQ -Value $pass `
|
select-Object -ExpandProperty component `
|
where-Object -Property name -EQ -Value $component

```

```
}
```

New-Unattendfile

The next step is to create an Unattend.xml file. To do this we will use the Get-UnattendChunk function that we created earlier to modify the values of the of the above Unattend file. As you can see below we will change values in the file like:

- Registered Organization
- Registered Owner
- Time Zone
- AdministratorPassword

```
function New-UnattendFile
{
    param
    (
        [string] $filePath
    )

    # Reload template - clone is necessary as PowerShell thinks this is a
    "complex" object
    $unattend = $unattendSource.Clone()

    # Customize unattend XML
    Get-UnattendChunk 'specialize' 'Microsoft-windows-Shell-Setup' $unattend |
    ForEach-Object -Process {
        $_.RegisteredOrganization = 'Azure Sea Class Covert Trial' #TR-Egg
    }
    Get-UnattendChunk 'specialize' 'Microsoft-windows-Shell-Setup' $unattend |
    ForEach-Object -Process {
        $_.RegisteredOwner = 'Thomas Rayner - @MrThomasRayner -
workingsysadmin.com' #TR-Egg
    }
    Get-UnattendChunk 'specialize' 'Microsoft-windows-Shell-Setup' $unattend |
    ForEach-Object -Process {
        $_.TimeZone = 'Pacific Standard Time'
    }
    Get-UnattendChunk 'oobeSystem' 'Microsoft-windows-Shell-Setup' $unattend |
    ForEach-Object -Process {
        $_.UserAccounts.AdministratorPassword.value = 'P@ssw0rd'
    }

    $unattend.Save($filePath)
}
```

New-BaseImage

The last function that we use is called New-BaseImage and it leverages a PowerShell Script called Convert-WindowsImage.PS1 that will be extracted from the Windows Server 2016 ISO to build our Gold Base VHDx files for both Windows Server 2016 Full GUI and Core.

This Function New-BaseImage will do the following:

- Mount the Windows Server ISO
- Copy Convert-WindowsImage.PS1 to the working directory
- Create a new Unattend.xml file in the working directory
- Check to see if the file VMServerBaseCore.vhdx exists
 - If it does not exist we continue execution and define the structure of the new vhdx
 - Once done then we execute Convert-WindowsImage.PS1 to build the Gold Image
- Step 4 repeats to build VMServerBase.vhdx which is the full UI version of Windows Server 2016.

```
Function New-BaseImage
{

    Mount-DiskImage $ServerISO
    $DVDDriveLetter = (Get-DiskImage $ServerISO | Get-Volume).DriveLetter
    Copy-Item -Path
"$($DVDDriveLetter):\NanoServer\NanoServerImageGenerator\Convert-
windowsImage.ps1" -Destination "$($WorkingDir)\Convert-windowsImage.ps1" -Force

    New-UnattendFile "$WorkingDir\unattend.xml"

    #Build the windows 2016 Core Base VHDx for the Lab
    if (!(Test-Path "$($BaseVHDPath)\VMServerBaseCore.vhdx"))
    {

        Set-Location $workingdir

        # Load (aka "dot-source") the Function
        . .\Convert-windowsImage.ps1
        # Prepare all the variables in advance (optional)
        $ConvertWindowsImageParam = @{
            SourcePath = $ServerISO
        }
    }
}
```

```

        RemoteDesktopEnable = $True
        Passthru             = $True
        Edition              = "ServerDataCenterCore"
        VHDFormat            = "VHDX"
        SizeBytes            = 60GB
        WorkingDirectory     = $workingdir
        VHDPath              = "$($BaseVHDPath)\VMServerBaseCore.vhdx"
        DiskLayout           = 'UEFI'
        UnattendPath         = "$($workingdir)\unattend.xml"
    }

    $VHDx = Convert-WindowsImage @ConvertWindowsImageParam
}

#Build the windows 2016 Full UI Base VHDx for the Lab
if (!(Test-Path "$($BaseVHDPath)\VMServerBase.vhdx"))
{

    Set-Location $workingdir

    # Load (aka "dot-source") the Function
    . .\Convert-WindowsImage.ps1
    # Prepare all the variables in advance (optional)
    $ConvertWindowsImageParam = @{
        SourcePath              = $ServerISO
        RemoteDesktopEnable    = $True
        Passthru                 = $True
        Edition                  = "ServerDataCenter"
        VHDFormat                = "VHDX"
        SizeBytes                = 60GB
        WorkingDirectory        = $workingdir
        VHDPath                  = "$($BaseVHDPath)\VMServerBase.vhdx"
        DiskLayout               = 'UEFI'
        UnattendPath            = "$($workingdir)\unattend.xml"
    }

    $VHDx = Convert-WindowsImage @ConvertWindowsImageParam
}

Dismount-DiskImage $ServerISO
}

```

The finished result will look like this:

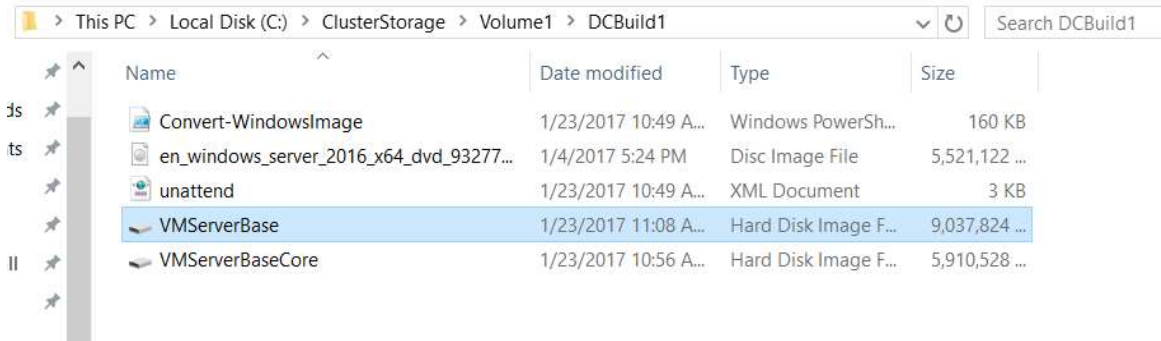


Figure 8 – Base Gold VHDx images created with PowerShell

Now you are ready to go build your lab on Hyper-V.

Chapter 8

Using Try Catch to help with Decision Making in a Script

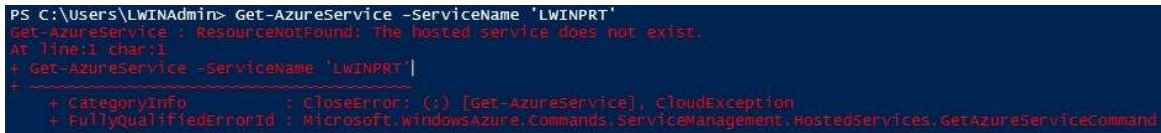
By: Will Anderson – MVP

Recently, while working on my scripts for rolling out server deployments in Azure, I came across an interesting issue with a cmdlet throwing a terminating error when I wasn't expecting one.

I was attempting to use the Get-AzureService cmdlet to verify if the cloud service that I specified already existed or not. It was necessary to check its existence in case VMs had already been deployed to the service and we were adding machines to the pool. If it didn't exist, I would add script logic to create the cloud service before deploying the VMs. So when I execute:

```
Get-AzureService -ServiceName 'LWINPRT'
```

Returns with the following terminating error:



```
PS C:\Users\LWINAdmin> Get-AzureService -ServiceName 'LWINPRT'  
Get-AzureService : ResourceNotFound: The hosted service does not exist.  
At line:1 char:1  
+ Get-AzureService -ServiceName 'LWINPRT'  
+ ~~~~~  
+ CategoryInfo          : CloseError: (:) [Get-AzureService], CloudException  
+ FullyQualifiedErrorId : Microsoft.WindowsAzure.Commands.ServiceManagement.HostedServices.GetAzureServiceCommand
```

Figure 9 – Error with Get-AzureService

Now, I expected the service to not be there, because I haven't created it, but I didn't expect the cmdlet to terminate in a way that would stop the rest of the script from running. Typically, when using a command to look for something, it doesn't throw an error if it can't find it. For example, when I look to see if a VM exists in the service:

```
Get-AzureVM -ServiceName 'LWINPRT' -Name 'LWINPRT01'
```

I get the following return:

```
PS C:\Users\LWINAdmin> Get-AzureVM -ServiceName 'LWINPRT' -Name 'LWINPRT01'  
WARNING: No deployment found in service: 'LWINPRT'.  
PS C:\Users\LWINAdmin>
```

Figure 10 – Retrieving azure service using Get-AzureVM

While the error wasn't expected, it's certainly not a show-stopper. We just have to rethink our approach. So instead of a ForEach statement looking for a null-value, why don't we instead look at leveraging Try-Catch?

The Try-Catch-Finally blocks are what allows you to catch .NET exception errors in PowerShell, and provide you with a means to alert the user and take a corrective action if needed. You can read about them here, or there's an exceptional article by Ashley McGlone on using it (<http://bit.ly/2jUU0KO>). So, we'll go ahead and set this up to test.

```
Try {  
    Get-AzureService -ServiceName 'LWINPRT' -ErrorAction Stop  
}#EndTry  
  
Catch [System.Exception]  
{  
    Write-Host "An error occurred"  
}#EndCatch
```


And we execute...

```
PS C:\Users\LWINAdmin> Try {  
    Get-AzureService -ServiceName 'LWINPRT' -ErrorAction Stop  
}#EndTry  
  
Catch [System.Exception]  
{  
    Write-Host "An error occurred"  
}#EndCatch  
An error occurred  
PS C:\Users\LWINAdmin>
```

Figure 11 – Testing Try-Catch

And we get a return! But I don't want an error in this case. What I want is to create the cloud service if it doesn't exist. so let's do this instead:

```
Try {  
  Get-AzureService -ServiceName 'LWINPRT' -ErrorAction Stop  
}#EndTry  
  
Catch [System.Exception]  
{  
  New-AzureService 'LWINPRT' -Location "West US"  
}#EndCatch
```



The screenshot shows a PowerShell terminal window with a dark blue background. The prompt is 'PS C:\Users\LWINAdmin>'. The script being executed is a try-catch block. The output shows a table with three columns: OperationDescription, OperationId, and OperationStatus. The first row of data shows 'New-AzureService' as the description, a long GUID as the operation ID, and 'Succeeded' as the status.

```
PS C:\Users\LWINAdmin> Try {  
  Get-AzureService -ServiceName 'LWINPRT' -ErrorAction Stop  
}#EndTry  
  
catch [System.Exception]  
{  
  New-AzureService 'LWINPRT' -Location "West US"  
}#Endcatch
```

OperationDescription	OperationId	OperationStatus
New-AzureService	36ed5d6b-bb07-8d28-956f-55e1a95437e6	Succeeded

Figure 12 – Using Try-Catch

And now we get the service created. And we can now see it in our Azure console:

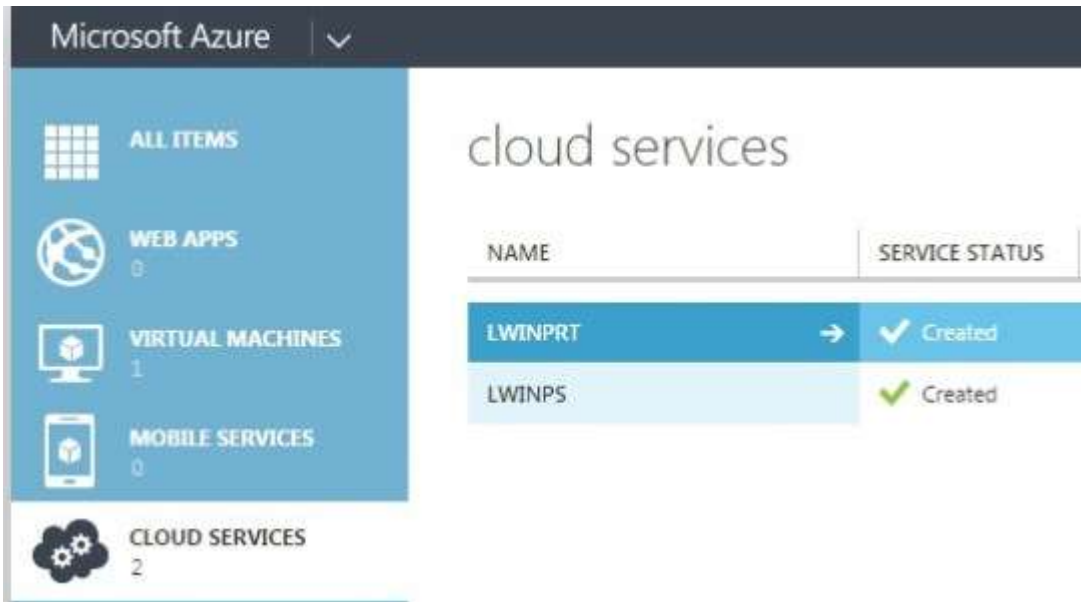


Figure 13 – Showing the newly installed Service in Azure

Now we can use this Try block to check if a cloud service exists or not, knowing that if it can't find the cloud service it will throw a terminating error. And when it does, we can use the Catch block to create the existing service. Decision made.

Chapter 9

Invoking your SCCM Client Remotely with PowerShell

By: Will Anderson – MVP

I actually wrote about this some time ago when I first started PowerShell'ing. But I've since improved my script, and I'm sharing it with you now.

Sometimes we need to make a change to our CM clients', or the environment that they reside in that requires us to trigger an action on the CM client itself. Be it pull in an updated hardware inventory, force the machine to run a Software Updates Scan, or a Machine Policy to get a machine to detect a new software deployment; it can be a painful experience for an administrator to have to go from machine to machine to run an action.

There are the right-click tools for those that have access to the console, but not everyone does, and not every company will allow you to install them. Wouldn't it be nice to have a function that you could include in a module you've built, that has the ability to invoke a CM client? Well, now I have something to offer you:

```
Function Invoke-CMClient{
<# .SYNOPSIS Invoke commands remotely on an SCCM Client for a system or systems.
.DESCRIPTION This function allows you to remotely trigger some of the more
common actions that you would find on the local Configuration Manager console.
.PARAMETER -ComputerName <string[]> Specifies the target computer for the
management operation.
    Enter a fully qualified domain name, a NetBIOS name, or an IP address. When
the remote computer is in a different domain than the local computer,
the fully qualified domain name is required. This command defaults to
localhost.
.PARAMETER -Action Specifies the action to be taken on the SCCM Client. The
available actions are as follows:
    HardwareInv - Runs a Hardware Inventory Cycle on the target machine.
    SoftwareInv - Runs a Software Inventory Cycle on the target machine.
    UpdateScan - Runs a Software Updates Scan Cycle on the target machine.
    MachinePol - Runs a Machine Policy Retrieval and Evaluation Cycle on the
target machine.
    UserPolicy - Runs a User Policy Retrieval and Evaluation Cycle on the target
machine.
    FileCollect - Runs a File Collection Cycle on the target machine.
.INPUTS You can pipe a computer name to Invoke-CMClient
.EXAMPLE Invoke-CMClientAction -ComputerName server01 -Action HardwareInv
```

The above command will invoke the Configuration Manager Client's Hardware Inventory Cycle on the targeted computer.

```
The return will look like the following: __GENUS : 1 __CLASS : __PARAMETERS
__SUPERCLASS : __DYNASTY : __PARAMETERS __RELPATH : __PARAMETERS
__PROPERTY_COUNT : 1
__DERIVATION : {} __SERVER : server01 __NAMESPACE : ROOT\ccm __PATH :
\\server01\ROOT\ccm:\__PARAMETERS ReturnValue : PSComputerName : server01
.NOTES Created by Will Anderson.
http://lastwordinnerd.com/category/posts/powershell-scripting/ This script is
provided AS IS without warranty of any kind. #>
PARAM(
[Parameter(Mandatory=$True,ValueFromPipeline=$True,ValueFromPipelineByPropertyNa
me=$True)]
[string[]]$ComputerName = $env:COMPUTERNAME,
[Parameter(Mandatory=$True)]
[ValidateSet('HardwareInv','SoftwareInv','UpdateScan','MachinePol','UserPolicy',
'DiscoveryInv','FileCollect')]
[string]$Action
)#Close Param
#$Action...actions...actions...
SWITCH ($action) {
'HardwareInv' {$action = "{00000000-0000-0000-0000-000000000001}"}
'SoftwareInv' {$action = "{00000000-0000-0000-0000-000000000002}"}
'UpdateScan' {$action = "{00000000-0000-0000-0000-0000000000113}"}
'MachinePol' {$action = "{00000000-0000-0000-0000-000000000021}"}
'UserPolicy' {$action = "{00000000-0000-0000-0000-000000000027}"}
'FileCollect' {$action = "{00000000-0000-0000-0000-000000000010}"}
} #switch
FOREACH ($Computer in $ComputerName){
if ($PSCmdlet.ShouldProcess("$action $computer")) {
Invoke-WmiMethod -ComputerName $Computer -Namespace root\CCM -Class SMS_Client -
Name TriggersSchedule -ArgumentList "$_action"
}#if
}#End FOREACH Statement
```

This function allows you to invoke an SCCM Client action remotely on a number of computers simultaneously. I've included the actions most commonly used from my perspective, but you can add more if you like. You can get a good list of them here, or just explore the SMS_Client class for a full list.

Chapter 10

Use PowerShell to Parse RSS Feeds

By: Ed Wilson – The Scripting Guy & Will Anderson - MVP

Last week as I took my seat on a connecting flight in New York from beautiful Charlotte, North Carolina, my thoughts drifted to the wonderful experiences and memories that I took with me from the Windows PowerShell Summit. One thought in particular crossed my mind. It was a challenge posed to me by a member of the Windows PowerShell team.

“Will,” he said, “One of the challenges our team faces is juggling back and forth between working on new releases and managing the feedback we receive from the community.”

My interest was piqued as he continued, “A couple of times a week, we have to review all of the feedback on Connect. Depending on what we’re working on, we usually triage the bug reports or the feature requests, and we’re usually looking for the ones with the most votes so we know which ones the community is asking for most. We don’t have time to whip up a new tool, so do you think you’d be up to the challenge of creating one?”

Challenge accepted!

Initially, I attempted to leverage `Invoke-WebRequest` against a search URL on the Connect website. But parsing through the data, which consisted of many pages, would have been too cumbersome, and it would not necessarily retrieve the data I wanted.

Instead, I decided to request the Most Recent Requests RSS feed to retrieve the data I wanted. So, I start by invoking the web request against that page and output it to an XML page to parse through:

```
Invoke-WebRequest -Uri  
'https://connect.microsoft.com/rss/99/RecentFeedbackForConnection.xml' -OutFile  
C:\scripts\ConnectFeed.xml
```

Now, I can reference the file I created and start diving into the XML for our data:

```
$Content = Get-Content C:\scripts\ConnectFeed.xml  
$Feed = $Content.rss.channel
```

Now that we have something to look at, let’s gather some data. We’ll start by getting the last time the request was updated, the description, category, author, and of course, the link so the Windows PowerShell team can go straight to the article when they’re ready to work on it.

Of course, the data we're looking for is in multiple properties called **Item**, so we're going to have our script go through them recursively to pull the data we want. Note that I'm stating the **\$msg.updated** object is going to be designated as a **DateTime** object. This is to make sure that we can do some cool filtering by date and time later.

```
ForEach ($msg in $Feed.Item){
[PSCustomObject]@{
'LastUpdated' = [datetime]$msg.updated
'Description' = $msg.description
'Category' = $msg.category
'Author' = $msg.author
'Link' = $msg.link
}#EndPSCustomObject
}#EndForEach
```

Now we get the following return:

```
'LastUpdated' = [datetime]$msg.updated
'Description' = $msg.description
'Category' = $msg.category
'Author' = $msg.author
'Link' = $msg.link
}#EndPSCustomObject

LastUpdated : 3/27/2015 7:14:33 AM
Description  : If you try to pass an embedded CIM instance to a resource using Invoke-DscResource, you'll
              get either an error that the parameter cannot take a string or that you are passing an
              unrecognized command.

              For example:
              If I want to use, for example, the xwebAdministration module's xwebsite resource (which has
              a BindingInfo property that is an embedded CIM instance), how can I pass that with
              Invoke-DscResource? It won't cast a string and if you don't make it a string, it tries to
              interpret MSFT_xWebBi...<BR><BR>Status: Active, 2 Up-Votes, 0 Down-Votes, 0 validations, 0
              workarounds, 0 comments, feedback id: 1207374

Category    : Bug
Author      : Steve
Link        : http://connect.microsoft.com/PowerShell/feedback/details/1207374/wmf-5-invoke-dscresource-can
              not-take-embedded-cim-instances-as-parameters
```

Figure 14 – Using PowerShell to Parse RSS Feeds

So far, that looks good. But looking at the description, I see some data that I think would be really useful as objects on their own. I'm referring mostly to the Up-Votes and Down-Votes, but I think we could break out pretty much everything after that break.

So, let's split off that section of the string into useable chunks and get rid of any oddball spaces in the line:

```
(( $msg.description ).split('<BR>') ).split(',').trim()
```

I execute again and we get this:

```
Status: Active
2 Up-Votes
0 Down-Votes
0 validations
0 workarounds
0 comments
feedback id: 1207374
```

```
PS C:\windows\system32>
```

Figure 15 – Updated \$msg.description

Aha! Now we have all of this magnificent data, but it's string data, and not very sortable. So I'm going to create some rules to fix that issue. Let's start by focusing on those last seven string objects I created overall, and specifically target the Up-Votes and Down-Votes for our test. Then I'll add them to our **PSCustomObject**:

```
ForEach ($msg in $Feed.Item){
  $ParseData = (($msg.description).split('<BR>')).split(',').trim() | select-
  Object -Last 7
  ForEach ($Datum in $ParseData){
    If ($Datum -like "*up*"){[int]$Upvote = ($Datum).split(' ') | select-Object -
    First 1}#EndIf
    If ($Datum -like "*down*"){[int]$Downvote = ($Datum).split(' ') | select-Object
    -First 1}#EndIf
  }#EndForEach
  [PSCustomObject]@{
    'LastUpdated' = [datetime]$msg.updated
    'Description' = $msg.description
    'Category' = $msg.category
    'Author' = $msg.author
    'Link' = $msg.link
    'UpVotes' = $Upvote
    'DownVotes' = $Downvote
  }#EndPSCustomObject
}
```

When I execute this, we get the following:

```

LastUpdated : 3/27/2015 7:14:33 AM
Description : If you try to pass an embedded CIM instance to a resource using Invoke-DscResource, you'll
              get either an error that the parameter cannot take a string or that you are passing an
              unrecognized command.

              For example:
              If I want to use, for example, the xwebAdministration module's xwebsite resource (which has
              a BindingInfo property that is a embedded CIM instance), how can I pass that with
              Invoke-DscResource? It won't cast a string and if you don't make it a string, it tries to
              interpret MSFT_xWebBi...<BR><BR>Status: Active, 2 Up-Votes, 0 Down-Votes, 0 validations, 0
              workarounds, 0 comments, feedback id: 1207374

Category    : Bug
Author      : Steve
Link        : http://connect.microsoft.com/PowerShell/feedback/details/1207374/wmf-5-invoke-dscresource-can
              not-take-embedded-cim-instances-as-parameters
UpVotes     : 2
DownVotes   : 0

PS C:\windows\system32>

```

Figure 16 – Updating the script

Now we're cooking with gas! Let's go ahead and add the rest of our objects to the mix. I'll create a rule for each object in the mix:

```

ForEach ($msg in $Feed.Item){
  $ParseData = (($msg.description).split('<BR>')).split(',').trim() | select-
Object -Last 7
  ForEach ($Datum in $ParseData){
    If ($Datum -like "*up*"){[int]$Upvote = ($Datum).split(' ') | select-Object
-First 1}#EndIf
    If ($Datum -like "*down*"){[int]$Downvote = ($Datum).split(' ') | select-
Object -First 1}#EndIf
    If ($Datum -like "*validations*"){[int]$Validation = ($Datum).split(' ') |
select-Object -First 1}#EndIf
    If ($Datum -like "*workarounds*"){[int]$Workaround = ($Datum).split(' ') |
select-Object -First 1}#EndIf
    If ($Datum -like "*comments*"){[int]$Comment = ($Datum).split(' ') | select-
Object -First 1}#EndIf
    If ($Datum -like "*feedback*"){[int]$FeedbackID = ($Datum).split(' ') |
select-Object -Last 1}#EndIf
  }#EndForEach
  [PSCustomObject]@{
    'LastUpdated' = [datetime]$msg.updated
    'Description' = $msg.description
    'Category' = $msg.category
    'Author' = $msg.author
    'Link' = $msg.link
    'UpVotes' = $Upvote
    'DownVotes' = $Downvote
    'Validations' = $Validation
    'workArounds' = $workaround
    'Comments' = $Comment
    'FeedbackID' = $FeedBackID
  }#EndPSCustomObject
}

```

Here is the result:

```
LastUpdated : 4/1/2015 8:19:08 PM
Description : Using WMF 5.0.10018.0 on Server 2012 R2 we found a few of our servers stopped downloading new configuration files and kept using
              the old settings. The LCM is setup using the following settings:

              LocalConfigurationManager
              {
                  AllowModuleOverwrite = 'True'
                  CertificateID = $LocalCertificateThumbprint
                  ConfigurationID = $ConfigurationID
                  ConfigurationModeFrequencyMins = 60
                  ConfigurationMode = 'ApplyAndAutoCorrect'
                  RebootNodeAfterUpdate = $true
                  Status: Active, 1 Up-Vote, 0 Down-Votes, 0 validations, 0 workarounds, 0 comments, feedback id:
              }

Category    : 1219332
Author      : Bug
Link        : http://connect.microsoft.com/PowerShell/feedback/details/1219332/server-doesnt-download-new-dsc-configuration-file-keeps-using-old-
              configuration-instead
UpVotes     : 1
DownVotes   : 0
Validations : 0
WorkArounds : 0
Comments    : 0
FeedbackID  : 1219332
```

Figure 17 – RSS Feeds using PowerShell

That looks a lot better—except we still have the text that we’ve sorted out left in the description. Let’s see if I can remove that.

I’ll do this by using the **IndexOf** method to find the pattern that matches the HTML line break tag (
) that separates the description from the data we parsed out for our **PSCustomObject**. Then, I’ll peel away that and anything after it from the string data that we want by using the **SubString** method:

```
$Description =
($msg.description).Substring(0,($msg.description).IndexOf('<BR>'))
```

Now I replace the **\$msg.description** in our **PSCustomObject** with our new description variable and execute the command. Here is the result:

```
FeedbackID : 1219378
Status      : Active
LastUpdated : 4/1/2015 9:11:18 PM
Description : I recently moved over to a new Window 8.1 development machine running PowerShell v4.
              Trying to use Out-GridView but it fails to run with the following error from within a powershell.exe shell:
              -----
              PS> Get-Process | out-gridview

              Out-GridView : To use the Out-GridView cmdlet, install the windows PowerShell Integrated Scripting Environment feature
              from Server Manager. (Could not load file or assembly 'Microsoft.PowerShell.GraphicalHost, Version=1.0.0.0, Culture=neu-
              tral, Pu...

Category    : Bug
Author      : m
Link        : http://connect.microsoft.com/PowerShell/feedback/details/1219378/out-gridview-failing-on-powershell-v3-windows-8-1
UpVotes     : 1
DownVotes   : 0
Validations : 0
WorkArounds : 0
Comments    : 0
```

Figure 18 – And the finished product

We have a much cleaner, more readable view. I'll wrap this up in a function to make it an easy-to-use one-liner. Now our friends on the Windows PowerShell team can easily dig into the latest bug reports like so:

```
(Get-ConnectFeedback).where({$PSitem.LastUpdated -lt (Get-Date).AddDays(-21) -  
and $PSitem.Category -eq 'Bug' }) | Sort-Object UpVotes
```

We've managed to pull down an RSS feed, parse through the data to find the information that is useful to us, and separated it into useable objects to create a one-line script for easy reading. Now the Windows PowerShell team has a little more time to build awesome new features, and we've learned another way to use Windows PowerShell as an excellent tool for gathering information on the web!

You can download the entire script from the Script Center Repository:

<https://gallery.technet.microsoft.com/scriptcenter/Parse-RSS-Feeds-With-db84ced2>

Chapter 11

How to alter the public IP Address of an Azure Virtual Machine using PowerShell

By: Will Anderson – MVP

Recently, I incorrectly configured an Azure Resource Manager virtual machine (VM) in my lab environment and needed to make some changes to the public IP settings. A brief look around the Internet came up empty, so I thought I'd share the challenge and the solution to this puzzle with you.

The Challenge

If we take a look at **Set-AzureRmPublicIpAddress**, it uses only one parameter: -**PublicIpAddress**. To actually change the properties in the object, however, we need to call them. How do we do that? It depends on the property in question, so I'll give you an example of two different properties that we're going to change on our target VM. In this example, we're going to change our public IP allocation from Dynamic to Static, and we're going to give our PublicIP a friendly name and a fully qualified domain name (FQDN).

Data formatting or using an existing example

We can look up an existing PublicIP configuration by doing the following:

```
$ExResGrp = Get-AzureRmResourceGroup -Name 'lwindsc'  
$PubIP = (Get-AzureRmPublicIpAddress -ResourceGroupName  
$ExResGrp.ResourceGroupName).where({$PSItem.Name -eq 'lwindstgtwestuspubip'})
```

And as we can see, we have a configuration to look at:

```

Name                : lwindstgtwestuspubip
ResourceGroupName   : lwindsc
Location            : westus
Id                  : /subscriptions/f2007bbf-f802-4a47-9336-cf7c6b89b378/resourceG
Etag                : w/"fa9e83f9-e160-4ecc-94eb-59a61b7bfe8f"
ResourceGuid        : 7547bc4a-2513-4435-8eaf-559716bee980
ProvisioningState    : Succeeded
Tags                :
PublicIpAllocationMethod : Dynamic
IpAddress           : Not Assigned
PublicIpAddressVersion : IPv4
IdleTimeoutInMinutes : 4
IpConfiguration     : {
  "Id": "/subscriptions/f2007bbf-f802-4a47-9336-cf7c6b89b378/
uscontrolwindstgt-nifconfig"
}
DnsSettings         : {
  "domainNameLabel": "lwindstgtwestuspubip",
  "Fqdn": "lwindstgtwestuspubip.westus.cloudapp.azure.com"
}

```

Figure 19 – Showing the configuration of an Azure Virtual Machine

The **PublicIpAllocationMethod**, where we define our configuration as Static or Dynamic, is a simple string, which is easy enough to pass along. But, if you notice, the **DnsSettings** are displayed in a hashtable. So, let's construct our changes. First, we cast our target PublicIP configuration object to a variable:

```

$TgtResGrp = Get-AzureRmResourceGroup -Name 'lwinpubip'
$PubIP = Get-AzureRmPublicIpAddress -ResourceGroupName
$TgtResGrp.ResourceGroupName

```

If we call the object, we'll see that the **PublicIpAllocationMethod** is set to **Dynamic**, and the **DnsSettings** property is null


```

Name                : lwinpubip
ResourceGroupName   : lwinpubip
Location            : westus
Id                  : /subscriptions/f2007bbf-f802-4a47-9336-cf7c6b89b378/resourcegroups/lwinpubip/prov
Etag                : W/"c287317a-3139-4d51-8aa7-858b689efed2"
ResourceGuid        : cd6194f6-8555-488d-9908-d05747281ea2
ProvisioningState    : Succeeded
Tags                :
PublicIpAllocationMethod : Dynamic
IpAddress           : 40.118.166.50
PublicIpAddressVersion : IPv4
IdleTimeoutInMinutes : 4
IpConfiguration     : {
                        "Id":
                        "/subscriptions/f2007bbf-f802-4a47-9336-cf7c6b89b378/r
                    }
DnsSettings          : null

```

Figure 20 – Validating that the IP of the Azure Virtual Machine is dynamic

Make the change

Now we call the properties that we want to modify and the values that we want to input.

```

$PubIp.PublicIpAllocationMethod = 'Static'
$PubIP.DnsSettings = @{
'DomainNameLabel' = ($TgtResGrp.ResourceGroupName + $TgtResGrp.Location +
'pubip')
'Fqdn' = ($TgtResGrp.ResourceGroupName + '.westus.cloudapp.azure.com')
}

```

If we look at our stored object, we can see the changes that we've made:

```

Name                : lwinpubip
ResourceGroupName   : lwinpubip
Location            : westus
Id                  : /subscriptions/f2007bbf-f802-4a47-9336-cf7c6b89b378/r
Etag                : W/"c287317a-3139-4d51-8aa7-858b689efed2"
ResourceGuid        : cd6194f6-8555-488d-9908-d05747281ea2
ProvisioningState    : Succeeded
Tags                :
PublicIpAllocationMethod : Static
IpAddress           : 40.118.166.50
PublicIpAddressVersion : IPv4
IdleTimeoutInMinutes : 4
IpConfiguration     : {
                        "Id":
                        "/subscriptions/f2007bbf-f802-4a47-9336-cf7c6b89b378/
                    }
DnsSettings          : {
                        "DomainNameLabel": "lwinpubipwestuspubip",
                        "Fqdn": "lwinpubip.westus.cloudapp.azure.com"
                    }

```

Figure 21 – Validating that we now have a static IP Address on the Azure Virtual Machine

Now we commit the changes to Azure by passing our object back with Set-AzureRmPublicIpAddress.

```
$PubIP | Set-AzureRmPublicIpAddress
```

And now our system is accessible remotely by a friendly name!

Chapter 12

Get the Public IP of an Azure VM with PowerShell

By: Will Anderson – MVP

Here is a quick hitter for you. I often get asked how to retrieve the public IP Address of an Azure Virtual Machine. Well the answer is simple, all you need to do is use **Get-AzureRmVm** to find the VM and pass it to **Get-AzureRmPublicIpAddress** as in the following

```
Get-AzureRmVM -ResourceGroupName 'HSG-ResourceGroup' -Name 'HSG-LinuxVM' | Get-AzureRmPublicIpAddress
```

There you have it.

Chapter 13

SCCM Local Administrator Reporting with PowerShell

By: Mick Pletcher – MVP

Here is a script that will gather a list of local administrators on a machine. The script can report the list to SCCM by writing the list to a WMI entry. It can also write the list to a text file at a specified location for admins that do not have SCCM. The text file is named <%COMPUTERNAME%>.txt. You can do both SCCM reporting and text file reporting if desired.

To implement this into SCCM, run the script once on a machine with the following command line:

```
powershell.exe -file LocalAdmins.ps1 -SCCMReporting
```

Once this has executed, do the following:

1. Open the SCCM console
2. Click on Administration
3. Click Client Settings
4. Right-click Default Client Settings
5. Left-click Properties
6. Click Hardware Inventory
7. Click Set Classes
8. Click Add
9. Click Connect
10. Enter the computer name of the system you ran the script on
11. Check Recursive
12. Check Credentials required
13. Enter the domain\username for user name
14. Enter the associated password

15. Click Connect
16. Once the list of classes appears, click on Class Name to sort the classes
17. Scroll down to find Local_Administrators and check the box to the left
18. Click OK
19. Click OK
20. Click OK
21. Now go back to the machine you ran the script on and run a hardware inventory to send the data up to SCCM. It will take a few minutes until the data appears in SCCM

The next step is to setup the script to execute through SCCM as a package. The script will need to be executed on a routine basis if you want it to be reported regularly to SCCM. As a package, the following pictures show how I have it configured in SCCM.

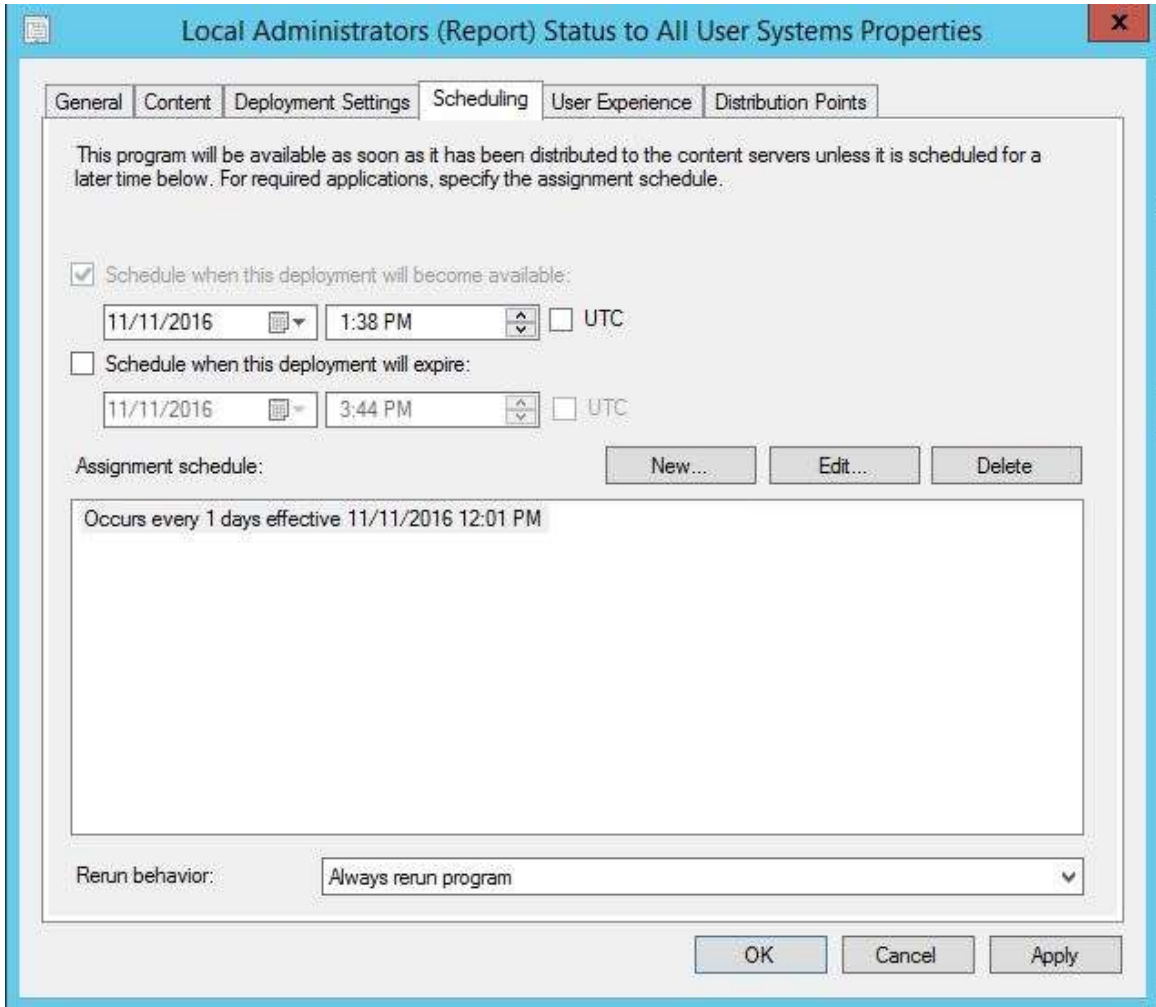


Figure 22 – Configuring the SCCM Deployment

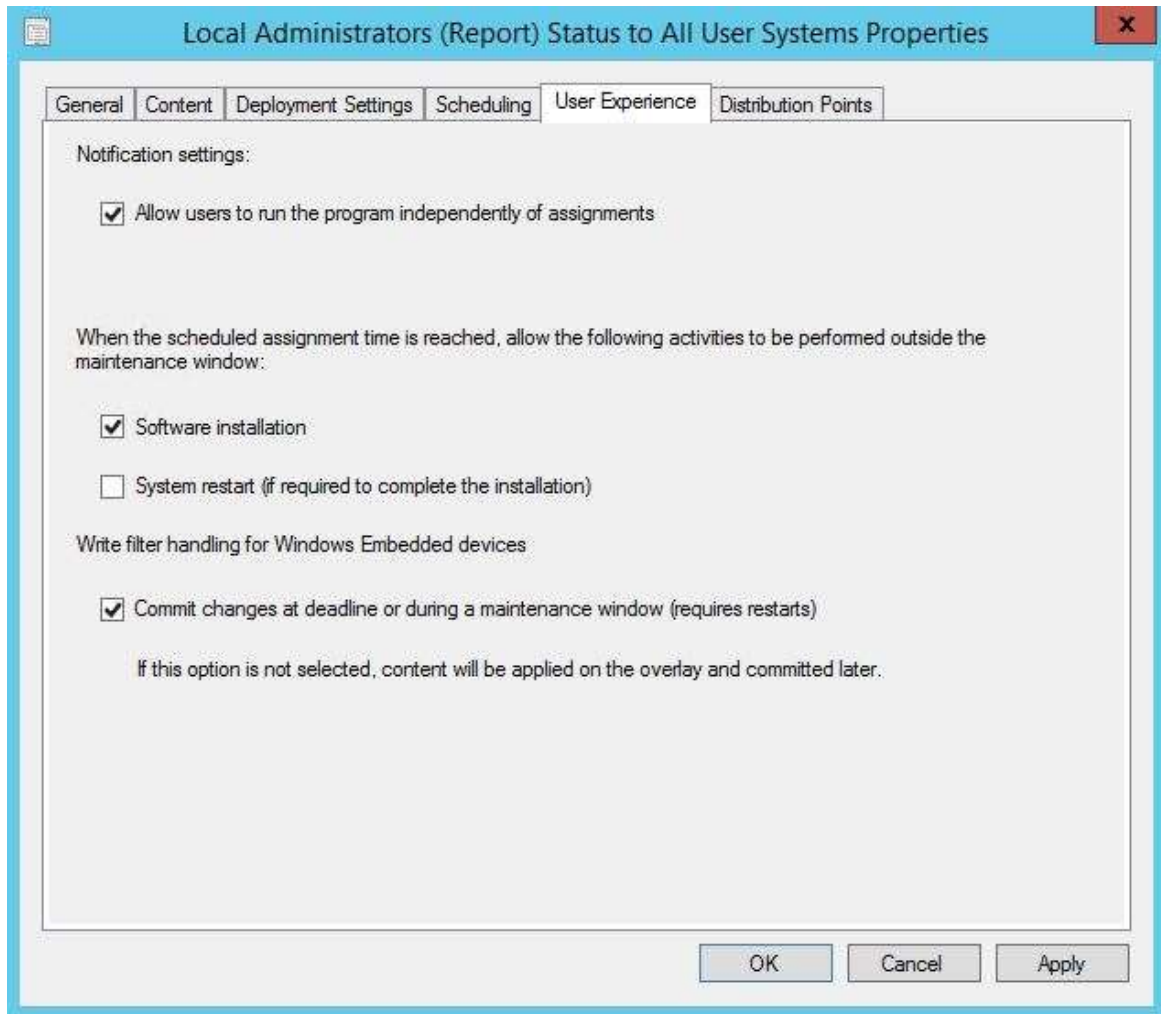


Figure 23 – Configuring the SCCM Deployment

Finally, you will want to be able to look at the results. You can create a query to show the systems that have reported users in the local administrators group. Here is the WQL I use:

```
select distinct SMS_R_System.Name,  
SMS_G_System_LOCAL_ADMINISTRATORS.Domain,  
SMS_G_System_LOCAL_ADMINISTRATORS.User from SMS_R_System inner  
join SMS_G_System_LOCAL_ADMINISTRATORS on
```

```
SMS_G_System_LOCAL_ADMINISTRATORS.ResourceID =  
SMS_R_System.ResourceId order by SMS_R_System.Name,  
SMS_G_System_LOCAL_ADMINISTRATORS.Domain,  
SMS_G_System_LOCAL_ADMINISTRATORS.User
```

LocalAdmins.ps1

You can download the script from my GitHub:

<https://github.com/MicksITBlogs/PowerShell/blob/master/LocalAdmins.ps1>

```
<#  
    .SYNOPSIS  
        Report Local administrators  
  
    .DESCRIPTION  
        Report a list of local administrators on machines to a designated  
text file, screen, and/or to SCCM via a WMI entry.  
  
    .PARAMETER MemberExclusionsFile  
        Text file containing a list of users to exclude  
  
    .PARAMETER OutputFile  
        Specifies if the output is to be written to a text file. The  
OutputFileLocation parameter also needs to be populated with the location to  
write the text file to.  
  
    .PARAMETER OutputFileLocation  
        Location where to write the output text files  
  
    .PARAMETER SCCMReporting  
        Report results to SCCM  
  
    .PARAMETER SystemExclusionsFile  
        Text file containing a list of systems to not generate a report on  
  
    .EXAMPLE  
        Get a list of local admins without reporting to SCCM or writing  
output to text file  
powershell.exe -file LocalAdmins.ps1  
  
        Get a list of local admins and report to SCCM  
powershell.exe -file LocalAdmins.ps1 -SCCMReporting  
  
        Get a list of local admins and write report to a text file at a  
specified location  
powershell.exe -file LocalAdmins.ps1 -OutputFile  
  
    .NOTES  
  
=====
```

v5.2.129 Created with: SAPIEN Technologies, Inc., PowerShell Studio 2016

```

Created on:      11/9/2016 12:47 PM
Created by:     Mick Pletcher
Organization:
Filename:      LocalAdministrators.ps1
    
```

```

=====
#>
[CmdletBinding()]
param
(
    [string]
    $MemberExclusionsFile = 'MemberExclusions.txt',
    [switch]
    $OutputFile,
    [string]
    $OutputFileLocation = '',
    [switch]
    $SCCMReporting,
    [string]
    $SystemExclusionsFile = 'SystemExclusions.txt'
)

function Get-RelativePath {
<#
    .SYNOPSIS
        Get the relative path

    .DESCRIPTION
        Returns the location of the currently running PowerShell script

    .NOTES
        Additional information about the function.
#>

[CmdletBinding()][OutputType([string])]
param ()

    $Path = (split-path $SCRIPT:MyInvocation.MyCommand.Path -parent) + "\"
    Return $Path
}

function Invoke-SCCMHardwareInventory {
<#
    .SYNOPSIS
        Initiate a Hardware Inventory

    .DESCRIPTION
        This will initiate a hardware inventory that does not include a full
hardware inventory. This is enough to collect the WMI data.

    .EXAMPLE
        PS C:\> Invoke-SCCMHardwareInventory

    .NOTES
        Additional information about the function.
#>

[CmdletBinding()]
param ()

    $ComputerName = $env:COMPUTERNAME
    $SMScli = [wmiclass] "\\$ComputerName\root\ccm:SMS_Client"
    
```



```

Null    $SMSCli.TriggersSchedule("{00000000-0000-0000-0000-000000000001}") | Out-
}

function New-WMIClass {
    [CmdletBinding()]
    param
    (
        [validateNotNullOrEmpty()][string]
        $Class
    )

    $WMITest = Get-WmiObject $Class -ErrorAction SilentlyContinue
    If ($WMITest -ne $null) {
        $Output = "Deleting " + $Class + " WMI class....."
        Remove-WmiObject $Class
        $WMITest = Get-WmiObject $Class -ErrorAction SilentlyContinue
        If ($WMITest -eq $null) {
            $Output += "success"
        } else {
            $Output += "Failed"
            Exit 1
        }
        Write-Output $Output
    }
    $Output = "Creating " + $Class + " WMI class....."
    $newClass = New-Object System.Management.ManagementClass("root\cimv2",
[String]::Empty, $null);
    $newClass["__CLASS"] = $Class;
    $newClass.Qualifiers.Add("Static", $true)
    $newClass.Properties.Add("Domain", [System.Management.CimType]::String,
>false)
    $newClass.Properties["Domain"].Qualifiers.Add("key", $true)
    $newClass.Properties["Domain"].Qualifiers.Add("read", $true)
    $newClass.Properties.Add("User", [System.Management.CimType]::String,
>false)
    $newClass.Properties["User"].Qualifiers.Add("key", $false)
    $newClass.Properties["User"].Qualifiers.Add("read", $true)
    $newClass.Put() | Out-Null
    $WMITest = Get-WmiObject $Class -ErrorAction SilentlyContinue
    If ($WMITest -eq $null) {
        $Output += "success"
    } else {
        $Output += "Failed"
        Exit 1
    }
    Write-Output $Output
}

function New-WMIInstance {
    <#
        .SYNOPSIS
            Write new instance

        .DESCRIPTION
            A detailed description of the New-WMIInstance function.

        .PARAMETER MappedDrives
            List of mapped drives

        .PARAMETER Class
            A description of the Class parameter.

```

```

.EXAMPLE
  PS C:\> New-WMIInstance

.NOTES
  Additional information about the function.
#>

[CmdletBinding()]
param
(
    [ValidateNotNullOrEmpty()][array]
    $LocalAdministrators,
    [string]
    $Class
)

foreach ($LocalAdministrator in $LocalAdministrators) {
    $Output = "writing" + [char]32 + $LocalAdministrator.User + [char]32 +
"instance to" + [char]32 + $Class + [char]32 + "class....."
    $Return = Set-WmiInstance -Class $Class -Arguments @{ Domain =
$LocalAdministrator.Domain; User = $LocalAdministrator.User }
    If ($Return -like "*" + $LocalAdministrator.User + "*") {
        $Output += "Success"
    } else {
        $Output += "Failed"
    }
    Write-Output $Output
}
}

cls
#Get the path this script is being executed from
$RelativePath = Get-RelativePath
#Name of the computer this script is being executed on
$ComputerName = $Env:COMPUTERNAME
#Read the list of systems to exclude from reporting
$File = $RelativePath + $SystemExclusionsFile
$SystemExclusions = Get-Content $File
If ($SystemExclusions -notcontains $Env:COMPUTERNAME) {
    #Get list of users to exclude from reporting
    $File = $RelativePath + $MemberExclusionsFile
    $MemberExclusions = Get-Content $File
    #Get list of local administrators while excluding specified members
    $Members = net localgroup administrators | where-object { $_ -AND $_ -
notmatch "command completed successfully" } | select -skip 4 | where-object {
$MemberExclusions -notcontains $_ }
    $LocalAdmins = @()
    foreach ($Member in $Members) {
        #Create new object
        $Admin = New-Object -TypeName System.Management.Automation.PSObject
        $Member = $Member.Split("\")
        If ($Member.Length -gt 1) {
            Add-Member -InputObject $Admin -MemberType NoteProperty -Name
Domain -value $Member[0].Trim()
            Add-Member -InputObject $Admin -MemberType NoteProperty -Name
User -value $Member[1].Trim()
        } else {
            Add-Member -InputObject $Admin -MemberType NoteProperty -Name
Domain -value ""
            Add-Member -InputObject $Admin -MemberType NoteProperty -Name
User -value $Member.Trim()
        }
        $LocalAdmins += $Admin
    }
}

```

```
    }
}
#Report output to WMI which will report up to SCCM
If ($SCCMReporting.IsPresent) {
    New-WMIClass -Class "LocalAdministrators"
    New-WMIInstance -Class "LocalAdministrators" -LocalAdministrators
$LocalAdmins
    #Report WMI entry to SCCM
    Invoke-SCCMHardwareInventory
}
If ($OutputFile.IsPresent) {
    If ($OutputFileLocation[$OutputFileLocation.Length - 1] -ne "\") {
        $File = $OutputFileLocation + "\" + $ComputerName + ".log"
    } else {
        $File = $OutputFileLocation + $ComputerName + ".log"
    }
    #Delete old log file if it exists
    $Output = "Deleting $ComputerName.log....."
    If ((Test-Path $File) -eq $true) {
        Remove-Item -Path $File -Force
    }
    If ((Test-Path $File) -eq $false) {
        $Output += "Success"
    } else {
        $Output += "Failed"
    }
    Write-Output $Output
    $Output = "Writing local admins to $ComputerName.log....."
    $LocalAdmins | Out-File $File
    If ((Test-Path $File) -eq $true) {
        $Output += "Success"
    } else {
        $Output += "Failed"
    }
    Write-Output $Output
}
#Display list of local administrators to screen
$LocalAdmins
```

Chapter 14

Configuring Power Settings using PowerShell

By: Mick Pletcher - MVP

This is another part of the Windows 10 project I am working on automating. This script will set the Windows 10 power settings. I created this, with the help of Sapien's PowerShell Studio that helped make this script much more robust, so that it can be executed during the image to configure the settings there. This allows me to take the settings out of GPO and put them on the local machine to help speed up the logon process. The users at the firm I work at do not have local administrator privileges, so this works great.

The advantage to using this script for configuring power settings, over using powercfg.exe directly, is that the script will verify the setting took place. It goes back and checks the new value to make sure it coincides with the desired value. If it was not successful, the script will return an error code 5, which I arbitrarily chose. This allows you to use the script in a build and it will report the error back to alert you at the end of the build if the power setting was unsuccessful.

The script can set individual power settings per the command line, you can hardcode the settings into the script (I commented out an example), or you can import a power scheme. I also included the feature to generate a detailed and formatted report of all power settings on a machine. The report not only displays to the screen, but it also generates a file named PowerSchemeReport.txt in the same directory as the script. I have included command line examples in the comments of the script.

Set-PowerScheme.ps1

```
<#  
    .SYNOPSIS  
        Set the Power Options  
  
    .DESCRIPTION  
        This script will set the preferred plan. It can also  
        customize specific settings within a plan. There is an option to use the script  
        for generating a report on the currently selected plan, along with all of the  
        plan settings that is written both to the screen and to a log file. The script
```

will exit with an error code 5 if any power setting failed. This allows for an error flag when used during a build.

```
.PARAMETER Balanced
    Selects the balanced plan

.PARAMETER ConsoleTitle
    Name for the PowerShell Console Title

.PARAMETER Custom
    Enter a name to create a custom Power Plan

.PARAMETER HighPerformance
    Selects the High Performance Plan

.PARAMETER ImportPowerSchemeFile
    Import a power scheme file

.PARAMETER PowerSaver
    Selects the Power Saver Plan

.PARAMETER PowerSchemeName
    Name to use when renaming an imported scheme

.PARAMETER Report
    Select this switch to generate a report of the currently
selected plan

.PARAMETER SetPowerSchemeSetting
    Set individual power scheme setting

.PARAMETER SetPowerSchemeSettingValue
    Value associated with the Power Scheme Setting

.PARAMETER SetImportedPowerSchemeDefault
    This is used in conjunction with the
ImportPowerSchemeFile parameter. This tells the script to set the imported power
scheme as the default.

.EXAMPLE
    Set Power Settings to Balanced
powershell.exe -executionpolicy bypass -file Set-
PowerScheme.ps1 -Balanced

    Set Power Settings to High Performance
powershell.exe -executionpolicy bypass -file Set-
PowerScheme.ps1 -HighPerformance

    Set Power Settings to Power Saver
powershell.exe -executionpolicy bypass -file Set-
PowerScheme.ps1 -PowerSaver

    Generate a report named PowerSchemeReport.txt that
resides in the same directory as this script. It contains a list of all power
settings.
powershell.exe -executionpolicy bypass -file Set-
PowerScheme.ps1 -Report

    Set individual power scheme setting
powershell.exe -executionpolicy bypass -file Set-
PowerScheme.ps1 -SetPowerSchemeSetting -MonitorTimeoutAC 120
```

```

        Import power scheme file that resides in the same
directory as this script and renames the scheme to the name defined under
PowerSchemeName
        powershell.exe -executionpolicy bypass -file Set-
PowerScheme.ps1 -ImportPowerSchemeFile "CustomScheme.cfg" -PowerSchemeName
"Custom"

```

.NOTES

```

=====
PowerShell Studio 2016 v5.2.127      Created with:          SAPIEN Technologies, Inc.,
                                     Created on:             8/16/2016 10:13 AM
                                     Created by:             Mick Pletcher
                                     Organization:
                                     Filename:              Set-PowerScheme.ps1
=====

```

```

=====
#>
[CmdletBinding()]
param
(
    [switch]
    $Balanced,
    [string]
    $ConsoleTitle = 'PowerScheme',
    [string]
    $Custom,
    [switch]
    $HighPerformance,
    [string]
    $ImportPowerSchemeFile,
    [switch]
    $PowerSaver,
    [string]
    $PowerSchemeName,
    [switch]
    $Report,
    [ValidateSet('MonitorTimeoutAC', 'MonitorTimeoutDC',
'DiskTimeoutAC', 'DiskTimeoutDC', 'StandbyTimeoutAC', 'StandbyTimeoutDC',
'HibernateTimeoutAC', 'HibernateTimeoutDC')] [string]
    $SetPowerSchemeSetting,
    [string]
    $SetPowerSchemeSettingValue,
    [switch]
    $SetImportedPowerSchemeDefault
)

function Get-PowerScheme {
<#
    .SYNOPSIS
        Get the currently active PowerScheme

    .DESCRIPTION
        This will query the current power scheme and return the
GUID and user friendly name

    .EXAMPLE
        PS C:\> Get-PowerScheme

    .NOTES

```

```

#>                                     Additional information about the function.

[CmdletBinding()][OutputType([object])]
param ()

#Get the currently active power scheme
$query = powercfg.exe /getactivescheme
#Get the alias name of the active power scheme
$ActiveSchemeName = ($query.Split("()").Trim())[1]
#Get the GUID of the active power scheme
$ActiveSchemeGUID = ($query.Split(":").Trim())[1]
$query = powercfg.exe /query $ActiveSchemeGUID
$GUIDAlias = ($query | where { $_.Contains("GUID Alias:")
}).Split(":")[1].Trim()
$scheme = New-Object -TypeName PSObject
$scheme | Add-Member -Type NoteProperty -Name PowerScheme -Value
$ActiveSchemeName
$scheme | Add-Member -Type NoteProperty -Name GUIDAlias -Value
$GUIDAlias
$scheme | Add-Member -Type NoteProperty -Name GUID -Value
$ActiveSchemeGUID
Return $scheme
}

function Get-PowerSchemeSubGroupSettings {
<#
    .SYNOPSIS
        Get the Power Scheme SubGroup Settings

    .DESCRIPTION
        Retrieve all settings and values within a subgroup

    .PARAMETER Subgroup
        Name and GUID of desired subgroup

    .PARAMETER ActivePowerScheme
        GUID and name of the active ActivePowerScheme

    .EXAMPLE
        PS C:\> Get-PowerSchemeSubGroupSettings -Subgroup
$value1

    .NOTES
        Additional information about the function.
#>

[CmdletBinding()]
param
(
    [ValidateNotNullOrEmpty()]$Subgroup,
    [ValidateNotNullOrEmpty()][object]
    $ActivePowerScheme
)

$query = powercfg.exe /query $ActivePowerScheme.GUID $Subgroup.GUID
$query = $query | where { (!(($_.Contains($ActivePowerScheme.GUID)))
-and (!(($_.Contains($ActivePowerScheme.GUIDAlias)))) ) }
$Settings = @()
For ($i = 0; $i -lt $query.Length; $i++) {
    If ($query[$i] -like "*Power Setting GUID:*") {
        $Setting = New-Object System.Object
        #Get the friendly name of the Power Setting

```

```

    $SettingName = $Query[$i].Split("(").Trim()
    $SettingName = $SettingName[1]
    #Get the alias of the power setting
    If ($Query[$i + 1] -like "*GUID Alias:*") {
        $SettingAlias = $Query[$i +
1].split(":").Trim()
    } else {
        $SettingAlias = $SettingAlias[1]
    }
    #Get the GUID of the power setting
    $SettingGUID = $Query[$i].Split("(").Trim()
    $SettingGUID = $SettingGUID[1]
    #Get the AC and DC power settings
    $j = $i
    Do {
        $j++
    } while ($Query[$j] -notlike "*Current AC
Power Setting*")
    $SettingAC = $Query[$j].Split(":").Trim()
    $SettingAC =
[Convert]::ToInt32($SettingAC[1], 16)
    $SettingDC = $Query[$j +
1].split(":").Trim()
    $SettingDC =
[Convert]::ToInt32($SettingDC[1], 16)
    $Setting | Add-Member -Type NoteProperty -
Name Subgroup -Value $Subgroup.Subgroup
    $Setting | Add-Member -Type NoteProperty -
Name Name -Value $SettingName
    $Setting | Add-Member -Type NoteProperty -
Name Alias -Value $SettingAlias
    $Setting | Add-Member -Type NoteProperty -
Name GUID -Value $SettingGUID
    $Setting | Add-Member -Type NoteProperty -
Name AC -Value $SettingAC
    $Setting | Add-Member -Type NoteProperty -
Name DC -Value $SettingDC
    $Settings += $Setting
    }
    }
    Return $Settings
}
}

function Get-RelativePath {
<#
    .SYNOPSIS
        Get the relative path

    .DESCRIPTION
        Returns the location of the currently running PowerShell
script

    .NOTES
        Additional information about the function.

#>

[CmdletBinding()][OutputType([string])]
param ()

    $Path = (split-path $SCRIPT:MyInvocation.MyCommand.Path -parent) +
"\\"

```



```

        Return $Path
    }
function Get-SubGroupsList {
<#
    .SYNOPSIS
        Generate a list of subgroups

    .DESCRIPTION
        This will generate a list of the subgroups within the
designated power scheme

    .PARAMETER ActivePowerScheme
        GUID and name of the active ActivePowerScheme

    .EXAMPLE
        PS C:\> Get-SubGroupsList

    .NOTES
        Additional information about the function.
#>

[CmdletBinding()][OutputType([object])]
param
(
    [ValidateNotNullOrEmpty()][object]
    $ActivePowerScheme
)

#Get all settings for the active power scheme
$query = powercfg.exe /query $ActivePowerScheme.GUID
#Get a list of the subgroups
$subgroups = @()
for ($i = 0; $i -lt $query.Length; $i++) {
    if (($query[$i] -like "*Subgroup GUID:*") -and
($query[$i + 1] -notlike "*Subgroup GUID:*")) {
        $Subgroup = New-Object System.Object
        $SubgroupName =
$query[$i].Split("(").Trim()
        $SubgroupName = $SubgroupName[1]
        If ($query[$i + 1] -like "*GUID Alias:*") {
            $SubgroupAlias = $query[$i +
1].Split(":").Trim()
            $SubgroupAlias =
$SubgroupAlias[1]
        } else {
            $SubgroupAlias = $null
        }
        $SubgroupGUID =
$query[$i].Split(":(").Trim()
        $SubgroupGUID = $SubgroupGUID[1]
        $Subgroup | Add-Member -Type NoteProperty -
Name Subgroup -value $SubgroupName
        $Subgroup | Add-Member -Type NoteProperty -
Name Alias -value $SubgroupAlias
        $Subgroup | Add-Member -Type NoteProperty -
Name GUID -value $SubgroupGUID
        $Subgroups += $Subgroup
    }
}
Return $subgroups
}

```

```

function Import-PowerScheme {
<#
    .SYNOPSIS
        Import a Power Scheme

    .DESCRIPTION
        Imports a power scheme configuration file

    .PARAMETER File
        Name of the configuration file. This must reside in the
same directory as this script.

    .PARAMETER PowerSchemeName
        Desired name for the imported power scheme

    .PARAMETER SetActive
        Set the imported scheme to active

    .EXAMPLE
        PS C:\> Import-PowerScheme -File 'value1'

    .NOTES
        Additional information about the function.
#>

[CmdletBinding()][OutputType([boolean])]
param
(
    [ValidateNotNullOrEmpty()][string]
    $File,
    [ValidateNotNullOrEmpty()][string]
    $PowerSchemeName,
    [switch]
    $SetActive
)

$RelativePath = Get-RelativePath
$File = $RelativePath + $File
#Get list of all power schemes
$OldPowerSchemes = powercfg.exe /l
#Filter out all data except for the GUID
$OldPowerSchemes = $OldPowerSchemes | where { $_ -like "*Power
Scheme GUID*" } | ForEach-Object { $_ -replace "Power Scheme GUID: ", "" } |
ForEach-Object { ($_.split("?(")[0]}
Write-Host "Importing Power Scheme....." -NoNewline
#Import Power Scheme
$output = powercfg.exe -import $File
#Get list of all power schemes
$NewPowerSchemes = powercfg.exe /l
#Filter out all data except for the GUID
$NewScheme = $NewPowerSchemes | where { $_ -like "*Power Scheme
GUID*" } | ForEach-Object { $_ -replace "Power Scheme GUID: ", "" } | ForEach-
Object { ($_.split("?(")[0]} | where { $OldPowerSchemes -notcontains $_ }
If ($NewScheme -ne $null) {
    Write-Host "Success" -ForegroundColor Yellow
    $Error = $false
} else {
    Write-Host "Failed" -ForegroundColor Red
    $Error = $true
}
#Rename imported power scheme
Write-Host "Renaming imported power scheme
to"$PowerSchemeName"....." -NoNewline

```

```

    $Switches = "/changenam" + [char]32 + $NewScheme.Trim() + [char]32
+ [char]34 + $PowerSchemeName + [char]34
    $ErrCode = (Start-Process -FilePath "powercfg.exe" -ArgumentList
$Switches -windowStyle Minimized -wait -Passthru).ExitCode
    $NewPowerSchemes = powercfg.exe /l
    If ($ErrCode -eq 0) {
$PowerSchemeName + "*" ) {
        If ($Test -ne $null) {
            Write-Host "Success" -ForegroundColor Yellow
            $Error = $false
        } else {
            Write-Host "Failed" -ForegroundColor Red
            $Error = $true
            Return $Error
        }
    }
    Write-Host "Setting"$PowerSchemeName" to default...." -NoNewline
    $Switches = "-setactive " + $NewScheme.Trim()
    $ErrCode = (Start-Process -FilePath "powercfg.exe" -ArgumentList
$Switches -windowStyle Minimized -wait -Passthru).ExitCode
    $Query = powercfg.exe /getactivescheme
    #Get the alias name of the active power scheme
    $ActiveSchemeName = (powercfg.exe
/getactivescheme).split("(").Trim()[1]
    If ($ActiveSchemeName -eq $PowerSchemeName) {
        Write-Host "Success" -ForegroundColor Yellow
        $Error = $false
    } else {
        Write-Host "Failed" -ForegroundColor Red
        $Error = $true
    }
    Return $Error
}

function Publish-Report {
<#
    .SYNOPSIS
        Publish a Power Scheme Report

    .DESCRIPTION
        This will publish a report of the currently active power
scheme, a list of the power scheme subgroups, and a list of all subgroup
settings.

    .EXAMPLE
        PS C:\> Publish-Report

    .NOTES
        Additional information about the function.
#>

[CmdletBinding()]
param ()

#Get the relative path this script is being executed from
$RelativePath = Get-RelativePath
#Get the currently enabled power scheme data
$ActivePowerScheme = Get-PowerScheme
#Get a list of all available subgroups
$PowerSchemeSubGroups = Get-SubGroupsList -ActivePowerScheme
$ActivePowerScheme
#Get a list of all settings under each subgroup

```

```

    $PowerSchemeSettings = @()
    for ($i = 0; $i -lt $PowerSchemeSubGroups.Length; $i++) {
        $PowerSchemeSubGroupSettings = Get-
PowerSchemeSubGroupSettings -ActivePowerScheme $ActivePowerScheme -Subgroup
$PowerSchemeSubGroups[$i]
        $PowerSchemeSettings += $PowerSchemeSubGroupSettings
    }
    #Define the Report text file to write to
    $ReportFile = $RelativePath + "PowerSchemeReport.txt"
    #Remove old report if it exists
    If ((Test-Path $ReportFile) -eq $true) {
        Remove-Item -Path $ReportFile -Force
    }
    #Generate Header for Power Scheme Report
    $Header = "ACTIVE POWER SCHEME REPORT"
    $Header | Tee-Object -FilePath $ReportFile -Append
    $Header = "-----"
-----
    $Header | Tee-Object -FilePath $ReportFile -Append
    #Get Active Power Scheme report
    $Output = $ActivePowerScheme | Format-Table
    #Write output to report screen and file
    $Output | Tee-Object -FilePath $ReportFile -Append
    #Generate Header for power scheme subgroups report
    $Header = "POWER SCHEME SUBGROUPS REPORT"
    $Header | Tee-Object -FilePath $ReportFile -Append
    $Header = "-----"
-----
    $Header | Tee-Object -FilePath $ReportFile -Append
    $Output = $PowerSchemeSubgroups | Format-Table
    #Write output to report screen and file
    $Output | Tee-Object -FilePath $ReportFile -Append
    #Generate Header for power scheme subgroup settings report
    $Header = "POWER SCHEME SUBGROUP SETTINGS REPORT"
    $Header | Tee-Object -FilePath $ReportFile -Append
    $Header = "-----"
-----
    $Header | Tee-Object -FilePath $ReportFile -Append
    $Output = $PowerSchemeSettings | Format-Table
    #Write output to report screen and file
    $Output | Tee-Object -FilePath $ReportFile -Append
}

function Set-ConsoleTitle {
<#
    .SYNOPSIS
        Console Title

    .DESCRIPTION
        Sets the title of the PowerShell Console

    .PARAMETER Title
        Title of the PowerShell Console

    .NOTES
        Additional information about the function.

#>

[CmdletBinding()]
param
(
    [Parameter(Mandatory = $true)][String]
    $Title

```

```

    )
    $host.ui.RawUI.WindowTitle = $Title
}
function Set-PowerScheme {
<#
    .SYNOPSIS
        Set the power scheme to the specified scheme
    .DESCRIPTION
        Sets the power scheme to the specified scheme
    .PARAMETER PowerScheme
        Friendly power scheme name
    .PARAMETER CustomPowerScheme
        Create a custom power scheme
    .EXAMPLE
        PS C:\> Set-PowerScheme -PowerScheme 'Value1'
    .NOTES
        Additional information about the function.
#>

    [CmdletBinding()][OutputType([boolean])]
    param
    (
        [ValidateSet('Balanced', 'High Performance', 'Power
Saver')][string]
        $PowerScheme,
        [string]
        $CustomPowerScheme
    )

    #Get list of existing power schemes
    $PowerSchemes = powercfg.exe /1
    If ($PowerScheme -ne $null) {
        #Filter out all schemes except for $PowerScheme and
return the GUID
        $PowerSchemes = ($PowerSchemes | where { $_ -like "*" +
$PowerScheme + "*" }) .split(":").Trim()[1]
        #Set power scheme
        $ActivePowerScheme = Get-PowerScheme
        $ActivePowerScheme.PowerScheme
        Write-Host "Setting Power Scheme
from"$ActivePowerScheme.PowerScheme"to"$PowerScheme"....." -NoNewline
        $Output = powercfg.exe -setactive $PowerSchemes
        $ActivePowerScheme = Get-PowerScheme
        If ($PowerScheme -eq $ActivePowerScheme.PowerScheme) {
            Write-Host "Success" -ForegroundColor Yellow
            Return $false
        } else {
            Write-Host "Failed" -ForegroundColor Red
            Return $true
        }
    }
}

function Set-PowerSchemeSettings {
<#
    .SYNOPSIS

```

```

        Modify current power scheme
    .DESCRIPTION
    This will modify settings of the currently active power
scheme.
    .PARAMETER MonitorTimeoutAC
    Modify the time until the screensaver turns on while
plugged into AC outlet
    .PARAMETER MonitorTimeoutDC
    Modify the time until the screensaver turns on while on
battery power
    .PARAMETER DiskTimeoutAC
    Time that windows will wait for a hard disk to respond
to a command while plugged into AC outlet
    .PARAMETER DiskTimeoutDC
    Time that windows will wait for a hard disk to respond
to a command while on battery power
    .PARAMETER StandbyTimeoutAC
    Amount of time before a computer is put on standby while
plugged into AC outlet
    .PARAMETER StandbyTimeoutDC
    Amount of time before a computer is put on standby while
on battery power
    .PARAMETER HibernateTimeoutAC
    Amount of time before a computer is put in hibernation
while plugged into AC outlet
    .PARAMETER HibernateTimeoutDC
    Amount of time before a computer is put in hibernation
while on battery power
    .EXAMPLE
    PS C:\> Set-PowerSchemeSettings -MonitorTimeoutAC
$value1 -MonitorTimeoutDC $value2
    .NOTES
    Additional information about the function.
#>
```

```
[CmdletBinding()]
param
(
    [string]
    $MonitorTimeoutAC,
    [string]
    $MonitorTimeoutDC,
    [string]
    $DiskTimeoutAC,
    [string]
    $DiskTimeoutDC,
    [string]
    $StandbyTimeoutAC,
    [string]
    $StandbyTimeoutDC,
    [string]
    $HibernateTimeoutAC,
```

```

        [string]
        $HibernateTimeoutDC
    )

    $Scheme = Get-PowerScheme
    If (($MonitorTimeoutAC -ne $null) -and ($MonitorTimeoutAC -ne "")) {
        Write-Host "Setting monitor timeout on AC"
        to"$MonitorTimeoutAC" minutes....." -NoNewline
        $Switches = "/change" + [char]32 + "monitor-timeout-ac"
    + [char]32 + $MonitorTimeoutAC
        $TestKey =
        "Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\User\PowerS
        chemes\" + $Scheme.GUID + "\7516b95f-f776-4464-8c53-06167f40cc99\3c0bc021-c8a8-
        4e07-a973-6b14cbcb2b7e"
        $TestValue = $MonitorTimeoutAC
        $PowerIndex = "ACSettingIndex"
    }
    If (($MonitorTimeoutDC -ne $null) -and ($MonitorTimeoutDC -ne "")) {
        Write-Host "Setting monitor timeout on DC"
        to"$MonitorTimeoutDC" minutes....." -NoNewline
        $Switches = "/change" + [char]32 + "monitor-timeout-dc"
    + [char]32 + $MonitorTimeoutDC
        $TestKey =
        "Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\User\PowerS
        chemes\" + $Scheme.GUID + "\7516b95f-f776-4464-8c53-06167f40cc99\3c0bc021-c8a8-
        4e07-a973-6b14cbcb2b7e"
        $TestValue = $MonitorTimeoutDC
        $PowerIndex = "DCSettingIndex"
    }
    If (($DiskTimeoutAC -ne $null) -and ($DiskTimeoutAC -ne "")) {
        Write-Host "Setting disk timeout on AC"
        to"$DiskTimeoutAC" minutes....." -NoNewline
        $Switches = "/change" + [char]32 + "disk-timeout-ac" +
        [char]32 + $DiskTimeoutAC
        $TestKey =
        "Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\User\PowerS
        chemes\" + $Scheme.GUID + "\0012ee47-9041-4b5d-9b77-535fba8b1442\6738e2c4-e8a5-
        4a42-b16a-e040e769756e"
        $TestValue = $DiskTimeoutAC
        $PowerIndex = "ACSettingIndex"
    }
    If (($DiskTimeoutDC -ne $null) -and ($DiskTimeoutDC -ne "")) {
        Write-Host "Setting disk timeout on DC"
        to"$DiskTimeoutDC" minutes....." -NoNewline
        $Switches = "/change" + [char]32 + "disk-timeout-dc" +
        [char]32 + $DiskTimeoutDC
        $TestKey =
        "Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\User\PowerS
        chemes\" + $Scheme.GUID + "\0012ee47-9041-4b5d-9b77-535fba8b1442\6738e2c4-e8a5-
        4a42-b16a-e040e769756e"
        $TestValue = $DiskTimeoutDC
        $PowerIndex = "DCSettingIndex"
    }
    If (($StandbyTimeoutAC -ne $null) -and ($StandbyTimeoutAC -ne "")) {
        Write-Host "Setting standby timeout on AC"
        to"$StandbyTimeoutAC" minutes....." -NoNewline
        $Switches = "/change" + [char]32 + "standby-timeout-ac"
    + [char]32 + $StandbyTimeoutAC
        $TestKey =
        "Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\User\PowerS
        chemes\" + $Scheme.GUID + "\238c9fa8-0aad-41ed-83f4-97be242c8f20\29f6c1db-86da-
        48c5-9fdb-f2b67b1f44da"
        $TestValue = $StandbyTimeoutAC
    }

```

```

        $PowerIndex = "ACSettingIndex"
    }
    If (($StandbyTimeoutDC -ne $null) -and ($StandbyTimeoutDC -ne "")) {
        Write-Host "Setting standby timeout on DC
to"$StandbyTimeoutDC" minutes...." -NoNewline
        $Switches = "/change" + [char]32 + "standby-timeout-dc"
+ [char]32 + $StandbyTimeoutDC
        $TestKey =
"Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\User\PowerS
chemes\" + $Scheme.GUID + "\238c9fa8-0aad-41ed-83f4-97be242c8f20\29f6c1db-86da-
48c5-9fdb-f2b67b1f44da"
        $TestValue = $StandbyTimeoutDC
        $PowerIndex = "DCSettingIndex"
    }
    If (($HibernateTimeoutAC -ne $null) -and ($HibernateTimeoutAC -ne
"")) {
        Write-Host "Setting hibernate timeout on AC
to"$HibernateTimeoutAC" minutes...." -NoNewline
        $Switches = "/change" + [char]32 + "hibernate-timeout-
ac" + [char]32 + $HibernateTimeoutAC
        $TestKey =
"Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\User\PowerS
chemes\" + $Scheme.GUID + "\238c9fa8-0aad-41ed-83f4-97be242c8f20\9d7815a6-7ee4-
497e-8888-515a05f02364"
        [int]$TestValue = $HibernateTimeoutAC
        $PowerIndex = "ACSettingIndex"
    }
    If (($HibernateTimeoutDC -ne $null) -and ($HibernateTimeoutDC -ne
"")) {
        Write-Host "Setting hibernate timeout on DC
to"$HibernateTimeoutDC" minutes...." -NoNewline
        $Switches = "/change" + [char]32 + "hibernate-timeout-
dc" + [char]32 + $HibernateTimeoutDC
        $TestKey =
"Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\User\PowerS
chemes\" + $Scheme.GUID + "\238c9fa8-0aad-41ed-83f4-97be242c8f20\9d7815a6-7ee4-
497e-8888-515a05f02364"
        $TestValue = $HibernateTimeoutDC
        $PowerIndex = "DCSettingIndex"
    }
    $ErrCode = (Start-Process -FilePath "powercfg.exe" -ArgumentList
$Switches -WindowStyle Minimized -Wait -Passthru).ExitCode
    $RegValue = (((Get-ItemProperty $TestKey).$PowerIndex) /60)
    #Round down to the nearest tenth due to hibernate values being 1
decimal off
    $RegValue = $RegValue - ($RegValue % 10)
    If (($RegValue -eq $TestValue) -and ($ErrCode -eq 0)) {
        Write-Host "Success" -ForegroundColor Yellow
        $Errors = $false
    } else {
        Write-Host "Failed" -ForegroundColor Red
        $Errors = $true
    }
    Return $Errors
}

cls
#Set Errors variable to false to begin the script with no errors
$Errors = $false
#Set the title of the PowerShell console
Set-ConsoleTitle -Title $ConsoleTitle

```



```

<#Hardcoded Power Scheme Settings
$Errors = Set-PowerSchemeSettings -MonitorTimeoutAC 120
$Errors = Set-PowerSchemeSettings -MonitorTimeoutDC 120
$Errors = Set-PowerSchemeSettings -DiskTimeOutAC 120
$Errors = Set-PowerSchemeSettings -DiskTimeOutDC 120
$Errors = Set-PowerSchemeSettings -StandbyTimeoutAC 120
$Errors = Set-PowerSchemeSettings -StandbyTimeoutDC 120
$Errors = Set-PowerSchemeSettings -HibernateTimeoutAC 60
$Errors = Set-PowerSchemeSettings -HibernateTimeoutDC 60
#>

#Generate a report if -Report is specified
If ($Report.IsPresent) {
    Publish-Report
}
#Set the Power Scheme to Balanced
If ($Balanced.IsPresent) {
    $Errors = Set-PowerScheme -PowerScheme 'Balanced'
}
#Set the Power Scheme to Power Saver
If ($PowerSaver.IsPresent) {
    $Errors = Set-PowerScheme -PowerScheme 'Power Saver'
}
#Set the Power Scheme to High Performance
If ($HighPerformance.IsPresent) {
    $Errors = Set-PowerScheme -PowerScheme 'High Performance'
}
#Set the Power Scheme to Custom
If (($Custom -ne $null) -and ($Custom -ne "")) {
    $Errors = Set-PowerScheme -PowerScheme $Custom
}
#Import a power scheme
If (($ImportPowerSchemeFile -ne $null) -and ($ImportPowerSchemeFile -ne "")) {
    If ($SetImportedPowerSchemeDefault.IsPresent) {
        $Errors = Import-PowerScheme -File
    }
    $ImportPowerSchemeFile -PowerSchemeName $PowerSchemeName -SetActive
} else {
    $Errors = Import-PowerScheme -File
}
$ImportPowerSchemeFile -PowerSchemeName $PowerSchemeName
}
}
#Set individual power scheme setting from command line
If (($SetPowerSchemeSetting -ne $null) -and ($SetPowerSchemeSetting -ne "")) {
    switch ($SetPowerSchemeSetting) {
        "MonitorTimeoutAC" { $Errors = Set-PowerSchemeSettings -
MonitorTimeoutAC $SetPowerSchemeSettingValue }
        "MonitorTimeoutDC" { $Errors = Set-PowerSchemeSettings -
MonitorTimeoutDC $SetPowerSchemeSettingValue }
        "DiskTimeOutAC" { $Errors = Set-PowerSchemeSettings -
DiskTimeOutAC $SetPowerSchemeSettingValue }
        "DiskTimeOutDC" { $Errors = Set-PowerSchemeSettings -
DiskTimeOutDC $SetPowerSchemeSettingValue }
        "StandbyTimeoutAC" { $Errors = Set-PowerSchemeSettings -
StandbyTimeoutAC $SetPowerSchemeSettingValue }
        "StandbyTimeoutDC" { $Errors = Set-PowerSchemeSettings -
StandbyTimeoutDC $SetPowerSchemeSettingValue }
        "HibernateTimeoutAC" { $Errors = Set-PowerSchemeSettings
-HibernateTimeoutAC $SetPowerSchemeSettingValue }
        "HibernateTimeoutDC" { $Errors = Set-PowerSchemeSettings
-HibernateTimeoutDC $SetPowerSchemeSettingValue }
    }
}
}

```

```
#Exit with an error code 5 if errors were encountered during any of the power
settings
If ($Errors -eq $true) {
    Exit 5
}
```

Chapter 15

Automated SCCM Endpoint Full System Scan upon Infection with Email Notification

By: Mick Pletcher - MVP

While helping to manage Microsoft Endpoint, a former colleague suggested that I setup Endpoint to automatically run a full system scan each time an infection is detected. I Googled the blog posting on it and although it is a great post, I figured it could be streamlined even more by just using SCCM alone to achieve the same outcome. It is nice when you are out of the office and your backup might not have the time to keep an eye on the antivirus infections. It is also beneficial if you are in an environment where the office is closed overnight and on weekends. If an infection takes place, this can help remediate it without your presence.

This is the third edition. The first edition just initiated a full system scan upon infection. The second edition initiated a full system scan plus would send an email to the designated email address. The third edition now combines the first two and allows for them to be designated using parameters. The code has also been optimized.

I decided to use the SCCM custom application detection to query a system and see if a full system scan has been performed in the event of an infection logged in the event viewer. I first started out by writing a PowerShell script that would perform a WMI query on the SCCM server for the status of the system the application detection was being run on. The problem I ran across was that the application is being run under system credentials, which would require me to pass network credentials within the script. Instead of having to do this, I decided to query the event viewer logs on the local machine to look for the last infection date/time, which is event 1116. I also found that event 1001 and provider name Microsoft Antimalware designate are used when a system scan has been performed. Here are the steps SCCM and PowerShell go through:

1. SCCM deploys the package to the system.

2. The application detection queries the event viewer logs for the last 1116 ID (infection).
3. The application detection queries the event viewer logs for the last 1001 ID and "Microsoft Antimalware" provider name.
4. If a system 1001 ID does not exist since the last infection, the custom detection method will exit out as a failure.
5. If the custom detection failed, the AntiVirusScanEmail.ps1 file will be executed on the machine.
6. An email is sent that tells a scan was performed on %COMPUTERNAME% with the virus details in the body if -Email switch was specified
7. Once the scan is complete, an application deployment evaluation cycle is initiated to update the SCCM server with the status of the system.
8. The application detection is initiated again to confirm the scan occurred.

If you desire emails be sent alerting you of an infection and system scan, then you will need to download and place PsExec.exe in the same directory as this script. The next thing will be to define the Installation program in SCCM using psexec.exe. This allows the PowerShell script to be executed under a domain account, thereby giving it the ability to use the send-mailmessage commandlet. Here is how to do this:

```
psexec.exe \\%computername% -u <domain>\<username> -p <password>
-h cmd.exe /c "echo . | powershell.exe -executionpolicy bypass -
file AntiVirusScanEmail.ps1"
```

Note: Do not change %computername%. The only parts of the psexec.exe command line that need to be changed are <domain>, <username>, and <password>.

If you do not want emails sent, then you can use the following command line parameter:

```
powershell.exe -executionpolicy bypass -
file AntiVirusScanEmail.ps1 -FullScan
```

or

```
powershell.exe -executionpolicy bypass -  
file AntiVirusScanEmail.ps1 -QuickScan
```

This is setup in SCCM as a normal application deployment. The only thing that differs from a standard deployment is the application detection method. The ApplicationVirusDetectionMethodEmail.ps1 script is imported in for the detection method. The AntiVirusScanEmail.ps1 file is setup as the installation program. I have mine entered like this:

```
powershell.exe -executionpolicy bypass -file  
AntiVirusScanEmail.ps1 -FullScan
```

If you also want it to email you, then refer to the section above on using psexec.exe and the example to enter in as the installation program.

One more thing is that I have the application hidden from the software center. There really isn't a need for it to be seen by the end-users.

In order for this to work in a timely manor, you will need to change the software deployment frequency under the client settings. I have mine set at every 8 hours, or three times a day.

ApplicationVirusDetectionMethodEmail.ps1

```
<#  
.NOTES  
=====
```

Created with:	SAPIEN Technologies, Inc., PowerShell Studio 2016 v5.2.127
Created on:	8/5/2016 11:11 AM
Created by:	Mick Pletcher
Organization:	
Filename:	ApplicationVirusDetectionMethod.ps1

```
=====
```

```
.DESCRIPTION  
#>
```

```
$LastInfection = get-winevent -filterhashtable @{ logname = 'system'; ID = 1116  
} -maxevents 1 -ErrorAction SilentlyContinue  
$LastScan = Get-WinEvent -FilterHashtable @{ logname = 'system'; ProviderName =  
'Microsoft Antimalware'; ID = 1001 } -MaxEvents 1  
If ($LastScan.TimeCreated -lt $LastInfection.TimeCreated) {
```

```

        #No scan since last infection
        Start-Sleep -Seconds 5
        exit 0
    } else {
        #No infection since last scan
        Write-Host "No Infection"
        Start-Sleep -Seconds 5
        exit 0
    }
}

```

AntiVirusScanEmail.ps1

```

<#
.SYNOPSIS      EndPoint Virus Scan

.DESCRIPTION

This script will initiate a full or quick scan, whichever switch is selected at
the command line. Once the scan is completed, it will check the event viewer
logs for a scan completed entry to verify the scan successfully completed. If
the Email switch is designated at the command line, then an email is sent to the
specified recipient. It is suggested the $EmailRecipient, $EmailSender, and
$SMTPServer be predefined in the parameter field. I have also included a trigger
of the application deployment evaluation cycle to expedite the process.

.PARAMETER FullScan
        Initiate a full system scan

.PARAMETER QuickScan
        Initiate a quick scan

.PARAMETER Email
        Select if you want an email report sent to the specified email
address

.PARAMETER EmailRecipient
        Receipient's email address

.PARAMETER EmailSender
        Sender's email address

.PARAMETER SMTPServer
        SMTP server address

.EXAMPLE
        Initiate a Quickscan
powershell.exe -executionpolicy bypass -file AntiVirusScanEmail.ps1 -QuickScan

Initiate a Fullscan
powershell.exe -executionpolicy bypass -file AntiVirusScanEmail.ps1 -FullScan

Initiate a Fullscan and send email report. To, From, and SMTP parameters are
pre-defined

```

```
powershell.exe -executionpolicy bypass -file AntiVirusScanEmail.ps1 -FullScan -
Email
```

```
.NOTES
```

```
=====
PowerShell Studio 2016 v5.2.127      Created with:      SAPIEN Technologies, Inc.,
                                      Created on:        8/5/2016 11:11 AM
                                      Created by:        Mick Pletcher
                                      Organization:
                                      Filename:          AntiVirusScanEmail.ps1
=====
```

```
#>
param
(
    [switch]
    $FullScan,
    [switch]
    $QuickScan,
    [switch]
    $Email,
    [string]
    $EmailRecipient = '',
    [string]
    $EmailSender = '',
    [string]
    $SMTPServer = ''
)

#Import the Endpoint Provider module
Import-Module $env:ProgramFiles"\Microsoft Security
Client\MpProvider\MpProvider.psd1"
#Get the relative execution path of this script
$RelativePath = (split-path $SCRIPT:MyInvocation.MyCommand.Path -parent) + "\"
#Find the last infection entry in the event viewer logs
$LastInfection = get-winevent -filterhashtable @{ logname = 'system'; ID = 1116
} -maxevents 1 -ErrorAction SilentlyContinue
#Full Scan
If ($FullScan.IsPresent) {
    #Initiate a full system scan
    Start-MProtScan -ScanType "FullScan"
    #Commented area only there if you want to manually execute this
script to watch it execute
<#
    cls
    Write-Warning "Error: $_"
    Write-Host $_.Exception.ErrorCode
#>
    #Get the last event viewer log written by Endpoint to check if the
full system scan has finished
    $LastScan = Get-WinEvent -FilterHashtable @{ logname = 'system';
ProviderName = 'Microsoft Antimalware'; ID = 1001 } -MaxEvents 1
    #
    If ($LastScan.Message -like '*Microsoft Antimalware scan has
finished*') {
        $EmailBody = "An Endpoint antimalware full system scan
has been performed on" + [char]32 + $env:COMPUTERNAME + [char]32 + "due to the
virus detection listed below." + [char]13 + [char]13 + $LastInfection.Message
    } else {
        $EmailBody = "An Endpoint antimalware full system scan
did not complete on" + [char]32 + $env:COMPUTERNAME + [char]32 + "due to the
virus detection listed below." + [char]13 + [char]13 + $LastInfection.Message
    }
}
}
```

```

#Quick Scan
If ($QuickScan.IsPresent) {
    #Initiate a quick system scan
    Start-MProtScan -ScanType "QuickScan"
    #Commented area only there if you want to manually execute this
    script to watch it execute
    <#
        cls
        Write-Warning "Error: $_"
        Write-Host $_.Exception.ErrorCode
    #>

    #Get the last event viewer log written by Endpoint to check if the
    quick system scan has finished
    $LastScan = Get-WinEvent -FilterHashtable @{ logname = 'system';
    ProviderName = 'Microsoft Antimalware'; ID = 1001 } -MaxEvents 1
    #
    If ($LastScan.Message -like '*Microsoft Antimalware scan has
    finished*') {
        $EmailBody = "An Endpoint antimalware quick system scan
        has been performed on" + [char]32 + $env:COMPUTERNAME + [char]32 + "due to the
        virus detection listed below." + [char]13 + [char]13 + $LastInfection.Message
    } else {
        $EmailBody = "An Endpoint antimalware quick system scan
        did not complete on" + [char]32 + $env:COMPUTERNAME + [char]32 + "due to the
        virus detection listed below." + [char]13 + [char]13 + $LastInfection.Message
    }
}
#Email Infection Report
If ($Email.IsPresent) {
    $Subject = "Microsoft Endpoint Infection Report"
    $EmailSubject = "Virus Detection Report for" + [char]32 +
    $env:COMPUTERNAME
    Send-MailMessage -To $EmailRecipient -From $EmailSender -Subject
    $Subject -Body $EmailBody -SmtpServer $SMTPServer
}
#Initiate Application Deployment Evaluation Cycle
$WMIPath = "\\\" + $env:COMPUTERNAME + "\root\ccm:SMS_Client"
$SMSwmi = [wmi]class $WMIPath
$strAction = "{00000000-0000-0000-0000-000000000121}"
[Void]$SMSwmi.TriggersSchedule($strAction)

```


Chapter 16

Laptop Mandatory Reboot Management with SCCM and PowerShell

By: Mick Pletcher – MVP

Managing laptops in certain environments can be daunting. Reboots are a must every now and then, especially for monthly windows updates. With the sleep and hibernate features being enabled, the chances of a user rebooting a laptop become far less. A laptop can go weeks and even months without a reboot. Working in the legal industry, as I do, adds to the complexity of forcing reboots as you have the issue of not causing a reboot during a hearing or during a client meeting for instance. You want to be as unobtrusive as possible. You might say that this is not needed as SCCM could be setup to automatically perform this same function on a regular schedule. That is true. Where this becomes valuable is when you don't want to force a reboot on users that regularly reboot their machines. If they are already doing this, which we have a fair number of users that do, then there is no need to reboot an additional time that will inconvenience them.

To make this process as unobtrusive as possible, I have written the following two PowerShell scripts, using Sapien's PowerShell Studio, that work in conjunction with SCCM 2012 to give users the leisure of a full business day to reboot the machine. One script is the custom detection method and the other is the installer.

The custom detection method works by reading the last event viewer 1074. It looks at the date of the ID and then sees if it is 14 days or older. This can be set to any number of days other than the 14 my firm has chosen. If it is 14 days old, the script then sets the registry key Rebooted to a DWORD value of 1 and fails the detection method. When the method fails, SCCM will then run the second PowerShell script.

The second script operates by running the same detection method. Once it detects the same values, it resets the Rebooted Key to 0 and then returns the error code 3010 back to SCCM. SCCM then reruns the detection method. The detection method sees there has been 14 days or more and the Rebooted key is set to 0. It returns an error code 0 back to SCCM with a write-host of "Success".

This tells SCCM the application ran successfully and to now process with the soft reboot, which was required by the 3010 return code.

The final part is to configure the Computer Restart under the SCCM client settings. I configured ours to be 480 minutes, which is 8 hours, with a mandatory dialog window that cannot be closed the final 60 minutes.

When the system reboots, the custom detection method runs again and sees there is a new 1074 entry in the event viewer, along with the registry key Rebooted being a 0, therefore it shows successfully installed. As the days progress and SCCM reruns the custom detection method, it will rerun the application script to reboot the machine again if the machine is not rebooted in the 14 allotted days. If the user reboots the machine every week, the SCCM application will never reboot the machine.

I had a few to ask for a tutorial on setting this up. Here is a video I made on how I setup and configured the scripts in SCCM.

MandatoryReboot.ps1

```
<#
.SYNOPSIS          Mandatory Reboot

.DESCRIPTION      This script will read the last time a system rebooted from the event
                  viewer logs. It then calculates the number of days since that time. If the
                  number of days equals or exceeds the RebootThreshold variable, the script will
                  change the registry key Rebooted to a 0. It then exits with an error code 3010,
                  which tells SCCM 2012 to perform a soft reboot.

.NOTES
=====
Created with:      SAPIEN Technologies, Inc., PowerShell Studio 2016
v5.2.122
Created on:        6/8/2016 1:58 PM
Created by:        Mick Pletcher
Organization:      MandatoryReboot.ps1
Filename:          MandatoryReboot.ps1
=====
#>

#Returns "32-bit" or "64-bit"
$Architecture = Get-WmiObject -Class win32_OperatingSystem | Select-Object
OSArchitecture
$Architecture = $Architecture.OSArchitecture
#Tests if the registry key Rebooted exists and creates it if it does not. It
then reads if the system has been rebooted by the value being either a 0 or 1.
```

```

This determines if the reboot has occurred and is set in the MandatoryReboot.ps1
file when the custom detection method triggers its execution
if ($Architecture -eq "32-bit") {
    if ((Test-Path
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot") -eq $false) {
        New-Item
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot" | New-ItemProperty -
Name Rebooted -Value 0 -Force | Out-Null
    }
    $Rebooted = Get-ItemProperty -Name Rebooted -Path
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot"
} else {
    if ((Test-Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot") -eq
$false) {
        New-Item
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot" | New-
ItemProperty -Name Rebooted -Value 0 -Force | Out-Null
    }
    $Rebooted = Get-ItemProperty -Name Rebooted -Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot"
}
#Get the 0 or 1 value if a system has been rebooted within the $RebootThreshold
period
$Rebooted = $Rebooted.Rebooted
#Number of days until reboot becomes mandatory
$RebootThreshold = 14
$Today = Get-Date
#Gets the last reboot from the event viewer logs
$LastReboot = get-winevent -filterhashtable @{ logname = 'system'; ID = 1074 } -
maxevents 1 -ErrorAction SilentlyContinue
#Calculate how long since last reboot. If no event viewer entries since last
reboot, then trigger a reboot
if ($LastReboot -eq $null) {
    $Difference = $RebootThreshold
} else {
    $Difference = New-TimeSpan -Start $Today -End
$LastReboot.TimeCreated
    $Difference = [math]::Abs($Difference.Days)
}
#Change the HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot
key to a 1 if the system is over the pre-determined threshold and $Rebooted = 0
if (($Difference -ge $RebootThreshold) -and ($Rebooted -eq 0)) {
    if ((Test-Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot") -eq $true)
    {
        Set-ItemProperty -Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot" -Name
Rebooted -Value 1 -Type DWORD -Force
        $Rebooted = Get-ItemProperty -Name Rebooted -Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot"
    } else {
        Set-ItemProperty -Path
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot" -Name Rebooted -Value 1
-Type DWORD -Force
        $Rebooted = Get-ItemProperty -Name Rebooted -Path
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot"
    }
}
$Rebooted = $Rebooted.Rebooted
#Change the HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot
key back to 0 if the system has been rebooted within the $RebootThreshold period
if (($Difference -lt $RebootThreshold) -and ($Rebooted -eq 1)) {

```

```

        if ((Test-Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot") -eq $true)
    {
        Set-ItemProperty -Name Rebooted -value 0 -Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot" -Type DWORD
-Force
        $Rebooted = Get-ItemProperty -Name Rebooted -Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot"
    } else {
        Set-ItemProperty -Name Rebooted -value 0 -Path
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot" -Type DWORD -Force
        $Rebooted = Get-ItemProperty -Name Rebooted -Path
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot"
    }
    $Rebooted = $Rebooted.Rebooted
    Write-Host "System is within"$RebootThreshold" Day Reboot Threshold"
}
Write-Host "Reboot Threshold:"$RebootThreshold
Write-Host "Difference:"$Difference
Write-Host "Rebooted:"$Rebooted
Exit 3010

```

MandatoryRebootCustomDetection.ps1

```

<#
.SYNOPSIS
    Mandatory Reboot Custom Detection Method

.DESCRIPTION
    This script will read the last time a system rebooted from the event
viewer logs. It then calculates the number of days since that time. If the
number of days equals or exceeds the RebootThreshold variable, the script will
exit with a return code 0 and no data output. No data output is read by SCCM as
a failure. If the number of days is less than the RebootThreshold, then a
message is written saying the system is within the threshold and the script
exits with a return code of 0. SCCM reads an error code 0 with data output as a
success.

.NOTES
=====
Created with:          SAPIEN Technologies, Inc., PowerShell Studio 2016
v5.2.122
Created on:            6/8/2016 2:04 PM
Created by:            Mick Pletcher
Organization:
Filename:              MandatoryRebootCustomDetection.ps1
=====
#>

#Number of days until reboot becomes mandatory
$RebootThreshold = 14
$Today = Get-Date
#Returns "32-bit" or "64-bit"
$Architecture = Get-WmiObject -Class Win32_OperatingSystem | Select-Object
OSArchitecture
$Architecture = $Architecture.OSArchitecture
#Gets the last reboot from the event viewer logs

```

```

$LastReboot = get-winevent -filterhashtable @{ logname = 'system'; ID = 1074 } -
maxevents 1 -ErrorAction SilentlyContinue
#Tests if the registry key Rebooted exists and creates it if it does not. It
then reads if the system has been rebooted by the value being either a 0 or 1.
This determines if the reboot has occurred and is set in the MandatoryReboot.ps1
file when the custom detection method triggers its execution
if ($Architecture -eq "32-bit") {
    if ((Test-Path
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot") -eq $false) {
        New-Item
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot" | New-ItemProperty -
Name Rebooted -Value 0 -Force | Out-Null
    }
    $Rebooted = Get-ItemProperty -Name Rebooted -Path
"HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Reboot"
} else {
    if ((Test-Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot") -eq
$false) {
        New-Item
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot" | New-
ItemProperty -Name Rebooted -Value 0 -Force | Out-Null
    }
    $Rebooted = Get-ItemProperty -Name Rebooted -Path
"HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Reboot"
}
#Get the 0 or 1 value if a system has been rebooted within the $RebootThreshold
period
$Rebooted = $Rebooted.Rebooted
#Calculate how long since last reboot. If no event viewer entries since last
reboot, then trigger a reboot
if ($LastReboot -eq $null) {
    $Difference = $RebootThreshold
} else {
    $Difference = New-TimeSpan -Start $Today -End
$LastReboot.TimeCreated
    $Difference = [math]::Abs($Difference.Days)
}
#The first two conditions report to SCCM that the deployment is "installed"
thereby not triggering a reboot. The last two report to SCCM the app is "not
installed" and trigger an install
if (($Difference -lt $RebootThreshold) -and ($Rebooted -eq 0)) {
    Write-Host "Success"
    exit 0
}
if (($Difference -ge $RebootThreshold) -and ($Rebooted -eq 1)) {
    Write-Host "Success"
    exit 0
}
if (($Difference -ge $RebootThreshold) -and ($Rebooted -eq 0)) {
    exit 0
}
if (($Difference -lt $RebootThreshold) -and ($Rebooted -eq 1)) {
    exit 0
}
}

```

Chapter 17

Set Windows Features with Verification

By: Mick Pletcher – MVP

I am in the beginning stages of creating a Windows 10 build. One of the first things I needed to do was to install and set the Windows 10 features. Before, I used a batch script that executed DISM to set each feature. I know there is the Install-WindowsFeatures cmdlet, but I also wanted to incorporate verification and other features into a single script.

This script allows you to set windows features while also verifying each feature was set correctly by querying the feature for the status. It then outputs the feature name and status to the display. I have also included the option to run a report of all available features and their state. Here are the four features the script provides:

1. Set an individual feature via command line:

```
powershell.exe -executionpolicy bypass -command windowsFeatures.ps1 -Feature 'RSATClient-Features' -Setting 'disable'
```

2. Set multiple features by reading a text file located in the same directory as the script. You can name the text file any name you want. The format for the file is: RSATClient,enable for example. Here is the command line:

```
powershell.exe -executionpolicy bypass -command windowsFeatures.ps1 -FeaturesFile 'FeaturesList.txt'
```

3. Hard code a feature setting at the bottom of the script:

```
Set-WindowsFeature -Name 'RSATClient-Features' -State 'disable'
```

4. Display a list of windows features:

```
powershell.exe -executionpolicy bypass -command windowsFeatures.ps1 -ListFeatures $true
```

You will need to use the `-command` when executing this at the command line instead of `-file`. This is because the `-ListFeatures` is a boolean value. I have also included code that identifies an error 50 and returns a status that you must include the parent feature before activating the specified feature. I have also made the additional command line window be minimized when running the `DISM.exe`.

WindowsFeatures.ps1

```
<#
.SYNOPSIS
    Process Windows Features

.DESCRIPTION
    This script can return a list of online windows features and/or set
    specific windows features.

.PARAMETER ListFeatures
    Return a list of all Windows Features

.PARAMETER Feature
    A description of the Feature parameter.

.PARAMETER Setting
    A description of the Setting parameter.

.PARAMETER FeaturesFile
    Name of the features file that contains a list of features with
    their corresponding settings for this script to process through. The file
    resides in the same directory as this script.

.EXAMPLE
    Return a list of all available online windows Features
    powershell.exe -executionpolicy bypass -command WindowsFeatures.ps1
-ListFeatures $true

    Set one windows Feature from the command line
    powershell.exe -executionpolicy bypass -command WindowsFeatures.ps1
-Feature 'RSATClient-Features' -Setting 'disable'

    Set multiple features by reading contents of a text file
    powershell.exe -executionpolicy bypass -command WindowsFeatures.ps1
-FeaturesFile 'FeaturesList.txt'

.NOTES
    You must use -command instead of -file in the command line because
    of the use of boolean parameters

    An error code 50 means you are trying to enable a feature in which
    the required parent feature is disabled

    I have also included two commented out lines at the bottom of the
    script as examples if you want to hardcode the features within the script.
=====
Created with:          SAPIEN Technologies, Inc., PowerShell Studio 2016
v5.2.122
```

Created on: 5/27/2016 2:46 PM
 Created by: Mick Pletcher
 Organization:
 Filename: windowsFeatures.ps1

```

=====
#>
[CmdletBinding()]
param
(
    [boolean]$ListFeatures = $false,
    [string]$Feature,
    [ValidateSet('enable', 'disable')][string]$Setting,
    [String]$FeaturesFile
)

function Confirm-Feature {
<#
    .SYNOPSIS
        Confirm the feature setting

    .DESCRIPTION
        Confirm the desired change took place for a feature

    .PARAMETER FeatureName
        Name of the feature

    .PARAMETER FeatureState
        Desired state of the feature

    .EXAMPLE
        PS C:\> Confirm-Feature

    .NOTES
        Additional information about the function.
#>

    [CmdletBinding()][OutputType([boolean])]
    param
    (
        [ValidateNotNull()][string]$FeatureName,
        [ValidateSet('Enable',
'Disable')][string]$FeatureState
    )

    $WindowsFeatures = Get-WindowsFeaturesList
    $WindowsFeature = $WindowsFeatures | Where-Object { $_.Name -eq
$FeatureName }
    switch ($FeatureState) {
        'Enable' {
            If (($WindowsFeature.State -eq 'Enabled') -
or ($WindowsFeature.State -eq 'Enable Pending')) {
                Return $true
            } else {
                Return $false
            }
        }
        'Disable' {
            If (($WindowsFeature.State -eq 'Disabled') -
or ($WindowsFeature.State -eq 'Disable Pending')) {
                Return $true
            } else {
                Return $false
            }
        }
    }
}
    
```



```

    }
    default {
        Return $false
    }
}

function Get-WindowsFeaturesList {
<#
    .SYNOPSIS
        List Windows Features

    .DESCRIPTION
        This will list all available online windows features

    .NOTES
        Additional information about the function.
#>

    [CmdletBinding()]
    param ()

    $Temp = dism /online /get-features
    $Temp = $Temp | where-object { ($_ -like '*Feature Name*') -or ($_ -
like '*State*') }
    $i = 0
    $Features = @()
    Do {
        $FeatureName = $Temp[$i]
        $FeatureName = $FeatureName.Split(':')
        $FeatureName = $FeatureName[1].Trim()
        $i++
        $FeatureState = $Temp[$i]
        $FeatureState = $FeatureState.Split(':')
        $FeatureState = $FeatureState[1].Trim()
        $Feature = New-Object PSObject
        $Feature | Add-Member noteproperty Name $FeatureName
        $Feature | Add-Member noteproperty State $FeatureState
        $Features += $Feature
        $i++
    } while ($i -lt $Temp.Count)
    $Features = $Features | Sort-Object Name
    Return $Features
}

function Set-WindowsFeature {
<#
    .SYNOPSIS
        Configure a Windows Feature

    .DESCRIPTION
        Enable or disable a Windows feature

    .PARAMETER Name
        Name of the Windows feature

    .PARAMETER State
        Enable or disable Windows feature

    .NOTES
        Additional information about the function.
#>

```

```

[CmdletBinding()]
param
(
    [Parameter(Mandatory =
>true)][ValidateNotNullOrEmpty()][string]$Name,
    [Parameter(Mandatory =
>true)][ValidateSet('enable', 'disable')][string]$State
)

$EXE = $env:windir + "\system32\dism.exe"
Write-Host $Name"....." -NoNewline
If ($State -eq "enable") {
    $Parameters = "/online /enable-feature /norestart
/featurename:" + $Name
} else {
    $Parameters = "/online /disable-feature /norestart
/featurename:" + $Name
}
$ErrCode = (Start-Process -FilePath $EXE -ArgumentList $Parameters -
Wait -PassThru -windowStyle Minimized).ExitCode
If ($ErrCode -eq 0) {
    $FeatureChange = Confirm-Feature -FeatureName $Name -
FeatureState $State
    If ($FeatureChange -eq $true) {
        If ($State -eq 'Enable') {
            Write-Host "Enabled" -
ForegroundColor Yellow
        } else {
            Write-Host "Disabled" -
ForegroundColor Yellow
        }
    } else {
        write-Host "Failed" -ForegroundColor Red
    }
} elseif ($ErrCode -eq 3010) {
    $FeatureChange = Confirm-Feature -FeatureName $Name -
FeatureState $State
    If ($FeatureChange -eq $true) {
        If ($State -eq 'Enable') {
            Write-Host "Enabled & Pending
Reboot" -ForegroundColor Yellow
        } else {
            Write-Host "Disabled & Pending
Reboot" -ForegroundColor Yellow
        }
    } else {
        write-Host "Failed" -ForegroundColor Red
    }
} else {
    If ($ErrCode -eq 50) {
        Write-Host "Failed. Parent feature needs to
be enabled first." -ForegroundColor Red
    } else {
        Write-Host "Failed with error code "$ErrCode
-ForegroundColor Red
    }
}
}

function Set-FeaturesFromFile {
<#
    .SYNOPSIS

```

```

        Set multiple features from a text file

        .DESCRIPTION
        This function reads the comma separated features and
values from a text file and executes each feature.

        .NOTES
        Additional information about the function.
#>

        [CmdletBinding()]
        param ()

parent) + '\',
        $RelativePath = (split-path $SCRIPT:MyInvocation.MyCommand.Path -
        $FeaturesFile = $RelativePath + $FeaturesFile
        If ((Test-Path $FeaturesFile) -eq $true) {
            $FeaturesFile = Get-Content $FeaturesFile
            foreach ($Item in $FeaturesFile) {
                $Item = $Item.split(',')
                Set-WindowsFeature -Name $Item[0] -State
$Item[1]
            }
        }
    }

Clear-Host
If ($ListFeatures -eq $true) {
    $WindowsFeatures = Get-WindowsFeaturesList
    $WindowsFeatures
}
If ($FeaturesFile -ne '') {
    Set-FeaturesFromFile
}
If ($Feature -ne '') {
    Set-WindowsFeature -Name $Feature -State $Setting
}
#Set-WindowsFeature -Name 'RSATClient-Features' -State 'disable'
#Set-WindowsFeature -Name 'RSATClient-ServerManager' -State 'disable'

```

Chapter 18

Easily Restore a Deleted Active Directory User with PowerShell

By: Thomas Rayner – MVP

If you have a modern version of Active Directory, you have the opportunity to enable the Active Directory Recycle Bin. Once enabled, you have a chance to recover a deleted item once it has been removed from Active Directory.

Here's a quick and easy script to recover a user based on their username.

```
$dn = (Get-ADObject -SearchBase (get-addomain).deletedobjectscontainer -  
IncludeDeletedObjects -filter "samaccountname -eq  
'$Username'").distinguishedname  
Restore-ADObject -identity $dn
```

On the first line, we're getting the DistinguishedName for the deleted user. The DN changes when a user gets deleted because it's in the Recycle Bin now. Where's your deleted objects container?

Well it's easily found with the **(Get-ADDomain).DeletedObjectsContainer** part of line 1. All we're doing is searching for AD objects in the deleted objects container whose username matches the one we're looking for. We need to make sure the **-IncludeDeletedObjects** flag is set or nothing that's deleted will be returned.

On the second line, we're just using the **Restore-ADObject** cmdlet to restore the object at the DN we found above.

Chapter 19

Getting Large Exchange Mailbox Folders with PowerShell

By: Thomas Rayner – MVP

I've been continuing my quest to identify users who have large Exchange mailboxes. I wrote a function in my last post to find large Exchange mailboxes, but, I wanted to take this a step further and identify the large folders within user mailboxes that could stand to be cleaned out. For instance, maybe I want to find all the users who have a large Deleted Items folder or Sent Items or Calendar. You get the idea. It's made to be run from a Remote Exchange Management Shell connection instead of by logging into an Exchange server via remote desktop and running such a shell manually. Remote administration is the future (just like my last post)!

So, let's define the function and parameters.

```
function Get-LargeFolder
{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $False)]
        [ValidateSet('All', 'Calendar', 'Contacts', 'ConversationHistory',
        'DeletedItems', 'Drafts', 'Inbox', 'JunkEmail', 'Journal',
        'LegacyArchiveJournals', 'ManagedCustomFolder', 'NonIpmRoot', 'Notes', 'Outbox',
        'Personal', 'RecoverableItems', 'RssSubscriptions', 'SentItems', 'SyncIssues',
        'Tasks')]
        [string]$FolderScope = 'All',
        [Parameter(Mandatory = $False)]
        [int]$Top = 1,
        [Parameter(Mandatory = $False,
        Position = 1,
        ValueFromPipeline = $True)]
        [string]$Identity = '*'
    )
}
```

My function is going to be named **Get-LargeFolder** and takes three parameters. `$FolderScope` is used in the **Get-MailboxFolderStatistics** cmdlet (spoiler alert) and must belong to the set of values specified. `$Top` is an integer used to define how many results we're going to return and `$Identity` can be specified as an individual username to examine a specific mailbox, or left blank (defaulted to `*`) to examine the entire organization.

```
function Get-LargeFolder
{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $False)]
        [ValidateSet('All', 'Calendar', 'Contacts', 'ConversationHistory',
        'DeletedItems', 'Drafts', 'Inbox', 'JunkEmail', 'Journal',
        'LegacyArchiveJournals', 'ManagedCustomFolder', 'NonIpMRoot', 'Notes', 'Outbox',
        'Personal', 'RecoverableItems', 'RssSubscriptions', 'SentItems', 'SyncIssues',
        'Tasks')]
        [string]$FolderScope = 'All',
        [Parameter(Mandatory = $False)]
        [int]$Top = 1,
        [Parameter(Mandatory = $False,
        Position = 1,
        ValueFromPipeline = $True)]
        [string]$Identity = '*'
    )

    Get-Mailbox -Identity $Identity -ResultSize Unlimited |
    Get-MailboxFolderStatistics -FolderScope $FolderScope
}

```

Now I've added a couple lines to get all the mailboxes in my organization (or a specific user's mailbox) which I pipe into a Get-MailboxFolderStatistics command with the FolderScope parameter set to the same value we passed to our function. Now we need to sort the results, but, see my last post for why that's going to be complicated.

```
function Get-LargeFolder
{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $False)]
        [ValidateSet('All', 'Calendar', 'Contacts', 'ConversationHistory',
        'DeletedItems', 'Drafts', 'Inbox', 'JunkEmail', 'Journal',
        'LegacyArchiveJournals', 'ManagedCustomFolder', 'NonIpMRoot', 'Notes', 'Outbox',
        'Personal', 'RecoverableItems', 'RssSubscriptions', 'SentItems', 'SyncIssues',
        'Tasks')]
        [string]$FolderScope = 'All',
        [Parameter(Mandatory = $False)]
        [int]$Top = 1,
        [Parameter(Mandatory = $False,
        Position = 1,
        ValueFromPipeline = $True)]
        [string]$Identity = '*'
    )

    Get-Mailbox -Identity $Identity -ResultSize Unlimited |
    Get-MailboxFolderStatistics -FolderScope $FolderScope |
    Sort-Object -Property @{
        e = {
            $_.Foldersize.split('(').split(' ')[-2].replace(',','') -as [double]
        }
    } -Descending
}

```

The FolderSize parameter that comes back with a Get-MailboxFolderStatistics cmdlet is a string which I'm splitting up in order to get back only the value in bytes which I am casting to a double. Now that we have gathered our stats and put them in order, I just need to select them so they may be returned. Here is the complete script.

```
function Get-LargeFolder
{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $False)]
        [ValidateSet('All', 'Calendar', 'Contacts', 'ConversationHistory',
        'DeletedItems', 'Drafts', 'Inbox', 'JunkEmail', 'Journal',
        'LegacyArchiveJournals', 'ManagedCustomFolder', 'NonIpmRoot', 'Notes', 'Outbox',
        'Personal', 'RecoverableItems', 'RssSubscriptions', 'SentItems', 'SyncIssues',
        'Tasks')]
        [string]$FolderScope = 'All',
        [Parameter(Mandatory = $False)]
        [int]$Top = 1,
        [Parameter(Mandatory = $False,
        Position = 1,
        ValueFromPipeline = $True)]
        [string]$Identity = '*'
    )

    Get-Mailbox -Identity $Identity -ResultSize Unlimited |
    Get-MailboxFolderStatistics -FolderScope $FolderScope |
    Sort-Object -Property @{
        e = {
            $_.FolderSize.split('(').split(' ')[-2].replace(',','') -as [double]
        }
    } -Descending |
    Select-Object -Property @{
        l = 'NameFolder'
        e = {
            $_.Identity.Split('/')[-1]
        }
    },
    @{
        l = 'FolderSize'
        e = {
            $_.FolderSize.split('(').split(' ')[-2].replace(',','') -as
[double]
        }
    } -First $Top
}
```

Now you can do this.

```
#Get 25 largest Deleted Items folders in your organization
Get-LargeFolder -FolderScope 'DeletedItems' -Top 25

#Get my largest 10 folders
Get-LargeFolder -Identity ThmsRynr -Top 10

#Get the top 25 largest Deleted Items folder for users in a specific group
$arrLargeDelFolders = @(
(Get-ADGroupMember 'GroupName' -Recursive).SamAccountName | ForEach-Object -
Process {
```

```
    $arrLargeDelFolders += Get-LargeFolder -FolderScope 'DeletedItems' -Identity
$}_
}
$arrLargeDelFolders |
Sort-Object -Property FolderSize -Descending |
Select-Object -Property NameFolder, @{
    l = 'FolderSize (Deleted Items)'
    e = {
        '{0:N0}' -f $_.FolderSize
    }
} -First 25 |
Format-Table -AutoSize
```


Chapter 20

Getting your Organizations Largest Exchange Mailboxes with PowerShell

By: Thomas Rayner – MVP

In a quest to hunt down users with large mailboxes, I wrote the following PowerShell function. It's made to be run from a Remote Exchange Management Shell connection instead of by logging into an Exchange server via remote desktop and running such a shell manually. Remote administration is the future!

My requirements were rather basic. I wanted a function that would return the top 25 (or another number of my choosing) Exchange mailboxes in my organization by total size. I also wanted the ability to specify an individual user's mailbox to see how large the specific box is.

So, let's get started.

```
function Get-LargeMailbox
{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $False)]
        [int]$Top = 1,
        [Parameter(Mandatory = $False,
            Position = 1,
            valueFromPipeline = $True)]
        [string]$Identity = '*'
    )
}
```

All I've done here is declare my new function named Get-LargeMailbox and specified its parameters. \$Top is the integer representing the number of mailboxes to return (defaulted to 1) and \$Identity is the specific mailbox we want to return (defaulted to * which will return all mailboxes).

Now, I know I need to get my mailboxes and retrieve some statistics.

```
function Get-LargeMailbox
```

```

{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $False)]
        [int]$Top = 1,
        [Parameter(Mandatory = $False,
            Position = 1,
            ValueFromPipeline = $True)]
        [string]$Identity = '*'
    )

    Get-Mailbox -Identity $Identity -ResultSize Unlimited |
    Get-MailboxStatistics
}

```

So far, so good. We haven't narrowed down the stats we care about yet but we're getting all the mailboxes in the organization and retrieving all the stats for them. Now we're about to run into a problem. There's a property returned by Get-MailboxStatistics called TotalItemSize but when you're in a remote session, but, it's hard to work with. Observe.

```
PS C:\> (Get-Mailbox ThmsRynr | Get-MailboxStatistics).TotalItemSize | Format-List
```

```

IsUnlimited : False
Value      : 2.303 GB (2,473,094,022 bytes)

```

You can see it returns a property consisting of a boolean value for if my quota is unlimited, and then a value of what my total size is. Ok, so that value is probably a number, right?

```
PS C:\> (Get-Mailbox ThmsRynr | Get-MailboxStatistics).TotalItemSize.value | Get-Member
```

```

    TypeName: Deserialized.Microsoft.Exchange.Data.ByteQuantifiedSize
#output omitted

```

Well, yeah, it is. The Value of TotalItemSize is a number but it's a *Deserialized.Microsoft.Exchange.Data.ByteQuantifiedSize* and when you're connected to a remote Exchange Management Shell, you don't have that library loaded unless you install some tools on your workstation. Rather than do that, can't we just fool around with it a bit and avoid installing a bunch of superfluous Exchange management tools? I bet we can, especially since this value has a ToString() method associated with it.

Back to our function. I need to sort the results of my "Get all the mailboxes, get all their stats" command by the total size of the mailboxes.

```
function Get-LargeMailbox
```

```

{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $False)]
        [int]$Top = 1,
        [Parameter(Mandatory = $False,
            Position = 1,
            ValueFromPipeline = $True)]
        [string]$Identity = '*'
    )

    Get-Mailbox -Identity $Identity -ResultSize Unlimited |
    Get-MailboxStatistics |
    Sort-Object -Property @{
        e = {
            $_.TotalItemSize.Value.ToString().split('(').split(' ')[-
2].replace(',', '').replace(' ', '') -as [double]
        }
    } -Descending
}

```

Oh boy, string manipulation is always fun, isn't it? What I've done here is sorted my mailboxes by an expression. That expression is the result of converting the value of the TotalItemSize attribute to a string and manipulating it. I'm splitting it on the open bracket character, and then again on the space character. I'm taking the second last item in that array, stripping out the commas and casting it as a double (because some values are too big to be integers). That's a lot of weird string manipulation for some of you to get your heads around, but look at the string returned by default. I need the number of bytes and that was the best way to get it.

Now all I need to do is select the properties from my sorted list of mailboxes and return the top number of results. Here's the final function.

```

function Get-LargeMailbox
{
    [CmdletBinding()]
    param (
        [Parameter(Mandatory = $False)]
        [int]$Top = 1,
        [Parameter(Mandatory = $False,
            Position = 1,
            ValueFromPipeline = $True)]
        [string]$Identity = '*'
    )

    Get-Mailbox -Identity $Identity -ResultSize Unlimited |
    Get-MailboxStatistics |
    Sort-Object -Property @{
        e = {
            $_.TotalItemSize.Value.ToString().split('(').split(' ')[-
2].replace(',', '').replace(' ', '') -as [double]
        }
    } -Descending |
    Select-Object -Property DisplayName, @{

```

```
l = 'MailboxSize'  
e = {  
    $_.TotalItemSize.Value.ToString().split('(').split(' ')[-  
2].replace(',','') -as [double]  
}  
} -First $Top  
}
```

Now you can do things like this.

```
#See how big my individual mailbox is  
Get-LargeMailbox -Identity ThmsRynr  
  
#Get the largest 20 mailboxes in the organization  
Get-LargeMailbox -Top 20  
  
#Get the mailboxes for a specific AD group and sort by size  
$arrLargeMailboxes = @()  
(Get-ADGroupMember 'GroupName' -Recursive).SamAccountName | ForEach-Object -  
Process {  
    $arrLargeMailboxes += Get-LargeMailbox -Identity $_  
}  
$arrLargeMailboxes |  
Sort-Object -Property MailboxSize -Descending |  
Select-Object -Property DisplayName, @{  
    l = 'MailboxSize'  
    e = {  
        '{0:N0}' -f $_.MailboxSize  
    }  
} |  
Format-Table -AutoSize
```

Before we end, let's take a closer look at the last example.

First, I'm declaring an array to hold the results of users and how large their mailbox is. Then I'm getting all the members of a group, taking the SamAccountName and performing an action on each of them. That action, of course, is retrieving their mailbox size using the function I just wrote and appending the results to the array. Then I need to sort that array and display it. The Select-Object command has the formatting I included to make the mailbox sizes have commas separating every three digits.

Chapter 21

Just Enough Administration (JEA) First Look

By: Thomas Rayner – MVP

If you're reading this, it means that Windows Server 2016 Technical Preview 4 is released (currently available on MSDN) and one of the new features that's available is Just Enough Administration (JEA)! Until now, you could use DSC to play with JEA but now it's baked into Windows Server 2016.

If you're not sure what JEA is or does, check out this page published by Microsoft.

<https://msdn.microsoft.com/en-us/library/dn896648.aspx>

So how do you get started?

JEA gets put together like a module. There are a bunch of different ways to dive in, but for convenience, I'm just covering this one example. Build on it and learn for yourself how JEA can work for you specifically!

First things first, make a new directory in your modules folder and navigate to it.

```
$dir = 'C:\windows\system32\windowsPowerShell\v1.0\Modules\JEA-Test'  
new-item -itemtype directory -path $dir  
cd $dir
```

So far, so easy. Now, we're going to use the brand new JEA cmdlets to configure what is basically our constrained endpoint.

```
New-PSSessionConfigurationFile -path "$dir\JEA-Test.pssc"
```

This PSSC is the first of two files we’re going to make. It’s a session config file that specifies the role mappings (we’ll get to roles in a second) and some other general config settings. A PSSC file looks like this.

```
@{
# Version number of the schema used for this document
SchemaVersion = '2.0.0.0'

# ID used to uniquely identify this document
GUID = 'c433f896-4241-4b12-b857-059a395c2d2b'

# Author of this document
Author = 'trayner'

# Description of the functionality provided by these settings
# Description = ''

# Session type defaults to apply for this session configuration. Can be
'RestrictedRemoteServer' (recommended), 'Empty', or 'Default'
SessionType = 'RestrictedRemoteServer'

# Directory to place session transcripts for this session configuration
# TranscriptDirectory = 'C:\Transcripts\'

# Whether to run this session configuration as the machine's (virtual)
administrator account
# RunAsVirtualAccount = $true

# Groups associated with machine's (virtual) administrator account
# RunAsVirtualAccountGroups = 'Remote Desktop Users', 'Remote Management Users'

# Scripts to run when applied to a session
# ScriptsToProcess = 'C:\ConfigData\InitScript1.ps1',
'C:\ConfigData\InitScript2.ps1'

# User roles (security groups), and the role capabilities that should be applied
to them when applied to a session
RoleDefinitions = @{ 'mvp-trayner\test users' = @{ RoleCapabilities = 'testers'
} }
}
```

If you’ve ever authored a PowerShell module before, this should look familiar. There’s only a few things you need to do here. The first is change the value for *SessionType* to *RemoteRestrictedServer*. You need to make it this in order to actually restrict the user connections.

You can enable *RunAsVirtualAccount* if you’re on an Active Directory Domain. I won’t get too deep into what virtual accounts do because my example is just on a standalone server.

The other important task to do is define the *RoleDefinitions* line. This is a hashtable where you set a group (in my case, local to my server) assigned to a “RoleCapability”. In this case, the role I’m assigning is just named “testers” and the local group on my server is named “test users”.

Save that and now it's time to make a new directory. Roles **must** be in a "RoleCapabilities" folder within your module.

```
new-item -itemtype directory "$dir\RoleCapabilities"
```

Now we are going to continue using our awesome new JEA cmdlets to create a PowerShell Role Capabilities file.

```
New-PSRoleCapabilityFile -path "$dir\RoleCapabilities\testers.psrc"
```

Note: It's very important to note here that the name of my PSRC file is the same as the RoleCapability that I assigned in the PSSC file above.

PSRC files look like this. Let's point out some of the key areas in this file and some of the tools you now have at your disposal.

Think of a PSRC as a giant white list. If you don't explicitly allow something, it's not going to happen. Because PSRCs all act as white lists, if you have users who are eligible for more than one PSRC (through more than one group membership/role assignment in a PSSC), the access a user gets is everything that's white listed by any role the user is eligible for. That is to say, PSRCs merge if users have more than one that apply.

```
@{  
  
# ID used to uniquely identify this document  
GUID = '3e2ca105-db93-4442-acfd-037593c6c644'  
  
# Author of this document  
Author = 'trayner'  
  
# Description of the functionality provided by these settings  
# Description = ''  
  
# Company associated with this document  
CompanyName = 'Unknown'  
  
# Copyright statement for this document  
Copyright = '(c) 2015 trayner. All rights reserved.'  
  
# Modules to import when applied to a session  
# ModulesToImport = 'MyCustomModule', @{ ModuleName = 'MyCustomModule';  
ModuleVersion = '1.0.0.0'; GUID = '4d30d5f0-cb16-4898-812d-f20a6c596bdf' }  
  
# Aliases to make visible when applied to a session  
# VisibleAliases = 'Item1', 'Item2'  
  
# Cmdlets to make visible when applied to a session  
VisibleCmdlets = 'Get-*', 'Measure-*', 'Select-Object', @{ Name = 'New-Item';  
Parameters = @{ Name = 'ItemType'; ValidateSet = 'Directory' }, @{ Name =  
'Force' }, @{ Name = 'Path'; ValidateSet = 'C:\Users\testguy\ONLYthis' } }
```

```
# Functions to make visible when applied to a session
# VisibleFunctions = 'Invoke-Function1', @{ Name = 'Invoke-Function2';
Parameters = @{ Name = 'Parameter1'; ValidateSet = 'Item1', 'Item2' }, @{ Name =
'Parameter2'; ValidatePattern = 'L*' } }

# External commands (scripts and applications) to make visible when applied to a
session
VisibleExternalCommands = 'c:\scripts\this.ps1'

# Providers to make visible when applied to a session
# VisibleProviders = 'Item1', 'Item2'

# Scripts to run when applied to a session
# ScriptsToProcess = 'C:\ConfigData\InitScript1.ps1',
'C:\ConfigData\InitScript2.ps1'

# Aliases to be defined when applied to a session
# AliasDefinitions = @{ Name = 'test-alias'; Value = 'Get-ChildItem' }

# Functions to define when applied to a session
# FunctionDefinitions = @{ Name = 'MyFunction'; ScriptBlock = { param($MyInput)
$MyInput } }

# Variables to define when applied to a session
# VariableDefinitions = @{ Name = 'Variable1'; Value = { 'Dynamic' +
'InitialValue' } }, @{ Name = 'Variable2'; Value = 'StaticInitialValue' }

# Environment variables to define when applied to a session
# EnvironmentVariables = @{ Variable1 = 'Value1'; Variable2 = 'value2' }

# Type files (.ps1xml) to load when applied to a session
# TypesToProcess = 'C:\ConfigData\MyTypes.ps1xml',
'C:\ConfigData\OtherTypes.ps1xml'

# Format files (.ps1xml) to load when applied to a session
# FormatsToProcess = 'C:\ConfigData\MyFormats.ps1xml',
'C:\ConfigData\OtherFormats.ps1xml'

# Assemblies to load when applied to a session
# AssembliesToLoad = 'System.Web', 'System.OtherAssembly, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a'

}
```

Let's skip ahead to line 25. What I'm doing here is white listing any cmdlet that starts with **Get-** or **Measure-** as well as **Select-Object**. Inherently, any of the parameters and values for the parameters are whitelisted, too. I can hear you worrying, though. "What if a Get- command contains a method that allows you to write or set data? I don't want that!" Well, rest assured. JEA runs in No Language mode which prevents users from doing any of those shenanigans.

Also in line 25, I'm doing something more specific. I'm including a hashtable. Why? Because I want to allow the **New-Item** cmdlet but only certain parameters and values. I'm allowing the **ItemType** parameter but only if the user sets it to **Directory**. I'm allowing **Force**, which doesn't take a value. I'm also allowing the **Path** attribute, but, only a specific path. If a user tries to use the **New-Item** cmdlet but violates these rules, the user will get an error.

On line 19, I can import specific modules without opening up the **Import-Module** cmdlet. These modules are automatically imported when the session starts.

On line 28, we can make specific functions available to connecting users.

Line 31 is interesting. Here I'm making an individual script available to the connecting user. The script contains a bunch of commands that I haven't white listed, so, is the user going to be able to run it? Yes. Yes they are. The user can run that script and the script will run correctly (assuming other permissions are in place) without having the individual cmdlets white listed. **It is a bad idea to allow your restricted users to write over scripts you make available to them this way.**

On line 37, you can basically configure a login script. Line 40 lets you define custom aliases and line 43 lets you define custom functions that only exist in these sessions. Line 46 is for defining custom variables (like "\$myorg = 'ThmsRynr Co.')" which can be static or dynamic.

With these tools at your disposal, you can configure absolutely anything about a user's session and experience. Sometimes, you might have to use a little creativity, but anything is possible here.

Lastly, you need to set up the JEA endpoint. You can also overwrite the default endpoint so every connection hits your JEA config but you may want to set up another unconstrained endpoint just for admins... just in case.

```
Register-PSSessionConfiguration -name 'JEA-Test' -path $dir
```

That's it. You're done. Holy, that was way too easy for how powerful it is. Now when a user wants to connect, they just run a command like this and they're in a session limited like you want.

```
Enter-PSSession -ComputerName mvp-trayner -ConfigurationName JEA-Test
```

I can think of a bunch of use cases for JEA. For instance...

Network Admins

I'd like my network admins to be able to administer DHCP and DNS on our Windows servers which hold these roles without having carte blanche admin rights to everything else. I think this would involve limiting the cmdlets available to those including *DHCP* or *DNS*.

Certificate Management

We use the PSPKI module for interacting with our Enterprise PKI environment. For this role, I'd deploy the module and give users permissions to use only the PSPKI cmdlets. I'd use the Windows CA permissions/virtual groups to allow or disallow users manage CA, manage certificates, or just request certificates.

Code Promotion

Allowing people connecting via JEA to read/write only certain areas of a filesystem isn't practical. The way I'd get around this is to allow access to run only one script which performed the copy commands or prompted for additional info as required. You could mix this in with PowerShell Direct and promote code to a server in a DMZ without opening network holes or allowing admin access to a DMZ server.

Service Account for Patching

We have a series of scripts that apply a set of rules and logic to determine if a server needs to be patched or not. All it needs to do is perform some WMI queries, communicate with SCCM (which has the service installed to actually do the patching) and reboot the server. Instead, right now, that service account has full admin rights on the server.

Help Desk

Everybody's help desk is different but one job I'd like to send to my help desk is some limited Active Directory management. I'd auto-load the AD module and then give them access to very restricted cmdlets and some parameters. For instance, Get-ADUser and allow -Properties but only allow the memberof, lockedout, enabled and passwordlastset values. I might also allow them to add users to groups but only if the group was in a certain OU or matched a certain string (ie: if the group ends in "distribution list").

Print Operators

We have a group of staff on-site 24/7 that service a giant high speed print device. There are a number of servers that send it jobs and many are sensitive. I'd like to give the print operators group permissions to reach out and touch these servers only for the purposes of managing print jobs.

Hyper-V Admins/Host Management

These guys need the Hyper-V module and commands within it as well as some limited rights on the host, like Get WMI/CIM objects but not the ability to set WMI/CIM objects.

Get playing!

The possibilities of what you can do with JEA are endless. While the DevOps mentality is flourishing, the need to enable access to different systems is growing. With JEA, you can enable whatever kind of access you need, without enabling a whole bunch of access you don't. That's probably why it's called "Just Enough Administration".

Chapter 22

Does a String Start or End in a Certain Character?

By: Thomas Rayner – MVP

Can you tell in PowerShell if a string ends in a specific character, or if it starts in one? Of course you can. Regex to the rescue!

It's a pretty simple task, actually. Consider the following examples:

```
'something\' -match '.*?\\$'  
#returns true  
  
'something' -match '.*?\\$'  
#returns false  
  
'\something' -match '^\\.?.*?'  
#returns true  
  
'something' -match '^\\.?.*?'  
#returns false
```

In the first two examples, I'm checking to see if the string ends in a backslash. In the last two examples, I'm seeing if the string starts with one. The regex pattern being matched for the first two is `.*?\\$`. What's that mean? Well, the first part `.*?` means "any character, and as many of them as it takes to get to the next part of the regex. The second part `\\` means "a backslash" (because `\` is the escape character, we're basically escaping the escape character. The last part `$` is the signal for the end of the line. Effectively what we have is "anything at all, where the last thing on the line is a backslash" which is exactly what we're looking for. In the second two examples, I've just moved the `\\` to the start of the line and started with `^` instead of ending with `$` because `^` is the signal for the start of the line.

Now you can do things like this.

```
$dir = 'c:\temp'  
if ($dir -notmatch '.*?\\$')  
{
```

```
    $dir += '\'  
  }  
$dir  
#returns 'c:\temp\'
```

Here, I'm checking to see if the string 'bears' ends in a backslash, and if it doesn't, I'm appending one.

Cool, right?

Chapter 23

Using PowerShell to List All the Fonts in a Word Document

By: Thomas Rayner – MVP

Recently I was challenged by a coworker to use PowerShell to list all the fonts in a Word document. It turned out to be easier than I thought it would be... but also slower than I thought it would be. Here's what I came up with.

```
$word = New-Object -ComObject word.Application
$OpenDoc = $word.Documents.open('c:\temp\test.docx')
$OpenDoc.words | % { $_ | select -ExpandProperty font } | select Name -Unique
$OpenDoc.close()
$word.quit()
```

There could very well be a better way of doing this but this is what I came up with in a hurry. Line 1 declares a new instance of Word and line 2 opens the document we're looking at. Then, for each word (which is handily a property of the open word document), we're expanding the font property and selecting all the unique names of the fonts on line 3. Lines 4 and 5 close the document and quit Word.

So, you can get something like this!

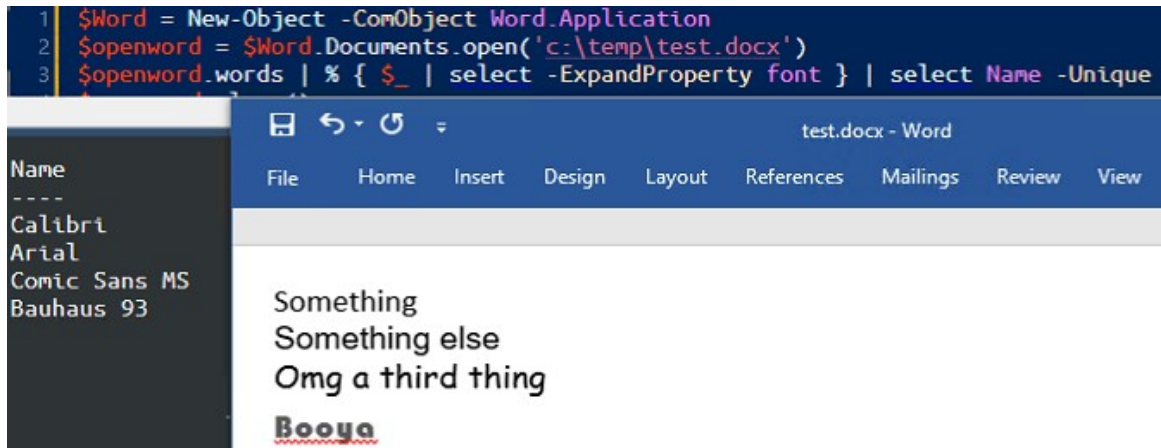


Figure 24 – Showing off our fonts from a word document using PowerShell

Chapter 24

Find PowerShell Scripts in the GUI

By: Sean Kearney – MVP

A lot of people currently use Active Directory Users and Computers. It's very fast and efficient. However, if you haven't looked at the Active Directory Administration Center in Server 2012, you're missing out on a gold mine!

This particular tool passes everything through PowerShell while you still work in the GUI. Here's an example. Let's pretend that my boss said, "Hey, who in our network has domain admin access?"

In the Active Directory, Administrative Center, I can search for a property in the Global Search context. Here I search for the group domain admins.

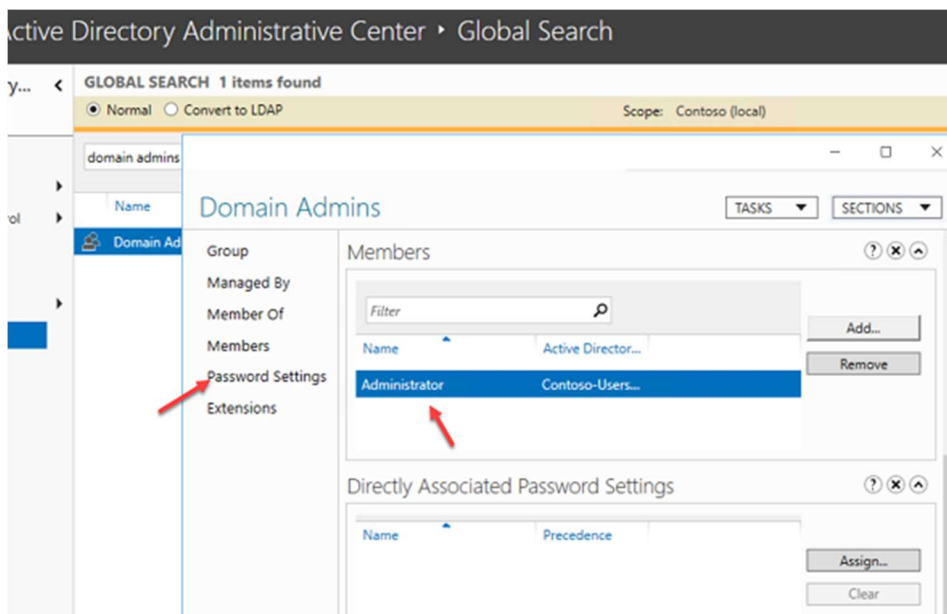


Figure 25 – Looking at the Active Directory Administrative Center

When I press Enter on the found object, I can see the properties of that object.

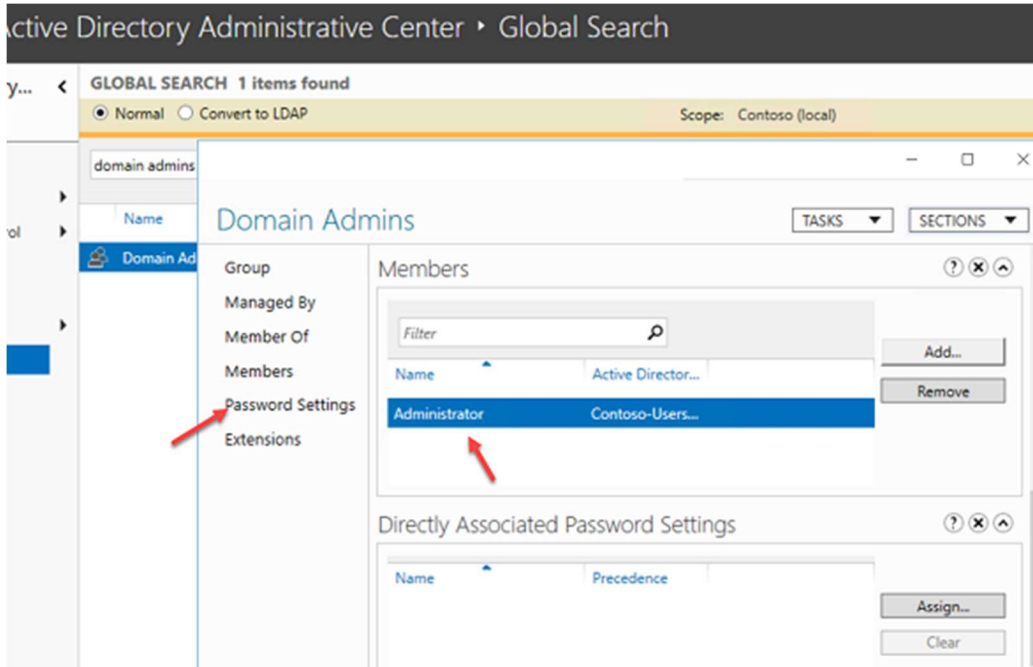


Figure 26 – Viewing Password settings on the Administrator Account

In this case, there is only one member. I’m going to show you something about PowerShell here. Every single action is logged as a PowerShell cmdlet. To view how this process was done in PowerShell, you need to look down on the bottom to see the words, **WINDOWS POWERSHELL HISTORY**. Click the little arrow to expand the window.



Figure 27 – Finding the Windows PowerShell History

This will expand a log window. You might not see anything until you select the Show All box.

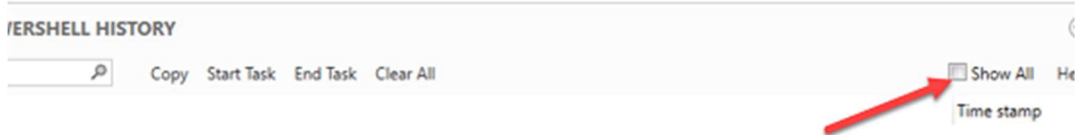


Figure 28 – Viewing a list of all of the previous commands that have run

Click the Clear All tab and do something basic in the console, such as viewing that same group again. When you do that, some information will appear in the window.

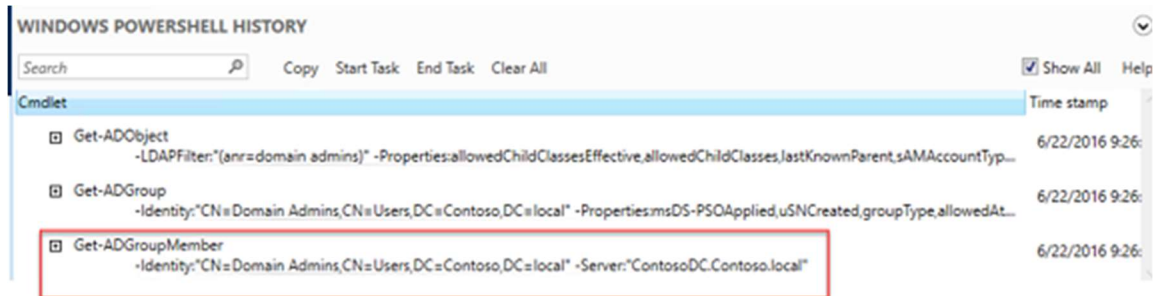


Figure 29 – Viewing the code

This is your search for the group in the console, followed by accessing the group, and then the group membership. This is the PowerShell code that did the work for you. If you'd like to grab the PowerShell code to play with or to use again, just click the code, and then choose your favorite “Copy All Combo” – CTRL-A, CTRL-C.

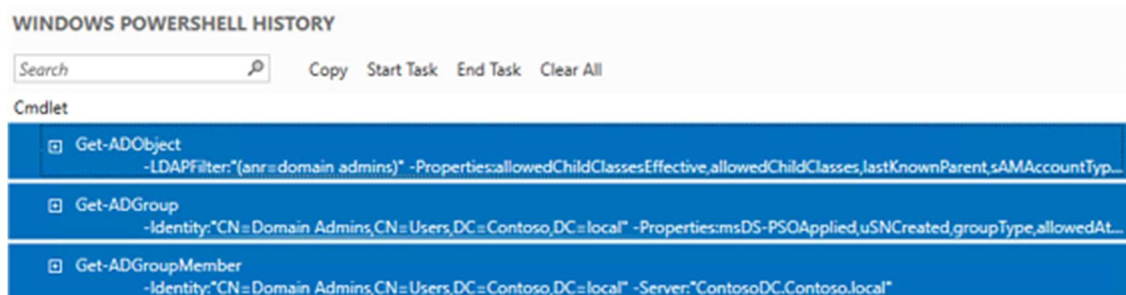


Figure 30 – Copying the PowerShell commands

You can then paste the code directly into Notepad for safekeeping. You can also paste each individual line into an open PowerShell console to see what each line does. This console alone is a pretty good way to use the GUI and get the PowerShell code to play with on your own.

In the following sample, the **Get-ADGroupMember** content from the console was pasted and run in a PowerShell console.

```
PS C:\> Get-ADGroupMember -Identity: CN=Domain Admins,CN=Users,DC=Contoso,DC=local -Server: ContosoDC.Contoso.local

distinguishedName : CN=Administrator,CN=Users,DC=Contoso,DC=local
name               : Administrator
objectClass        : user
objectGUID         : 9db2dd76-d748-473a-a7fa-e1ea49a6b33d
SamAccountName     : Administrator
SID                : S-1-5-21-1446109916-1062881444-1496387064-500
```

Figure 31 – Testing the PowerShell commands manually

This same technique works for almost anything you do in the Active Directory Administrative Center including disabling and unlocking accounts. It gives you the option to use PowerShell for some regular tasks without having to learn the code part.

Another great example is the Microsoft Deployment Toolkit, which is just ripe with automation in both VBScript and PowerShell. It offers a feature that you’ll often find in GUI-based applications from Microsoft.

It’s the mythical, magical View Script box that you may have been overlooking during your wizards.

Here in the Microsoft Deployment Toolkit, we’ll do a basic import of a driver from the C:\Drivers folder.

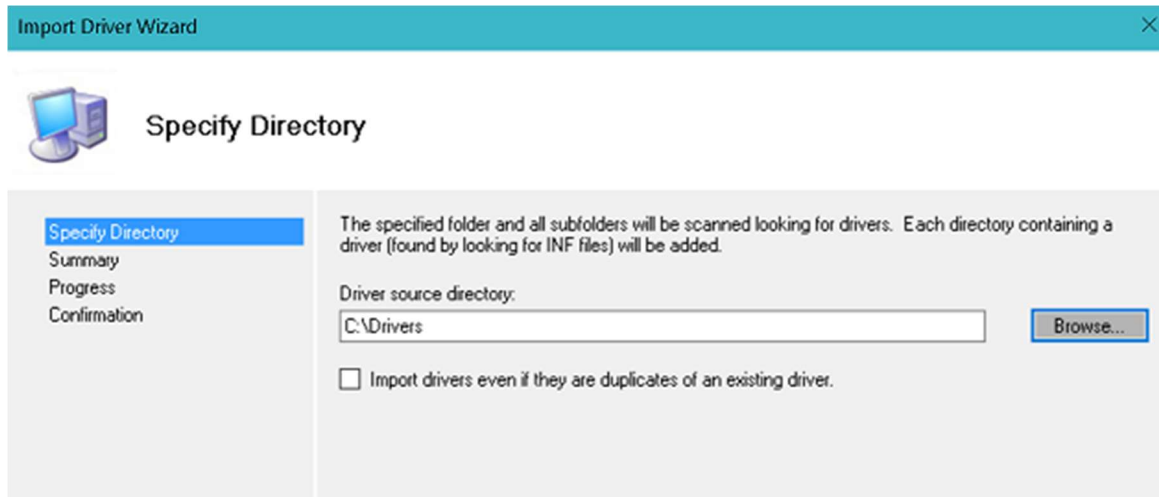


Figure 32– Configuring a task sequence in the Microsoft Deployment Toolkit

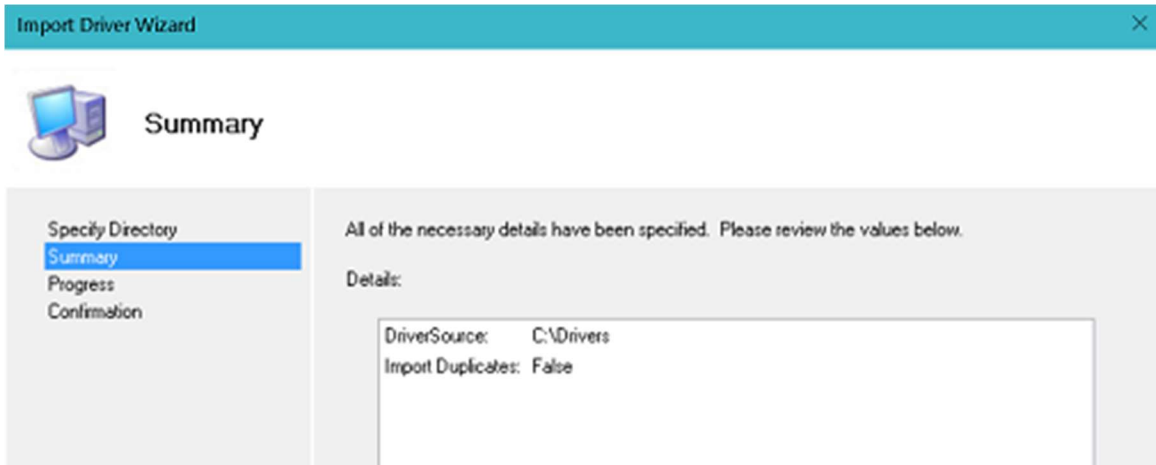


Figure 33 – Configuring a task sequence in the Microsoft Deployment Toolkit

As you continue, the wizard will import the drivers from the source folder to the Microsoft Deployment Toolkit. If this task were very long, a problem is that the process would tie up the Microsoft Deployment Toolkit console until it was done.

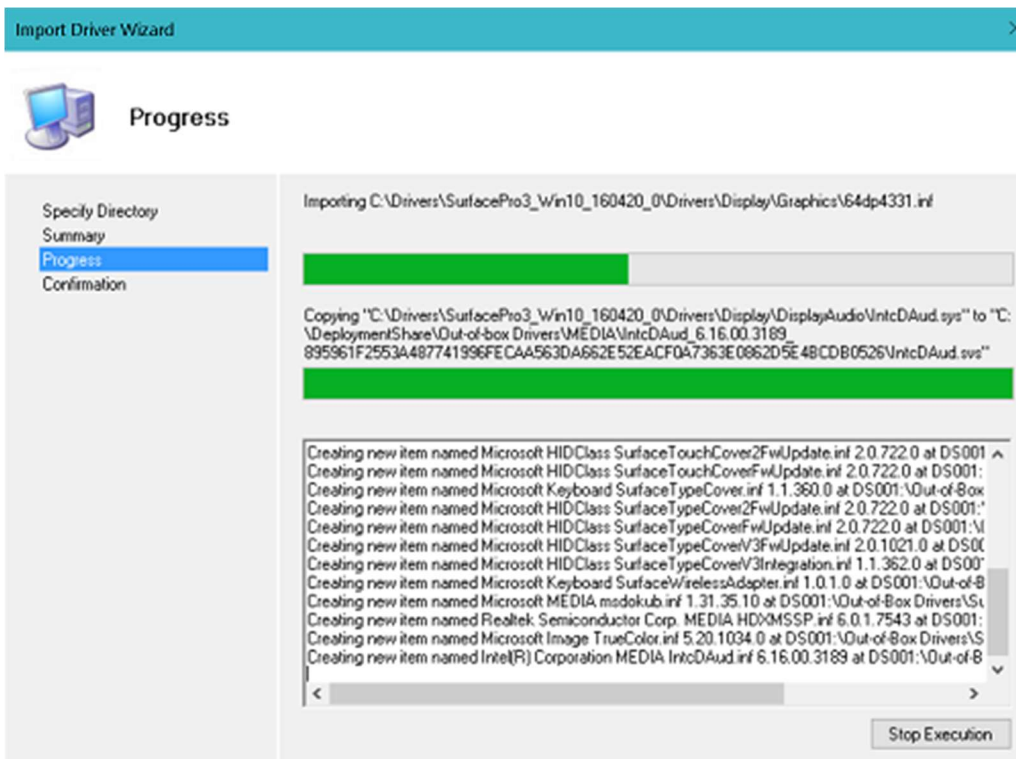


Figure 33 – Running a task sequence from the Microsoft Deployment Toolkit

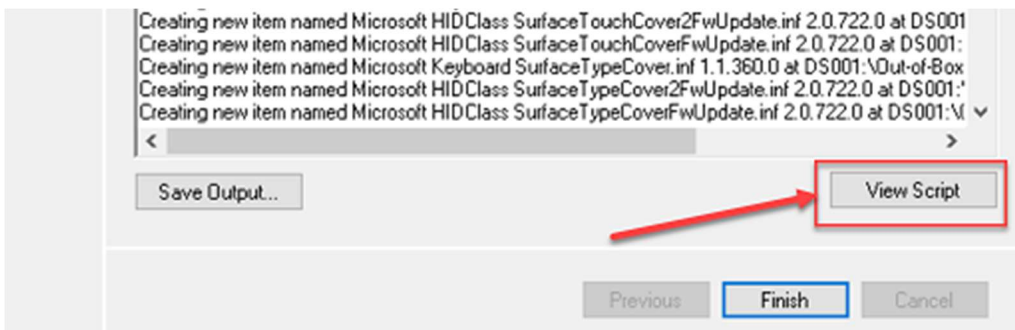


Figure 33 – Checking the View Script button in the Microsoft Deployment Toolkit

If you click View Script, you will see the actual PowerShell code that performed this operation. Here’s the example from the action where we imported the driver.

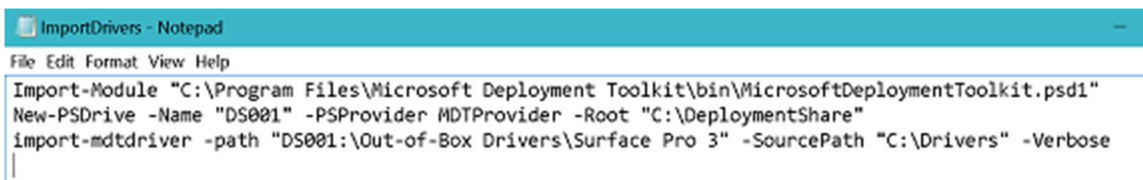


Figure 34 – Reviewing the code

So, even if you don’t know how to code in PowerShell, you can see the “C:\Drivers” folder. If you needed to add more drivers to the same structure, you could change “C:\Drivers” to the new source folder easily. If you just pasted the code into PowerShell right now, you’d notice something.

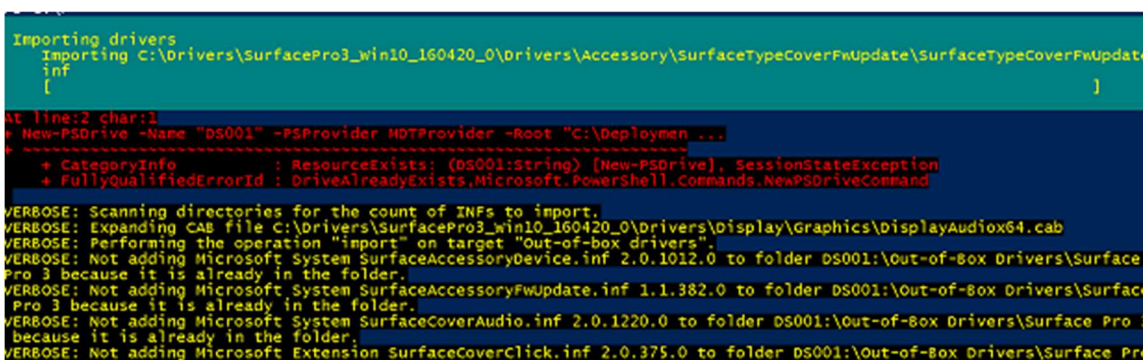


Figure 35 – Running the code manually

First, it does the action again and warns a lot about “Already existing.” The second and more critical piece is that the console is not tied up during this process.

Think about when you update media in the Microsoft Deployment Toolkit. Wouldn't this be handy? It also means that although you didn't learn how to script in PowerShell, you learned how you could make PowerShell useful for you.

Interesting thought, eh?

Other applications, such as System Center Virtual Machine Manager, do this as well. Keep your eyes out for various applications that contain View Script. You may just find some hidden gold in there!

Chapter 25

How to use PowerShell as an Administrative Console

By: Sean Kearney – MVP

I know that everyone has experienced that, especially on Friday. A handful of people come in and for whatever reason have locked themselves out.

We're not even going to get in to the root cause of this: the Shift key was stuck, there was a soft lock in the software, or their brains were sucked away by Martians.

Perhaps they were showing off their 'R@s K001 P@\$w0rd skillz' to a co-worker and incorrectly typed the last three punctuation marks in the password they invented.

But, you would almost always get an onslaught of "I locked myself out and NEED (not want ... capital N*E*E*D) to get back in now."

In the old world, I would have had to go to Active Directory Users and Computers, find the user, unlock the user, call the use, and then proceed to the next person.

But, it never works like that, does it? Invariably, some of those users (or maybe all on a bad day) will lock themselves out again in that same 10-minute period. Thus the phrase, "Lock out Hell", as you waste half an hour going through the GUI to unlock users.

People make mistakes, and there's no use getting angry with them. The process itself was just, well, lacking finesse.

I had been using PowerShell earlier that month to migrate users to a new Active Directory and had been playing with the Quest cmdlets at the time. One lunch hour, I played with PowerShell and found the **Unlock-QADUser** cmdlet.

After I found this cmdlet, the process was simply a matter of running something like the following for the five or six people:

```
Unlock-QADuser jsmith
```

Then I pressed an up arrow, entered the next name, and repeated the process for the next four or five. It was far less stressful. In the modern Windows Server 2008 R2 and higher environment, I would have used the following cmdlet:

```
Unlock-ADAccount
```

It accomplished the same result and, again, was far less aggravation for everyone...including me. Later on, I would have to start disabling users, often quietly and discreetly. Very much like “Shh...when you see your phone ring from the VP Disable Mr. X.” (Professor Xavier’s evil cousin, you see.)

For that, it was a matter of queuing up a cmdlet like **Disable-QADUser**. I would also correspondingly have an **Enable-QADUser** ready because sometimes it was actually Mrs. Y and not Mr. X so “Could you please quickly flip that around?”

As result, it was very easy to deal with those situations by using PowerShell just as a management console.

Later on in my IT life, we had to start producing some basic audits about who had Domain Admin and Enterprise Admin access. I already knew that I could use PowerShell to ask Active Directory questions, such as, “Show me all the members of this group.”

Auditors who saw the code that did the audit were very happy to see that it was PowerShell. The following cmdlet pulled up a list of Domain Admins and dumped it untouched to a CSV file.

```
Get-ADGroupMember -Identity 'Domain Admins' | Export-CSV DomainAdmins.csv
```

My environment actually had quite a few domains. Some were for the Development division, Vendor applications for a customer. Because of PowerShell, it was easy to pull the same data for any environment from the console.

As my work progressed into other environments, including managing my remote workstations, PowerShell was my view into systems. Using Windows Management Instrumentation (WMI), I would easily query a remote system for its serial number by using Get-Wmiobject:

```
(Get-wmiObject win32_bios -computername PC123).serialnumber
```

At the time, I didn’t have tools like System Center Configuration Manager. Our Division was pretty small. I still needed to manage systems and ask questions.

Those questions were readily answered by using PowerShell, even without scripts. It’s just some food for thought if you’re convincing yourself that PowerShell is only for scripters. Pop on in tomorrow. If you’re curious, I’ll show you some neat tricks that Microsoft offers to use PowerShell without actually have to learn it.

Strange concept, eh?

Chapter 26

How to Erase Files based on Date using PowerShell

By: Sean Kearney – MVP

You know the typical work for some field techs. You get in the car, go to the call, hop back in the car, off to the next call, lunch in a drive-thru, sleep in a parking lot. Also, factor in about five to six hours lost to commute.

Imagine a schedule like that for seven+ years and when do you have time to learn? I believed exactly the same thing about Windows PowerShell until I found myself in a position where I had to use it.

I clearly remember the day, too. It was sometime late in 2007. I got a call from a client. “There’s a vendor application filling up the server with logs. Please find some way to get rid of the old data. Anything over 24 hours old is garbage.”

“Okay,” I thought to myself. “I’m not a developer or coder, but maybe there’s a utility on the Internet, something from Sysinternals. If I have to spend two hours to learn how to write something, I’ll do it.”

Well, I got on the site, and the first thing that I did was hit the local search engines for “Erase old files based on date” (or something along those lines). I ran into an article on Hey Scripting Guys that showed how to remove files over seven (7) days old from a folder.

It showed this line in PowerShell to get a list of the files ending in .LOG:

```
Get-Childitem C:\Foldername\*.LOG
```

It then showed two additional lines to remove that content. One line grabs the current date, and the other filters the list.

```
$NOW=Get-Date  
Get-Childitem C:\Foldername\*.LOG | where-Object { $_.LastWriteTime -lt  
$NOW.AddDays(7) }
```


I didn't really understand much other than, "There's a folder, and this is supposed to show stuff over seven days old. I wonder if I just change that 7 to a 1?"

At that point, I installed PowerShell on a test machine and copied the folder structure that I had to purge. I then made a second copy because the network was slow that day, and I didn't want to waste time.

I edited the folder name to match the folder that I wanted and the file extension. I also changed that 7 to a 1 and crossed my fingers.

```
$Now=Get-Date
Get-Childitem C:\VendorApp\*.TXT | where-Object { $_.LastWriteTime -lt
$Now.AddDays(-1) }
```

I blinked and looked. It displayed only files that were older than today and that had the TXT extension. I was floored. To learn how to remove them, I was ready for a long, drawn-out session. "Grumble grumble...I'll bet VBScript is involved here somewhere."

To remove the files, I saw only one extra bit of data. Something called **Remove-Item** seemed a pretty sensible name. I appended the last bit of code and read a bit further as the article described how to test it without doing anything dangerous by using the **whatif** parameter.

I shook my head and muttered something about "Pigs flying before this would work," but I tried it anyway.

```
$Now=Get-Date
Get-Childitem C:\VendorApp\*.TXT | where-Object { $_.LastWriteTime -lt
$Now.AddDays(-1) } | Remove-Item -whatif
```

Staring and blinking at the screen of output, I scrolled up. It looked like something happened to the data. But, upon examining the folder, nothing happened. It really did a "What if I do this?"

To actually do the damage, it asked me to remove the **-whatif** and run the cmdlet.

```
$Now=Get-Date
Get-Childitem C:\VendorApp\*.TXT | where-Object { $_.LastWriteTime -lt
$Now.AddDays(-1) } | Remove-Item
```

“Wow!” Staring at the folder, I was speechless. Only 24 hours left of files ending in TXT. I had been on site for only 10 minutes, and the problem was solved....almost.

“I’ll bet this won’t work on the server,” I thought. I was ready for the worst. This was my first time using PowerShell and I was untrusting.

So, repeat process, install PowerShell 1.0 on the Windows Server 2003 R2 environment, copy this script, duplicate the folder of data. After all, I wasn’t about to see how good their backup system was.

I tested the process, and the results were the same. Now I had to figure out how to schedule this. Oh, this was like DOS. Instead of running cmd.exe or command.com, it was PowerShell.exe.

I made a little folder called “C:\DataClean”, saved this new script called LogClean.PS1, and set up a scheduled task.

```
PowerShell.exe -executionpolicy Bypass -file C:\DataClean\LogClean.PS1
```

I ran the task and crossed my fingers.

POOF! It all just worked!

I was on site for barely 20 minutes for a two-hour minimum billable call. I was also on the road early and beat rush hour traffic on a Friday.

Hearing AC/DC’s Highway to Hell on the radio and with the knowledge of what technology got me home early that day, I began singing loudly all the way home. Singing to the chorus of the song, the words “I’m using PowerShell! I’m using PowerShell!” kept bursting from my lips.

Yes, I love using PowerShell. Over the next while, I’ll try to relate some things that you can use PowerShell for without needing to learn how to script. Even if you, too, are that field tech with no time on your hands.

Chapter 27

Join us at MVPDays and meet great MVP's like this in person

If you liked their book, you will love to hear them in person.

Live Presentations

Dave frequently speaks at Microsoft conferences around North America, such as TechEd, VeeamOn, TechDays, and MVPDays Community Roadshow.

Cristal runs the MVPDays Community Roadshow.

You can find additional information on the following blog:

www.checkyourlogs.net

www.mvpdays.com

Video Training

For video-based training, see the following site:

www.mvpdays.com

Live Instructor-led Classes

Dave has been a Microsoft Certified Trainer (MCT) for more than 15 years and presents scheduled instructor-led classes in the US and Canada. For current dates and locations, see the following sites:

- www.truesec.com
- www.checkyourlogs.net

Consulting Services

Dave and Cristal have worked with some of the largest companies in the world and have a wealth of experience and expertise. Customer engagements are typically between two weeks and six months. For more information, see the following site:

www.triconelite.com and www.rsvccorp.com

Twitter

Dave, Cristal, Émile, Thomas, Allan, Sean, Mick, and Ed on Twitter tweet on the following aliases:

- Dave Kawula: @DaveKawula
- Cristal Kawula: @SuperCristal1
- Émile Cabot: @Ecabot
- Thomas Rayner: @MrThomasRayner
- Allan Rafuse: @AllanRafuse
- Sean Kearney: @EnergizedTech
- Mick Pletcher: @Mick_Pletcher
- Ed Wilson: @ScriptingGuy