Master PowerShell Tricks

Volume 3

Dave Kawula - MVP

Thomas Rayner - MVP

Allan Rafuse - MVP

Will Anderson - MVP

Mick Pletcher - MVP

Foreword by: Jeff Woolsey

PUBLISHED BY

MVPDays Publishing
http://www.mvpdays.com

ISBN: 978-1979733137

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.checkyourlogs.net or sending an email to feedback@mvpdays.com.

Foreword by: Jeff Woolsey

“PowerShell is awesome.” I’ve lost track of how many times I’ve heard this phrase and I never get tired of hearing it... 😊

I remember the launch of Windows Server 2012 like it was yesterday. It was a major release with innovation across the board in Hyper-V, Storage, Networking, Scale and Performance. Underlying all of this innovation was one unifying investment designed to make it easier for you to harness these technologies: PowerShell. With over 2500 PowerShell commandlets built-in, rich automation was now in everyone’s hands. Over the years, we listened as you used PowerShell for everything from simple repeatable tasks to complex deployments of servers, software and services where PowerShell removed human error.

A few releases later, we’ve added over a thousand new commandlets, Desired State Configuration (DSC), and Just Enough Administration (JEA) making PowerShell an indispensable and valuable skill for every résumé. If you haven’t learned PowerShell yet, there’s no time like the present. (Spoiler alert: It’s fun too…) The authors are PowerShell experts and MVPs who teach and coach in the Microsoft community with years of experience. In addition, they have used their learnings in the community to provide feedback to the PowerShell team and help influence the product direction. Whether you or a seasoned user or a PowerShell newbie, you can’t ever know too many PowerShell Tricks.

Jeff Woolsey

Windows Server/Hybrid Cloud

@wsv_guy

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, 1E, 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.

Recently Dave has been honored to take on the role of Conference Co-Chair of TechMentor with fellow MVP Sami Laiho. The lineup of speakers and attendees that have been to this conference over the past 20 years is really amazing. Come down to Redmond or Orlando in 2018 and you can meet him in person.

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

Thomas Rayner - MVP

Thomas Rayner is a Microsoft Most Valuable Professional (MVP) and Honorary Scripting Guy with many years of experience in IT. He is a master technologist, specializing in DevOps, systems and process automation, public, private and hybrid cloud, and PowerShell. Thomas is an international speaker, best-selling author, and instructor covering a vast array of IT topics.

Thomas works for PCL Constructors on their DevOps and Automation team. He enjoys working with a wide variety of different products and technologies, particularly emerging and disruptive technologies, and automation-related products. His position with PCL affords him the luxury of facing interesting challenges every day.

BLOG: http://workingsysadmin.com

Twitter: @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

"Mick Pletcher is a nationally respected technology expert specializing in System Center Configuration Manager, Microsoft Deployment Toolkit, Active Directory, PowerShell Scripting, Visual Basic Scripting and Automation. In 2017, Mick was honored as a Microsoft Most Valuable Professional for his work in Cloud and Data Center. Also in 2017, he was one of seven IT professionals worldwide to be recognized as a technical star with the SAPIEN MVP Award. Mick is an avid blogger on information technology tips and topics at http://mickitblog.blogspot.com/. His blog, which topped a million hits in less than four years, was highlighted in Adaptiva's 2016 round-up of Top 16 SCCM Tips for 2016.

A SCCM Administrator with Waller Lansden Dortch & Davis, LLP, a Nashville-based law firm with more than 230 attorneys in four offices, Mick deploys software to more than 500 users across the Southeast and is responsible for automating tasks via the use of PowerShell, administering group policies, deploying Windows updates, and the PC build process. Prior to joining Waller, Mick implemented alternate system design approaches and managed software and Operating Systems using SCCM 2012, along with SMS Installer, PowerShell, and VBScript at one of the nation's largest architecture and engineering design firms. In 2013, Mick co-founded the Nashville PowerShell User Group. He is a relentless world traveler who has climbed Mount Kilimanjaro. Other hobbies include astronomy, welding and fabrication, river boarding, sport bikes, cycling and mountaineering."

Blog: http://mickitblog.blogspot.ca

Twitter: @mick_pletcher

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.

Cristal can be found speaking at Microsoft Ignite, MVPDays, and other local user groups. She is extremely active in the community and has recently helped publish a book for other Women MVP's called Voices from the Data Platform.

BLOG: http://www.checkyourlogs.net

Twitter: @supercristal1

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

Cary Sun – CCIE #4531 (Future Microsoft MVP)

Cary Sun is CISCO CERTIFIED INTERNETWORK EXPERT (CCIE No.4531) and MCSE, MCIPT, Citrix CCA with over twenty years in the planning, design, and implementation of network technologies and Management and system integration. Background includes hands-on experience with multi-platform, all LAN/WAN topologies, network administration, E-mail and Internet systems, security products, PCs and Servers environment. Expertise analyzing user's needs and coordinating system designs from concept through implementation. Exceptional analysis, organization, communication, and interpersonal skills. Demonstrated ability to work independently or as an integral part of team to achieve objectives and goals. Specialties: CCIE /CCNA / MCSE / MCITP / MCTS / MCSA / Solution Expert / CCA

Cary's is a very active blogger at checkyourlogs.net and always available online for questions from the community. He passion about technology is contagious and he makes everyone around him better at what they do.

Blog:http://www.checkyourlogs.net

Twitter:@SifuSun

Contents

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.mvpdays.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.

Sample Files

All sample files for this book can be downloaded from www.checkyourlogs.net and www.github.com/mvpdays

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

Using PowerShell to Download Drivers via FTP

By: Dave Kawula MVP

Hey fellow IT Pro's in today's blog post we will look at a super quick and dirty way to download files from your favorite FTP Site.

Luckily there is already an FTP Module up in the PowerShell Gallery that we will use for this called PSFTP.

I currently use this little trick to download the current supported drivers for our Storage Spaces Direct builds on SuperMicro hardware.

```
install-module PSFTP -Force
Import-Module -Name PSFTP
$username = "anonymous"
$password = "anonymous"
$secstr = New-Object -TypeName System.Security.SecureString
$password.ToCharArray() | ForEach-Object {$secstr.AppendChar($_)}
$cred = new-object -typename System.Management.Automation.PSCredential -
argumentlist $username, $secstr
```

```
Set-FTPConnection -Credentials $Cred  -Server ftp://ftp.supermicro.com -Session
DownloadingDrivers -UsePassive
$Session = Get-FTPConnection -Session DownloadingDrivers
```

```
Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Management/5.0.0.2192/IATA_CD.exe -
LocalPath "c:\post-install\SuperMicroDrivers" -RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path /driver/VGA/ASPEED/v1.03.zip -LocalPath
"c:\post-install\SuperMicroDrivers" -RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Windows/5.0.0.2192/Win.zip -
RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path /driver/LAN/Intel/PRO_v22.4.zip -
RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Management/5.0.0.2192/rste_5.0.0.2192_cl
i.zip -LocalPath "c:\post-install\SuperMicroDrivers" -RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Management/5.0.0.2192/rste_5.0.0.2192_in
stall.zip -LocalPath "c:\post-install\SuperMicroDrivers" -RecreateFolders -
Overwrite

Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Windows/5.0.0.2192/rste_5.0.0.2192_f6-
drivers.zip -LocalPath
```

```
PS C:\temp> install-module PSFTP -Force

Import-Module -Name PSFTP

$username = "anonymous"

$password = "anonymous"

$secstr = New-Object -TypeName System.Security.SecureString

$password.ToCharArray() | ForEach-Object {$secstr.AppendChar($_)}

$cred = new-object -typename System.Management.Automation.PSCredential -
argumentlist $username, $secstr




Set-FTPConnection -Credentials $Cred  -Server ftp://ftp.supermicro.com -Session
DownloadingDrivers -UsePassive

$Session = Get-FTPConnection -Session DownloadingDrivers



Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Management/5.0.0.2192/IATA_CD.exe -
LocalPath "c:\post-install\SuperMicroDrivers" -RecreateFolders -Overwrite
```

```
Get-FTPItem -Session $Session -Path /driver/VGA/ASPEED/v1.03.zip -LocalPath
"c:\post-install\SuperMicroDrivers" -RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Windows/5.0.0.2192/Win.zip -
RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path /driver/LAN/Intel/PRO_v22.4.zip -
RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Management/5.0.0.2192/rste_5.0.0.2192_cl
i.zip -LocalPath "c:\post-install\SuperMicroDrivers" -RecreateFolders -Overwrite

Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Management/5.0.0.2192/rste_5.0.0.2192_in
stall.zip -LocalPath "c:\post-install\SuperMicroDrivers" -RecreateFolders -
Overwrite

Get-FTPItem -Session $Session -Path
/driver/SATA/Intel_PCH_RAID_Romley_RSTE/Windows/5.0.0.2192/rste_5.0.0.2192_f6-
drivers.zip -LocalPath




ContentLength           : -1

Headers                 : {}

SupportsHeaders         : True

ResponseUri             : ftp://ftp.supermicro.com/

StatusCode              : ClosingData

StatusDescription       : 226 Successfully transferred "/"


LastModified            : 1/1/0001 12:00:00 AM

BannerMessage           : 220 Welcome To Supermicro FTP Site


WelcomeMessage          : 230 Logged on


ExitMessage             : 221 Goodbye


IsFromCache             : False

IsMutuallyAuthenticated : False
```

```
ContentType             :
```

I hope you enjoy this and the rest of the tricks throughout this book.

Dave

Chapter 2

Using PowerShell to Download Videos from Channel 9

By: Dave Kawula MVP

Today I want to feature a really cool little PowerShell Script to download your favorite content from Microsoft Channel 9 @CH9. As I do most days at lunch I scour the internet for great IT News, Blog Posts, and cool tricks to help me with my day job. Today I was browsing my friend Vlad Catrinescu's @vladcatrinescu blog: https://absolute-sharepoint.com/ and I found this amazing post…

https://absolute-sharepoint.com/2017/05/the-ultimate-script-to-download-microsoft-build-2017-videos-and-slides.html

Basically, it can be used as a downloader for any Channel 9 content from Microsoft. Sometimes it is nice to have offline content for when you are on the plane and this one really does the trick.

Now the code you see below is slightly modified as I thought it would be cool to download all the @MVPDays 2017 content.

```
#Script written by Vlad Catrinescu

#Visit my site www.absolute-sharepoint.com

#Twitter: @vladcatrinescu

#Originally Posted here: https://wp.me/p3utgI-865

#Slight Modifications to work with MVPDays Community Roadshow Content on Channel
9

#by Dave Kawula - MVP

#@DaveKawula

#Nice work VLAD -- This might make Master PowerShell Tricks V3 :)



Param(

  [string]$keyword,[string]$session

)


######    Variables  #####

#Location - Preferably enter something not too long to not have filename
problems! cut and paste them afterwards

$downloadlocation = "G:\MVPDays2017"

#Ignite 2016 Videos RSS Feed

[Environment]::CurrentDirectory=(Get-Location -PSProvider
FileSystem).ProviderPath

$rss = (new-object net.webclient)

$video1 =
([xml]$rss.downloadstring("http://s.ch9.ms/events/MVPDays/MVPDays2017RoadShow/rs
s/mp4high"))

$video2 =
([xml]$rss.downloadstring("http://s.ch9.ms/events/MVPDays/MVPDays2017RoadShow/rs
s/mp4high?page=2"))

#other qualities for the videos only. Uncomment below and delete the two
previous lines to download Mid Quality videos

#$video1 =
([xml]$rss.downloadstring("http://s.ch9.ms/events/build/2017/rss/mp4"))

#$video2 =
([xml]$rss.downloadstring("http://s.ch9.ms/events/build/2017/rss/mp4?page=2"))
```

```
$slide1 =
([xml]$rss.downloadstring("http://s.ch9.ms/events/MVPDays/MVPDays2017RoadShow/rs
s/slides"))

$slide2 =
([xml]$rss.downloadstring("http://s.ch9.ms/events/MVPDays/MVPDays2017RoadShow/rs
s/slides?page=2"))


#SCRIPT/ Functions  Do not touch below this line :)#

if (-not (Test-Path $downloadlocation)) {

  Write-Host "Folder $fpath dosen't exist. Creating it..."

  New-Item $downloadlocation -type directory | Out-Null

}

set-location $downloadlocation

function CleanFilename($filename)

{

    return $filename.Replace(":", "-").Replace("?", "").Replace("/", "-
").Replace("<", "").Replace("|", "").Replace('"',"").Replace("*","")

}


function DownloadSlides($filter,$videourl)

{

    try

    {

        $videourl.rss.channel.item | Where{($_.title -like “*$filter*”) -or
($_.link -like "*/$filter")} |

        foreach {

         $code = $_.comments.split("/") | select -last 1


         # Grab the URL for the PPTX file

         $urlpptx = New-Object System.Uri($_.enclosure.url)

            $filepptx = $code + "-" + $_.creator + "-" +
(CleanFileName($_.title))

         $filepptx = $filepptx.substring(0, [System.Math]::Min(120,
$filepptx.Length))
```

```
        $filepptx = $filepptx.trim()
        $filepptx = $filepptx + ".pptx"
        if ($code -ne "")
        {
          $folder = $code + " - " + (CleanFileName($_.title
  $folder = $folder.substring(0, [System.Math]::Min(100, $folder.Length))
          $folder = $folder.trim()
        }
        else
        {
         $folder = "NoCodeSessions"
        }

        if (-not (Test-Path $folder)) {
         Write-Host "Folder $folder dosen't exist. Creating it..."
         New-Item $folder -type directory | Out-Null
        }
        # Make sure the PowerPoint file doesn't already exist
        if (!(test-path "$downloadlocation\$folder\$filepptx"))
        {
         # Echo out the  file that's being downloaded
         write-host "Downloading slides: $filepptx"
         #$wc = (New-Object System.Net.WebClient)
         # Download the MP4 file
         #$wc.DownloadFile($urlpptx, "$downloadlocation\$filepptx")
               Start-BitsTransfer $urlpptx "$downloadlocation\$filepptx" -
DisplayName $filepptx
         mv $filepptx $folder
        }
           else
           {
```

```
            write-host "Slides exist: $filepptx"
           }
     }
}

    catch
    {
        $ErrorMessage = $_.Exception.Message
        Write-host "$ErrorMessage"
    }
}

function DownloadVideos($filter,$slideurl)
{
#download all the mp4
# Walk through each item in the feed
$slideurl.rss.channel.item | Where{($_.title -like “*$filter*”) -or ($_.link -like "*/$filter*")} | foreach{
$code = $_.comments.split("/") | select -last 1

# Grab the URL for the MP4 file
$url = New-Object System.Uri($_.enclosure.url)

# Create the local file name for the MP4 download
$file = $code + "-" + $_.creator + "-" + (CleanFileName($_.title))
$file = $file.substring(0, [System.Math]::Min(120, $file.Length))
$file = $file.trim()
$file = $file + ".mp4"

if ($code -ne "")
{
```

```
    $folder = $code + " - " + (CleanFileName($_.title))
    $folder = $folder.substring(0, [System.Math]::Min(100, $folder.Length))
    $folder = $folder.trim()
}
else
{
  $folder = "NoCodeSessions"
}

if (-not (Test-Path $folder)) {
  Write-Host "Folder $folder) dosen't exist. Creating it..."
  New-Item $folder -type directory | Out-Null
}



# Make sure the MP4 file doesn't already exist
 if (!(test-path "$folder\$file"))
{
  # Echo out the  file that's being downloaded
  write-host "Downloading video: $file"
  #$wc = (New-Object System.Net.WebClient)
  # Download the MP4 file
  Start-BitsTransfer $url "$downloadlocation\$file" -DisplayName $file
  mv $file $folder
}
    else
    {
     write-host "Video exists: $file"
    }
```

```
#text description from session
$OutFile = New-Item -type file
"$($downloadlocation)\$($Folder)\$($Code.trim()).txt" -Force
    $Category = "" ; $Content = ""
    $_.category | foreach {$Category += $_ + ","}
    $Content = $_.title.trim() + "`r`n" + $_.creator + "`r`n" +
$_.summary.trim() + "`r`n" + "`r`n" + $Category.Substring(0,$Category.Length -1)
   add-content $OutFile $Content


}
}


 if ($keyword)
{
    $keywords = $keyword.split(",")

    foreach ($k in $keywords)
    {
        $k.trim()
        Write-Host "You are now downloading the sessions with the keyword $k"
        DownloadSlides $k $slide1
        DownloadSlides $k $slide2
        DownloadVideos $k $video1
        DownloadVideos $k $video2
    }
}
elseif ($session)
{
    $sessions = $session.Split(",")

    foreach ($s in $sessions)
```

```
    {
        $s.trim()
        Write-Host "You are now downloading the session $s"
        DownloadSlides $s $slide1
        DownloadSlides $s $slide2
        DownloadVideos $s $video1
        DownloadVideos $s $video2
    }
}
else
{
    DownloadSlides " " $slide1
    DownloadSlides " " $slide2
    DownloadVideos " " $video1
    DownloadVideos " " $video2
}
```

Hope you enjoy and happy learning,

Dave

Chapter 3

Snapshot Management of VMware with PowerShell

By: Allan Rafuse MVP

This is one of those management tasks that comes up at any location you're at, especially when you're trying to manage VMs, performance or datastore space. The cleanup task of deleting snapshots is easy, but the questions that always comes to mind are: Who created it, when and why. Take a look at a quick script I wrote to answer that information. I schedule it to run every Monday morning and email the results. Simple! This framework works for both VMware and Hyper-V.

Snapshot Management

As I mentioned above, it's easy to delete the snapshot, but why was it created. It would be great if everyone put in meaningful names, a descriptive description, and also told us when we could delete the snapshot. The longer we leave snapshots, the more we are going to degrade performance, not only to the VM itself, but as the snapshot grows, it will take extra cycles away from the hosts to serve up the required data.

Snapshots should really only be used as a Cover Your A** (CYA). Essentially:

1. Take a snapshot
2. Make a change to the VM (Upgrade, Patch)
3. Test the change
4. Make a choice
    a. Changes are good, delete the snapshot
    b. Changes failed, revert and delete the snapshot

Reality – I'll just keep the snapshot for a few days in case someone finds an issue.

Password Issues

Ever keep that snapshot for over 30 days? Well if you revert a Windows machine to snapshot that is older than 30 days, you're most likely going to have machine password authentication problems with the domain. By default, machine account password changes are initiated by the computer every 30 days. So that means when you go to log into the machine, the trust between the machine and AD is broken.

To get around this issue, you can try the following options:

1. Log on with a local account
2. Disconnect the network adapter, log on with a domain account with cached credentials

Now if there are legitimate reasons for retaining snapshots for a length of time (Packaging machines, Gold Images etc), you may want to look at the following security option (via local security editor or group policy)

Setting: Domain member: Maximum machine account password age

Location: Computer Configuration\Windows Settings\Security Settings\Local Policies\Security Options

Report Output

Here is a sample of what the PowerShell script will create.

Report Date: 06/08/2017 15:33:03

VM	Name	User	Created	Description
Packaging1	Ready for Packaging	CORP\admin2	4/29/2017 8:08	Used for SCCM Packaging
Packaging2	Ready for Packaging	CORP\admin2	4/29/2017 8:44	Used for SCCM Packaging
SQL01	before upgrade	CORP\vmadmin1	4/29/2017 17:32	before upgrade
AD03	before upgrade	CORP\vmadmin1	4/29/2017 17:33	before upgrade
SQL99	before upgrade	CORP\vmadmin3	4/29/2017 17:39	before upgrade
WEB03	Before upgrade to UR9	CORP\admin2	4/2/2017 15:57	Before upgrade to UR9
WEB04	Before upgrade to UR9	CORP\admin2	5/2/2017 15:59	Before upgrade to UR9
WEB05	Before upgrade to UR9	CORP\admin17	5/2/2017 16:04	Before upgrade to UR9
SPF01	Before upgrade to UR9	CORP\admin2	5/2/2017 16:11	Before upgrade to UR9
SPF02	Before upgrade to UR9	CORP\admin17	5/2/2017 16:11	Before upgrade to UR9
APP15	Application Upgrade	CORP\vmadmin3	5/7/2017 14:29	Delete after June 21/17

Number of Snapshots: 11

Generated on Mgmt01

The Code (VMWare Example)

The code itself for reporting is pretty simple and short. The script I have is larger cause I try and write all my useful scripts with script parameters, and in this case the code is a little larger as I kick it out to email.

Parameters

The parameters are pretty straight forward. Which virtual center machine are we going to connect to, how to send an email, and most importantly, only email machines that are older than X days (14 by default).

```
param (
    $VirtualCenter = "VirtualCenter.corp.local",
    $smtpServer = "smtp1.corp.local",
    $smtpFrom = "vmware@corp.local",
    $smtpTo = "arafuse@corp.local",
```

```
    $smtpSubject = "VMware Snapshots",
    $SnapShotsOlderThanXDays = 14
)
```

Connect to Virtual Center

Next step is to connect to Virtual Center

```
Get-Module -ListAvailable VMware.VimAutomation.* | Import-Module -ErrorAction
SilentlyContinue
If ($global:DefaultVIServer) {
    Disconnect-VIServer * -Confirm:$false -ErrorAction SilentlyContinue
}
$VCServer = Connect-VIServer -Server $VirtualCenter
```

Get and Create the Snapshot Report

Here is the worker code of this script. Some VMs are allowed to have snapshots, so we define a list of regular expressions to filter out. The next one is some of the secret sauce to figuring out who created the snapshot. To do this we need to go back through the VM events and look for the Create Snapshot even. From here we can determine who created the snapshot. To help limit the speed, we know what time the snapshot was created, so this code is going to look at the past 4000 events for that VM starting 10 seconds before the snapshot was created.

As I sometimes run this code interactively, I first create a report with all the snapshots regardless of the date created (excluding the allowed VM with snapshots). This allows me to see everything. But during script execution, I then filter out anything older than 14 days. Those are the culprits I want to delete!

```
$VmsWithAllowedSnaps = @(".*SnappyImage.*")
$LogEntriesPerVM = 4000
```

```
$VMs = Get-VM
Foreach ($VmsWithAllowedSnap in $VmsWithAllowedSnaps) {
    $VMs = $VMs | Where {$_.Name -notmatch $VmsWithAllowedSnap}
}
$SnapShots = $VMs | Get-Snapshot

$date = Get-Date
$measure = Measure-Command {
    $report = $Snapshots | Select-Object VM, Name, @{Name="User"; Expression = {
(Get-VIEvent -Entity $_.VM -MaxSamples $LogEntriesPerVM -Start
$_.Created.AddSeconds(-10) | Where {$_.Info.DescriptionId -eq
"VirtualMachine.createSnapshot"} | Sort-Object CreatedTime | Select-Object -
First 1).UserName}}, Created, @{Name="Days Old"; E={$_.Created - }}, Description
| Sort-Object -Property "Created"
}
#($measure).TotalMinutes

$report = $report | Where {($_.Created).AddDays([int]$SnapShotsOlderThanXDays) -
lt (Get-Date)}
```

Email the Results

Scripts are great! Scripts that email you the results are even greater! You can use this generic fragment of code almost anywhere. It takes your $report object, put it into an HTML table and emails it! If you don't like the colors, there are many things you can do in the $head block below by adding/modifying CSS styles.

```
$head = @"
<title>Snapshot Daily/Weekly Report</title>
<style type="text/css">
```

```
  body { background-color: white; }
  table { border-width: 1px; border-style: solid; border-color: black; border-collapse: collapse; }
  th {border-width: 1px; padding: 0px; border-style: solid; border-color: black; background-color:thistle }
  td {border-width: 1px ;padding: 0px; border-style: solid; border-color: black; }
  tr:nth-child(odd) { background-color:#d3d3d3; }
  tr:nth-child(even) { background-color:white; }
</style>
"@

$postContent = @"
<p>Number of Snapshots: $($report.count)</p>
<p>Generated on $($ENV:COMPUTERNAME)</p>
"@

#Send Email Report
$date = Get-Date
$message = New-Object System.Net.Mail.MailMessage $smtpFrom, $smtpTo
$message.Subject = $smtpSubject
$message.IsBodyHTML = $true

$SnapshotReportHTML = $report | ConvertTo-Html -Head $head -PreContent "Report Date: $date" -PostContent $PostContent
$message.Body = $SnapshotReportHTML | Out-String
$smtp = New-Object Net.Mail.SmtpClient($smtpServer)
$smtp.Send($message)
```

Schedule It

You'll see me type this over and over. I schedule this script to run on Monday mornings. This way when I come into the office, there is a report sitting in people's mailboxes. It's clean up time!

Happy Snapshot Management!

Allan

Chapter 4

Setting SQL Server Memory Allocation (Maximum and Minimum)

By: Allan Rafuse MVP

If you've ever run an installation of SQL Server, you'll know it's a database, and databases love, love, love memory. By design and by default, Microsoft SQL Server thinks it's the only process on the system and is therefore given all the available memory and CPUs. As a best practice, I limit this. Here is the script I use to edit these values.

The first thing to know is how much memory/RAM the server has been allocated or has installed. SQL Server will be happy to use it all as we all know, but sadly, SQL Server isn't the only process running. Depending on the organization, the environment, there are other processes running alongside of SQL Server. Think about AV and Backup software. Oh, did you forget about one of the most important areas of the system that needs memory available? This is the Operating System itself! Your performance will surely start to tank if your OS runs out of memory and starts to swap.

Another process that could be running on the SQL Server machine, is another instance of SQL Server! There are many reasons and implementations where multiple instances run on the same machine. It's not nice when one instance uses all the memory and the other instances don't get what they need!

Retrieving the Physical Memory

To keep the code clean, I've put this in a function. The function will return the amount of physical memory in Megabytes.

```
Function Get-ComputerMemory {
    $mem = Get-WMIObject -class Win32_PhysicalMemory |
            Measure-Object -Property Capacity -Sum
    return ($mem.Sum / 1MB);
}
```

Determine SQL Server Maximum Memory

Now that we know how much memory is in the system, it's time to make some choices on how much SQL Server will be allowed to use. These numbers have worked for me and can be found in most of my SQL Server implementations.

My calculations for how much memory to allow SQL Server to use are:

1. If the computer has less than 8GB of physical memory, allocate 80% of it to SQL Server and leave 20% for the OS and other applications
2. If the computer has more than 8GB of physical memory, reserve 2GB for the OS and other applications. SQL Server will get the remaining amount

This are my numbers that I use. And just because I'm sharing my #PowerShell code, doesn't mean that you have to use every piece of code, character by character!

```
Function Get-SQLMaxMemory {
    $memtotal = Get-ComputerMemory
    $min_os_mem = 2048 ;
    if ($memtotal -le $min_os_mem) {
        Return $null;
    }
```

```
    if ($memtotal -ge 8192) {
        $sql_mem = $memtotal - 2048
    } else {
        $sql_mem = $memtotal * 0.8 ;
    }
    return [int]$sql_mem ;
}
```

Reconfigure SQL Server Memory Allocation

This code is pretty straight forward, but SQL Server doesn't have too many PowerShell cmdlets for us. To reconfigure the memory allocations, we have to use SQL Server Management Objects (SMO). To access SMO and pull it into our PowerShell world, we access them via .the NET Framework class. Once the class is loaded into our environment, we can then create native PowerShell objects. Pretty cool I'd say!

```
Function Set-SQLInstanceMemory {
    param (
        [string]$SQLInstanceName = ".",
        [int]$maxMem = $null,
        [int]$minMem = 0
    )

    if ($minMem -eq 0) {
        $minMem = $maxMem
    }
    [reflection.assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo") | Out-
Null
```

```
    $srv = New-Object
Microsoft.SQLServer.Management.Smo.Server($SQLInstanceName)
    if ($srv.status) {
        Write-Host "[Running] Setting Maximum Memory to:
$($srv.Configuration.MaxServerMemory.RunValue)"
        Write-Host "[Running] Setting Minimum Memory to:
$($srv.Configuration.MinServerMemory.RunValue)"


        Write-Host "[New] Setting Maximum Memory to: $maxmem"
        Write-Host "[New] Setting Minimum Memory to: $minmem"
        $srv.Configuration.MaxServerMemory.ConfigValue = $maxMem
        $srv.Configuration.MinServerMemory.ConfigValue = $minMem
        $srv.Configuration.Alter()
    }
}
```

Note: These changes take place immediately. See https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/server-memory-server-configuration-options

The min server memory and max server memory options are advanced options. If you are using the sp_configure system stored procedure to change these settings, you can change them only when show advanced options is set to 1. These settings take effect immediately without a server restart.

The previous link also has a lot of great information about memory allocation.

Making IT Work

Now that we've defined a whole 2 functions, we need to call them. I actually put them into one line. Looks better and cleaner in my opinion

```
$MSSQLInstance = "sql01\SQLInstance01"
Set-SQLInstanceMemory $MSSQLInstance (Get-SQLMaxMemory)
```

Hope you enjoyed and happy scripting.

Allan

Chapter 5

Using PowerShell to Add a Direct Member to an SCCM Collection

By: Allan Rafuse MVP

I was working at a client site and was going through their server rollout procedure. I was quite shocked as to how many manual tasks they still had. One of these tasks was to add a computer directly to a SCCM collection. According to their requirements, they had to use direct membership and could not do a WMI call. So, I created the following script and added it to their task sequence.

Problem –

The task sequence runs on the client machine and we really don't want to install the SCCM PowerShell cmdlets on every server. Instead, what we'll do is we'll run the PowerShell remotely. The computer that is running the task sequence will open a remote connect and run them against the SCCM server. The SCCM server has the ConfigureManager PowerShell module, it can do the work for us!

Things to think about –

- **The computer running the task sequence needs to be able to use PowerShell remoting**
- **Firewall's are opened**
- **SCCM Server has had Windows Remote Shell enabled**
- **The account that runs it must have access to update the collection**

```
$SccmServer = "SCCM01"

$PathToSCCMModule = "D:\Program Files\Microsoft Configuration
Manager\AdminConsole\bin\ConfigurationManager.psd1"


$MemberName = $env:COMPUTERNAME

$SCCMSession = New-PSSession -ComputerName $SccmServer

Invoke-Command -Session $SccmSession -ArgumentList @($PathToSCCMModule,
$MemberName) -ScriptBlock {

    Param (

        [string]$PathToSCCMModule,

        [string]$MemberName

    )

    Import-Module $PathToSCCMModule -ErrorAction SilentlyContinue

    $SccmSite = (Get-PSDrive -PSProvider CMSite | Sort-Object -Property Name |
Select-Object -First 1).Name

    Set-Location "$($SccmSite):"


    $ResourceID = (Get-CMDevice -Name $MemberName).ResourceID

    If ($ResourceID) {

        Add-CMDeviceCollectionDirectMembershipRule -CollectionName "SCEP -
Servers" -ResourceId $ResourceID

    }

}
```

Until next time happy scripting.

Allan

Chapter 6

Using PowerShell to Manage the Datadog Cloud Service

By: Allan Rafuse MVP

PowerShell to the rescue again! Datadog is a Cloud service for aggregating real-time metrics, events and logs from all your servers. The easiest way is to install an agent and let it report via HTTPS directly to the internet or via a web proxy. Another cloud aggregation solution that I'm more familiar with is Microsoft Operations Management Suite (OMS). Both of these services provide access via PowerShell.

Anyways, back to the actual blog post as you've probably come across this searching for Datadog and PowerShell! Datadog doesn't provide a PowerShell module directly, but it does expose a lot of functionality via web services. There are a few authentication prerequisites that you need to do inside the Datadog portal though before you go ahead and attempt to communicate with the API.

1. Create an API-Key
2. Create an Application Key

Everyone connects to Datadog using their public URL, but instead of using a Username and Password combination, they've termed them API-Key and Application Key. Using these two together gives you access to your Datadog subscription and information.

Everyone connects to Datadog using their public URL, but instead of using a Username and Password combination, they've termed them API-Key and Application Key. Using these two together gives you access to your Datadog subscription and information.

Datadog publishes API documentation at http://docs.datadoghq.com/api/ . It has examples in Shell, Python and Ruby. Click on the area you want to see the API for and then click on the

desired language. As the Shell method is the closest to HTTPS web service calls, I suggest you use that in order to understand the Datadog API and web service call.

Windows PowerShell comes to the rescue again. Not only can we do a web service call using Invoke-WebRequest , we can also deal with the Datadog response. This response will be in a JSON format (Essentially a less complex/verbose form of XML). We'll use PowerShell's ConvertFrom-Json cmdlet to create our handy PowerShell object.

Authentication

At the top of all my Datadog scripts I have the API and authentication information:

```
# http://docs.datadoghq.com/api/#embeds
$url_base = "https://app.datadoghq.com/"
$api_key = "asdlfk771ja8z8m0980asz8knnn5f9a9"
$app_key = "x5jaja81jamnz81o85618fcce8a891912387a7f3"
```

Example Snippets

Below are a few snippets to get you going with Datadog. Most of the changes in each of the snippets are in the $url_signature line. This tells Datadog what information your actually after. Watch out as not all the API calls use api/v1, some may be api/v2.

After you prepare your URL line and parameters, you send it via Invoke-WebRequest and tell PowerShell to set the content type as JSON. Parse your way through $response.Content and find the relevant information you want.

Pulling Authenticated Users

```
#Users
$url_signature = "api/v1/user"
```

```
$url = $url_base + $url_signature + "?api_key=$api_key" + "&" +
"application_key=$app_key"

$response = Invoke-WebRequest -ContentType "application/json" -Uri $url

$response.Content | ConvertFrom-Json | Select-Object -ExpandProperty Users
```

Muting a Host

```
# Mute

$url_signature = "api/v1/host/MyHostName1/mute"

$url = $url_base + $url_signature + "?api_key=$api_key" + "&" +
"application_key=$app_key"

$response = Invoke-WebRequest -Uri $url -Method Post

$response.Content | ConvertFrom-Json
```

Unmuting a Host

```
# Unmute

$url_signature = "api/v1/host/WMAPMTSTEST/unmute"

$url = $url_base + $url_signature + "?api_key=$api_key" + "&" +
"application_key=$app_key"

$response = Invoke-WebRequest -Uri $url -Method Post

$response.Content | ConvertFrom-Json
```

Display Host/Agent Details

```
$includeInfo = @(
```

```
    "with_apps=true",
    "with_sources=true",
    "with_aliases=true",
    "with_meta=true",
    "with_mute_status=true",
    "with_tags=true"
)

$metricInfo = @(
    "metrics=avg",
    "system.cpu.idle avg",
    "aws.ec2.cpuutilization avg",
    "vsphere.cpu.usage avg",
    "azure.vm.processor_total_pct_user_time avg",
    "system.cpu.iowait avg",
    "system.load.norm.15"
)

$url_query = ""
$url_signature = "reports/v2/overview"
$url = $url_base + $url_signature + "?api_key=$api_key" + "&amp;" +
"application_key=$app_key" + "&amp;" + "window=3h" + "&amp;" + (($metricInfo -
join "%3A") -replace " ", "%2C") + "&amp;" + ($includeInfo -join "&amp;")
if ($url_query) {
    $url += "&amp;" + $url_query
}
$response = Invoke-WebRequest -Uri $url -Method Get
$response.Content | ConvertFrom-Json | Select-Object -ExpandProperty rows |
Select-Object Host_name,  @{n="Actively_Reporting"; e={$_.has_metrics}},
@{n="Agent_Version"; e={$_.meta.Agent_version}}, @{n="Agent_Branch";
e={($_.meta.gohai | ConvertFrom-Json).gohai | Select-Object -ExpandProperty
git_branch}}, @{n="ip"; e={($_.meta.gohai | ConvertFrom-Json).network | Select-
Object -ExpandProperty ipaddress}}, @{n="LogicalProcessors"; e={$logical =
($_.meta.gohai | ConvertFrom-Json).cpu | Select-Object -ExpandProperty
```

```
cpu_logical_processors; $cpu_cores = ($_.meta.gohai | ConvertFrom-Json).cpu |
Select-Object -ExpandProperty cpu_cores; ($logical / $cpu_cores) * $logical }} |
Sort-Object -Property host_name | ft
```

Searching for Events

In this example we'll query any Microsoft event log errors between a certain time range and have them passed back. Then we'll convert them from JSON and look for specific event log error messages.

```
# Event Log Errors
$dateStart = (Get-Date (Get-Date).AddDays(-30) -Uformat %s) -replace "\..*", ""
$dateEnd = (Get-Date (Get-Date).AddDays(0) -Uformat %s) -replace "\..*", ""
$url_signature = "api/v1/events"

$EventSearch = @(
    "start=$dateStart",
    "end=$dateEnd"
    "source=Event Viewer"
)

$url = $url_base + $url_signature + "?api_key=$api_key" + "&amp;" +
"application_key=$app_key" + "&amp;" + ($EventSearch -join "&amp;")
$response = Invoke-WebRequest -Uri $url -Method Get

$response.Content | ConvertFrom-Json | Select-Object -ExpandProperty events |
Where {$_.Title -eq "Application/Microsoft-Windows-Folder Redirection" -and
$_.Text -like "*redirect folder*"} | Select-Object -Unique -Property Text| fl
text
$response.Content | ConvertFrom-Json | Select-Object -ExpandProperty events |
Where {$_.Title -eq "System
```

Until next time happy scripting.

Allan

Chapter 7

Using PowerShell to Update the .Default and All User Profiles Registry

By: Allan Rafuse MVP

There are times that you may need to push out a change to all existing user profiles and to new profiles that are created on a system. I've seen a few PowerShell scripts floating around out there, but they didn't seem to work for Windows 7 SP1. You may or may not be surprised, but there are many organizations that still run Windows 7. The script is actually pretty simple.

Here is the breakdown of the script:

- Enumerate all the existing user profiles
- Add the .DEFAULT user profile to the list of existing user profiles
- Iterate through all the profiles
    - If the profile hive is not loaded, load it
    - Manipulate the users' registry
    - If the profile hive was loaded by the script, unload it
- Finished

Enumerate all the existing user profiles

Using the registry path below, we can find a list of all the user profiles on the system and where the profile path exists. Every user profile has the file NTuser.dat which contains the registry hive that is loaded into the HKEY_USERS and HKCU when a user logs on to the system. This NTuser.dat can for example also be loaded when using RunAs.exe. It will then only show up in HKEY_USERS\<users' SID>

```
# Get each user profile SID and Path to the profile
$UserProfiles = Get-ItemProperty "HKLM:\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\ProfileList\*" | where {$_.PSChildName -match "S-1-5-21-(\d+-
?){4}$" } | Select-Object @{Name="SID"; Expression={$_.PSChildName}},
@{Name="UserHive";Expression={"$($_.ProfileImagePath)\NTuser.dat"}}
```

Add the .DEFAULT user profile to the list of existing profiles

If you need to manipulate the registry of all new profiles, then you'll need to add the following code. The .DEFAULT user information does not exist in the registry key information above.

```
# Add in the .DEFAULT User Profile
$DefaultProfile = "" | Select-Object SID, UserHive
$DefaultProfile.SID = ".DEFAULT"
$DefaultProfile.Userhive = "C:\Users\Public\NTuser.dat"
$UserProfiles += $DefaultProfile
```

Iterate through all the profiles

This is the main code where we will determine if we need to load or unload any user registry hives. It is also where the registry changes will be made.

```
# Loop through each profile on the machine</p>

Foreach ($UserProfile in $UserProfiles) {

    # Load User ntuser.dat if it's not already loaded

    If (($ProfileWasLoaded = Test-Path Registry::HKEY_USERS\$($UserProfile.SID))
-eq $false) {

        Start-Process -FilePath "CMD.EXE" -ArgumentList "/C REG.EXE LOAD
HKU\$($UserProfile.SID) $($UserProfile.UserHive)" -Wait -WindowStyle Hidden

    }
```

Manipulate the users' registry

This is the area where you can create, delete or modify the registry. After the changes are made, the profile will be unloaded. Upon the next logon, the changes will come into effect.

```
# Manipulate the registry

$key =
"Registry::HKEY_USERS\$($UserProfile.SID)\Software\SomeArchaicSoftware\Configura
tion"

New-Item -Path $key -Force | Out-Null

New-ItemProperty -Path $key -Name "LoginURL" -Value
"https://www.myCompany.local" -PropertyType STRING -Force | Out-Null

New-ItemProperty -Path $key -Name "DisplayWelcome" -Value 0x00000001 -
PropertyType DWORD -Force | Out-Null


$key = "$key\UserInfo"

New-Item -Path $key -Force | Out-Null
```

```
New-ItemProperty -Path $key -Name "LoginName" -Value
"$($ENV:USERDOMAIN)\$($ENV:USERNAME)" -PropertyType STRING -Force | Out-Null
```

If the Profile hive was loaded by script, unload it

This is another area that is easier to just call out to REG.EXE again to unload the registry. One issue to keep in mind is that if any handles are open to the registry, they need to be closed. If they're not closed, you'll get "Access Denied" when trying to unload the registry hive. This is why I've added the Garbage Collector. This cleans up all open handles [gc]::Collector. I also noticed that if I was opening and closing registry hives too fast, they all weren't being closed. I'm guessing this is due to a race condition. I added a Start-Sleep 1 and this fixed the problem for me.

```
# Unload NTuser.dat
If ($ProfileWasLoaded -eq $false) {
    [gc]::Collect()
    Start-Sleep 1
    Start-Process -FilePath "CMD.EXE" -ArgumentList "/C REG.EXE UNLOAD
HKU\$($UserProfile.SID)" -Wait -WindowStyle Hidden| Out-Null
}
```

Happy coding and I hope this helps you solve whatever your problem was!

Allan

Chapter 8

Working with PowerShell Active Directory Module as a Non-Privileged User

By: Thomas Rayner - MVP

As a best practice, as an administrator you should have separate accounts for your normal activities (emails, IM, normal stuff) and your administrative activities (resetting passwords, creating new mailboxes, etc.). It's obviously best not to log into your normal workstation as your administrative user. You're also absolutely not supposed to remote desktop into a domain controller (or another server) just to launch a PowerShell console, import the ActiveDirectory module, and run your commands. Here's a better way.

We're going to leverage the `$PSDefaultParameterValues` built-in variable which allows you to specify default values for cmdlets every time you run them.

First, set up a variable to hold your credentials.

```
$acred = Get-Credential -Message 'Admin creds'
```

Now, import the Active Directory module.

```
Import-Module ActiveDirectory
```

And finally, a little something special.

```
$PSDefaultParameterValues += @{ 'activedirectory:\*:Credential' = $acred }
```

I'm adding a value to my `$PSDefaultParameterValues` variable. What I'm saying is for all the cmdlets in the ActiveDirectory module, set the -Credential parameter equal to the `$acred` variable that I set first.

Now when I run any commands using the ActiveDirectory module, they'll run the the administrative credentials I supplied, instead of the credentials I'm logged into the computer with.

Chapter 9

Using PowerShell to Split a String Without Losing the Character You Split On

By: Thomas Rayner – MVP

Previously, I've written about the difference between .split() and -split in PowerShell. We're going to keep splitting strings, but we're going to try to retain the character that we're splitting on. Whether you use .split() or -split, when you split a string, it takes that character and essentially turns it into the separation of the two items on either side of it. But, what if I want to keep that character instead of losing it to the split?

Well, we're going to have to dabble in regular expressions. Before you run away screaming, as I know some people do when it comes to regex, let me walk you through this and see if you don't mind dipping a toe in these waters.

In our scenario, I've got a filename and I'm going to split it based on the slashes in the path. Normally I'd get something like this.

```
$filename = get-item C:\temp\demo\thing.txt
$filename -split '\\'
```

```
C:
temp
demo
thing.txt
```

Notice how I had to split on "\"? I had to escape that backslash. We're regexing already! Also notice that I lost the backslash on which I split the string. Now let's do a tiny bit more regex in our split pattern to retain that backslash.

```
$filename -split '(?=\\)'
```

```
C:
\temp
\demo
\thing.txt
```

Look at that, we kept our backslash. How? Well look at the pattern we split on: (?=\). That's what regex calls a "lookahead". It's contained in round brackets and the "?=" part basically means "where the next character is a " and the "\" still means our backslash. So we're splitting the string on the place in the string where the next character is a backslash. we're effectively splitting on the space between characters.

NEAT! Now what if I wanted the backslash to be on the other side? That is, at the end of the string on each line instead of the start of the line after? No worries, regex has you covered there, too.

```
$filename -split '(?<=\\)'
```

```
C:\
temp\
demo\
thing.txt
```

This is a “lookbehind”. It’s the same as a lookahead, except it’s looking for a place where the character to the left matches the pattern, instead of the character to the right. A lookbehind is denoted with the “?<=” characters.

There are plenty of resources online about using lookaheads and lookbehinds in regex, but if you’re not looking specifically for regex resources, you probably wouldn’t have found them. If PowerShell string splitting is what you’re after, hopefully you found this interesting.

Regex isn’t that scary, right?

Chapter 10

What's the difference between -split and .split() in PowerShell?

By: Thomas Rayner – MVP

Here's a question I see over and over and over again: "I have a string and I'm trying to split it on this part, but it's jumbling it into a big mess. What's going on?" Well, there's splitting a string in PowerShell, and then there's splitting a string in PowerShell. Confused? Let me explain.

Say you have this string for our example.

```
$splitstring = 'this is an interesting string with the letters s and t all over
the place'

$splitstring.split('s')

thi
 i
 an intere
ting
tring with the letter

 and t all over the place
```

That did exactly what we thought it would. It took our string and broke it apart on all the “s”‘s. Now, what if I want to split it where there’s an “st”? There’s only two spots it should split: the “st” in “interesting” and in “string”. Let’s try the same thing we tried before.

```
$splitstring.split('st')
```

```
hi
 i
 an in
ere

ing

ring wi
h
he le

er

 and
 all over
he place
```

Well that ain’t right. What happened? If we look closely, we can see that our string was split anywhere that there was an “s” or a “t”, rather than where there was an “st” together.

.split() is a method that takes an array of characters and then splits the string anywhere it sees any of those characters.

-split is an operator that takes a pattern string and splits the string anywhere it sees that pattern.

Here's what I should have done to split our string anywhere there's an "st".

```
$splitstring -split 'st'
```

```
this is an intere
ing
ring with the letters s and t all over the place
```

That looks more like we're expecting.

Remember, .split() takes an array of characters, -split takes a string.

Chapter 11

PowerShell Rules for Format-Table and Format-List

By: Thomas Rayner – MVP

In PowerShell, when outputting data to the console, it's typically either organized into a table or a list. You can force output to take either of these forms using the Format-Table and the Format-List cmdlets, and people who write PowerShell cmdlets and modules can take special steps to make sure their output is formatted as they desire. But, when no developer has specifically asked for a formatted output, how does PowerShell choose to display a table or a list?

The answer is actually pretty simple and I'm going to highlight it with an example. Take a look at the following piece of code.

```
get-wmiobject -class win32_operatingsystem | select
pscomputername,caption,osarch*,registereduser
```

```
PS C:\Users\DKTCLAPTOP> get-wmiobject -class win32_operatingsystem | select
pscomputername,caption,osarch*,registereduser


PSComputerName caption                        OSArchitecture registereduser
-------------- -------                        -------------- --------------
DKLAPTOP99     Microsoft Windows 10 Enterprise 64-bit         DKTCLAPTOP
```

I used `Get-WmiObject` to get some information about my operating system. I selected four properties and PowerShell decided to display a table. Now, let's add another property to return.

```
PS C:\Users\DKTCLAPTOP> get-wmiobject -class win32_operatingsystem | select
pscomputername,caption,osarch*,registereduser,version



PSComputerName : DKLAPTOP99
caption        : Microsoft Windows 10 Enterprise
OSArchitecture : 64-bit
registereduser : DKTCLAPTOP
version        : 10.0.14393
```

Whoa, now we get a list. What gives?

Well here's how PowerShell decides, by default, whether to display a list or table:

- If showing four or fewer properties, show a table
- If showing five or more properties, show a list

That's it, that's how PowerShell decides by default whether to show you a list or table.

Chapter 12

The Difference Between Get-Member and .GetType() in PowerShell

By: Thomas Rayner – MVP

Recently, I was helping someone in a forum who was trying to figure out what kind of object their command was returning. They knew about the standard cmdlets people suggest when you're getting started (Get-Help, Get-Member, and Get-Command), but couldn't figure out what was coming back from a specific command.

In order to make this a more generic example, and to simplify it, let's approach this differently. Say I have these two objects where one is a string and the other is an array of two strings.

```
$thing1 = 'This is an item'
$thing2 = @('This is another item','This is one more item')
$thing1; $thing2
```

The third line shows you what you get if you write these out to the screen.

```
PS C:\Users\DKTCLAPTOP> $thing1 = 'This is an item'
$thing2 = @('This is another item','This is one more item')
$thing1; $thing2
This is an item
```

```
This is another item
This is one more item
```

It looks like three separate strings, right? Well we should be able to dissect these with Get-Member to get to the bottom of this and identify the types of objects these are. After all, one is a string and the other is an array, right?

```
$thing1 | Get-Member
```

```
PS C:\Users\DKTCLAPTOP> $thing1 | Get-Member



   TypeName: System.String


Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone(), System.Object
<OUTPUT Truncated>
```

Dang, $thing2 is an array but Get-Member is still saying the TypeName is System.String. What's going on?

Well, the key here is what we're doing is writing the output of $thing2 into Get-Member. So the output of $thing2 is two strings, and that's what's actually hitting Get-Member. If we want to see what kind of object $thing2 really is, we need to use a method that's built into every PowerShell object: GetType().

```
$thing2.GetType()
```

```
PS C:\Users\DKTCLAPTOP> $thing2.GetType()


IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

There you go. $thing2 is a System.Array object, just like we thought.

Chapter 13

Dynamically Create Preset Tests for PowerShell

By: Thomas Rayner – MVP

The Pester people don't really recommend this, but, I find it can be really helpful sometimes. What I'm talking about is dynamically creating assertions inside of a Pester test using PowerShell. While I think you should strive to follow best practices, sometimes what's best for you isn't always a best practice, and as long as you know what you're doing, I think you can get away with bending the rules sometimes. Don't tell anyone I said that.

Say you had a requirement to make sure that a function you wrote performed math, correctly. Maybe it looks like this.

```
function Get-Square {
    param (
        [int]$Number
    )
    $result = $Number * $Number
    $result
}
```

This will just get the square of the number we pass it. Your test might look like this.

```
describe 'Get-Square' {
    it 'squares 1' {
/4
```

```
        Get-Square 1 | Should Be 1
    }

    it 'squares 2' {
        Get-Square 2 | Should Be 4
    }

    it 'squares 3' {
        Get-Square 3 | Should Be 9
    }
}
```

```
PS C:\Users\DKTCLAPTOP> describe 'Get-Square' {
    it 'squares 1' {
        Get-Square 1 | Should Be 1
    }

    it 'squares 2' {
        Get-Square 2 | Should Be 4
    }

    it 'squares 3' {
        Get-Square 3 | Should Be 9
    }
}
Describing Get-Square
 [+] squares 1 749ms
 [+] squares 2 152ms
 [+] squares 3 14ms
```

This would work. It would test your function correctly, and give you all the feedback you expect. There's another way to do this, though. Check out this next example.

```
describe 'Get-Square' {
    $tests = @(
        @(1,1),
        @(2,4),
        @(3,9)
    )
    foreach ($test in $tests) {
        it "squares #($test[0])" {
            Get-Square $test[0] | Should Be $test[1]
        }
    }
}
```

```
PS C:\Users\DKTCLAPTOP> describe 'Get-Square' {
    $tests = @(
        @(1,1),
        @(2,4),
        @(3,9)
    )
    foreach ($test in $tests) {
        it "squares #($test[0])" {
            Get-Square $test[0] | Should Be $test[1]
        }
```

```
    }
}
Describing Get-Square
 [+] squares #(1 1[0]) 35ms
 [+] squares #(2 4[0]) 42ms
 [+] squares #(3 9[0]) 17ms
```

This particular example gets more complicated, but shows you what I'm talking about. $tests is an array of smaller arrays where the first number is the number to be squared, and the second number is the answer we expect. Then for each test (array in $tests), I'm generating a new it assertion. Neat, right?

Yes, in this particular situation, we ignored Pester test cases, which would have worked here too. This was just a silly example to show how you might tackle this problem differently, or in a situation where test cases wouldn't work for you.

Chapter 14

Piping PowerShell Output into Bash

By: Thomas Rayner – MVP

With Windows 10, you can install Bash on Windows. Cool, right? Having Bash on Windows goes a long way towards making Windows a more developer-friendly environment and opens a ton of doors. The one I'm going to show you today is more of a novelty than anything else, but maybe you'll find something neat to do with it.

If you've been around PowerShell, you're used to seeing the pipe character (|) used to pass the output from one command into the input of another. What you can do now, kind of, is pass the output of a PowerShell command into the input of a Bash command. Here's an example. Get ready for this biz.

```
Get-ChildItem c:\temp\demo | foreach-object { bash -c "echo $($_.Name) | awk
/\.csv/" }
```

In my c:\temp\demo folder, I have three files, two of which are CSVs. In an attempt to be super inefficient, I am piping the files in that directory into a foreach-object loop and using Bash to tell me which ones end in .csv, using awk. This is hardly the best way to do this, but it gives you an idea of how you can start to intermingle these two shells.

Chapter 15

How to List All the Shares on a Server using PowerShell

By: Thomas Rayner – MVP

There's a few ways to get all of the shared folders on a server, but not all of them work for all versions of Windows Server. You can use the `Get-SmbShare` cmdlet, or you can make CIM/WMI do the work for you. I'll show you what I prefer, though.

To use `Get-SmbShare` on a remote computer, you'll create a new CIM session.

```
$ComputerName = 'tccalst01'
New-CimSession -ComputerName $computername -Credential $creds
```

```
PS C:\Windows\system32> $ComputerName = 'tccalst01'
New-CimSession -ComputerName $computername -Credential $creds



Id           : 1
Name         : CimSession1
InstanceId   : 63a37d00-298f-4e82-8aa5-4b40de1e7709
ComputerName : tccalst01
Protocol     : WSMAN
```

Then you can pass that CIM session to `Get-SmbShare`

```
Get-SmbShare -CimSession $(get-cimsession -id 1)
```

```
PS C:\Windows\system32> Get-SmbShare -CimSession $(get-cimsession -id 1)


Name                                      ScopeName                                Path
Description

----                                      ---------                                ----
-----------

ADMIN$                                    *                                        C:\Windows
Remote Admin

C$                                        *                                        C:\
Default share

IPC$                                      *
Remote IPC

NETLOGON                                  *
C:\Windows\SYSVOL\sysvol\triconts.com\... Logon server share

SYSVOL                                    *
C:\Windows\SYSVOL\sysvol                  Logon server share
```

But what if the server is (heaven forbid!) older than Windows Server 2012R2? Well, you'd get an error telling you "`Get-Cimclass`: The WS-Management service cannot process the request. The CIM namespace win32_share is invalid.". That won't do.

Well, luckily for those older servers, you can use `Get-WmiObject` to retrieve this information.

```
$oldcomp = 'tccalst01'
Get-WmiObject -Class win32_share -ComputerName $oldComp -Credential $creds
```

Chapter 16

Get a ServiceNow User Using PowerShell

By: Thomas Rayner – MVP

Ever wanted to work with ServiceNow via PowerShell? Let me show you some basics like fetching a user.

Let's jump into some code first and I'll break down what I'm doing.

```
$user = $Credential.Username
$pass = $Credential.GetNetworkCredential().Password
$base64AuthInfo =
[Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("{0}:{1}" -f $user,
$pass)))

$headers = New-Object
"System.Collections.Generic.Dictionary[[String],[String]]"
$headers.Add('Authorization',('Basic {0}' -f $base64AuthInfo))
$headers.Add('Accept','application/json')

$uri = "https://$SubscriptionSubDomain.service-
now.com/api/now/v1/table/sys_user?sysparm_query=user_name=$Username"

$response = Invoke-WebRequest -Headers $headers -Method "GET" -Uri $uri
$result = ($response.Content | ConvertFrom-Json).Result
```

This isn't my favorite way of handling credentials, but it's what the ServiceNow documentation recommends and, well, it works.

On line 9, I'm constructing my URI using a variable holding my subdomain and another variable for the username I'm interested in ($SubscriptionSubDomain and $Username respectively).

Then on lines 11 and 12, I am invoking the web request to get the information about the user, and parsing the result. I can then use the $result variable later in my script.

This has been particularly helpful for me when I'm trying to figure out the sys_id (ServiceNow's unique ID) for a specific user and all I know is their username.

Chapter 17

Add a Work Note to a ServiceNow Incident with PowerShell

By: Thomas Rayner – MVP

Recently, I've been working more with ServiceNow and writing scripts and tools which sometimes interact with it. One of the things that I find myself doing a lot is using PowerShell to add a work note to an incident. Luckily, ServiceNow has an API that you can use to interact with it and do this (among many other things).

Since I know that all my information is stored in the Incident table, it's not too many steps to get an incident out of ServiceNow if I have the incident number.

```
$user = $Credential.Username

$pass = $Credential.GetNetworkCredential().Password

$base64AuthInfo =
[Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("{0}:{1}" -f $user,
$pass)))


$headers = New-Object
"System.Collections.Generic.Dictionary[[String],[String]]"

$headers.Add('Authorization',('Basic {0}' -f $base64AuthInfo))

$headers.Add('Accept','application/json')


$uriGetIncident = "https://$SubDomain.service-
now.com/api/now/table/incident?sysparm_query=number%3D$SNIncidentNumber&sysparm_
fields=&sysparm_limit=1"
```

```
$responseGetIncident = Invoke-WebRequest -Headers $headers -Method "GET" -Uri
$uriGetIncident

$resultGetIncident = ($responseGetIncident.Content | ConvertFrom-Json).Result
```

Assuming I already created a credential object named $Credential to hold my ServiceNow creds, I can add do some encoding to assemble them in a way that I can add them to the header of the request I'm about to make. I'm doing that on the first three lines.

On lines 5 – 7, I'm constructing those headers. So far, I'm following all the PowerShell examples given in the ServiceNow documentation.

Line 9 is where I create the URI for the incident get request. You'll notice I have a variable for both the subdomain (will be unique for your instance of ServiceNow) and the ServiceNow incident number.

Lines 10 and 11 get the incident and parse the results of my request.

Now I can add some work notes.

```
$workNotesBody = @"

{"work_notes":"$Message"}

"@


$uriPatchIncident = "https://$SubDomain.service-
now.com/api/now/table/incident/$($resultGetIncident.sys_id)"

$null = Invoke-WebRequest -Headers $headers -Method "PATCH" -Uri
$uriPatchIncident -body $workNotesBody
```

On lines 1 – 3, I'm making the body of my patch request, to say that I'm adding the value of $Message into the work_notes field of my incident. Line 5 is where I make the URI for this patch activity, using the sys_id that came out of the get query I performed earlier.

On line 5, I'm muting the output of the web request to add the work notes to the incident. I'm reusing the headers I set up for the get query.

Chapter 18

Use PowerShell to see how many items are in a Directory

By: Thomas Rayner – MVP

Here's a way to see how many items are in a directory, using PowerShell.

As you likely know, you can use `Get-ChildItem` to get all the items in a directory. Did you know, however, that you can have PowerShell quickly count how many files and folders there are?

```
(Get-ChildItem -Path c:\temp\).count
```

```
PS C:\WINDOWS\system32> (Get-ChildItem -Path c:\temp\).count
22
```

I probably could have counted the files in this specific directory pretty easily myself, since there's only 3 of them. If you want to see how many files are in an entire folder structure, use the -Recurse flag to go deeper.

You can do this with any output from a cmdlet when it's returned in an array of objects. Check this out.

```
(Get-AdUser -filter "Name -like 'Cristal *'").count
```

```
PS C:\Windows\system32> (Get-AdUser -filter "Name -like 'Cristal *'").count
7
```

In my test Active Directory, there are 7 AD users with a name that matches the pattern “Cristal *”.

Chapter 19

Add a Column to a CSV using PowerShell

By: Thomas Rayner – MVP

Say you have a CSV file full of awesome, super great, amazing information. It’s perfect, except it’s missing a column. Luckily, you can use Select-Object along with the other CSV cmdlets to add a column.

In our example, let’s say that you have a CSV with two columns “ComputerName” and “IPAddress” and you want to add a column for “Port3389Open” to see if the port for RDP is open or not. It’s only a few lines of code from being done.

```
$servers = Import-Csv C:\Temp\demo\servers.csv

$servers
```

```
PS C:\WINDOWS\system32> $servers = Import-Csv C:\Temp\demo\servers.csv

$servers


Name      IPAddress
```

```
----       ---------
Server01   10.1.2.10
Server02   10.1.2.11
TCCALST01  10.10.1.252
```

Now, let's borrow some code from my post on calculated properties in PowerShell to help us add this column and my post on seeing if a port is open using PowerShell to populate the data.

```
$servers = $servers | Select-Object -Property *, @{label = 'Port3389Open';
expression = {(Test-NetConnection -ComputerName $_.Name -Port
3389).TcpTestSucceeded}}
```

```
PS C:\WINDOWS\system32> $servers = $servers | Select-Object -Property *, @{label
= 'Port3389Open'; expression = {(Test-NetConnection -ComputerName $_.Name -Port
3389).TcpTestSucceeded}}
WARNING: TCP connect to Server01:3389 failed
WARNING: Ping to Server01 failed -- Status: TimedOut
WARNING: TCP connect to Server02:3389 failed
WARNING: Ping to Server02 failed -- Status: TimedOut
```

```
$servers | Export-Csv -Path c:\temp\demo\servers-and-port-data.csv -
NoTypeInformation
$Servers
```

```
PS C:\WINDOWS\system32> $Servers


Name       IPAddress    Port3389Open
----       ---------    ------------
Server01   10.1.2.10          False
Server02   10.1.2.11          False
TCCALST01  10.10.1.252         True
```

Chapter 20

Diagnosing slow PowerShell Load Times

By: Thomas Rayner – MVP

I could write an entire book on "why does my PowerShell console take so long to load?" but I don't want to write that book. Instead, here's a way to make sure the reason your console is loading slowly isn't because of something dumb.

When you launch PowerShell, one of the things that happens is that your profile is loaded. Your profile is basically its own script that runs to setup and configure your environment before you start using it. I use mine to define some custom aliases, functions, import some modules, and set my prompt up. You can see what your profile is doing by running notepad $profile. This will open your profile in notepad (but you can use the ISE or Visual Studio Code or Notepad++ etc. if you prefer).

There is more than one profile used by PowerShell depending on how you're running PowerShell, and $profile will always refer to the one that's currently applied to you. If you run that command above and are told that there's no such file, it means don't have anything configured in your PowerShell profile.

Keep in mind, there could be a lot of other reasons that your console loads slowly. This is just a quick way to clear out any dumb code from your profile.

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

C:\Users\dkawula_1\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.p
s1
```

Chapter 21

Use Test-NetConnection in PowerShell to see if a Port is Open

By: Thomas Rayner – MVP

The days of using ping.exe to see if a host is up or down are over. Your network probably shouldn't allow ICMP to just fly around unaddressed, and your hosts probably shouldn't return ICMP echo request (ping) messages either. So how do I know if a host is up or not?

Well, it involves knowing about what your host actually does. What ports are supposed to be open? Once you know that, you can use `Test-NetConnection` in PowerShell to check if the port is open and responding on the host you're interested in.

```
$Nodes = 'tccalst01','tccaldc04'
$nodes
$Nodes | % {Test-NetConnection -Computername $_.ToString() -Port 3389}
```

```
PS C:\Windows\system32> $Nodes = 'tccalst01','tccaldc04'
$nodes
$Nodes | % {Test-NetConnection -Computername $_.ToString() -Port 3389}
tccalst01
tccaldc04
```

```
ComputerName            : tccalst01
RemoteAddress           : 10.10.1.252
RemotePort              : 3389
InterfaceAlias          : Ethernet
SourceAddress           : 10.10.1.247
PingSucceeded           : True
PingReplyDetails (RTT) : 1 ms
TcpTestSucceeded        : True

ComputerName            : tccaldc04
RemoteAddress           : 10.10.1.249
RemotePort              : 3389
InterfaceAlias          : Ethernet
SourceAddress           : 10.10.1.247
PingSucceeded           : True
PingReplyDetails (RTT) : 0 ms
TcpTestSucceeded        : True
```

Here I just checked if port 3389 (for RDP) is open or not. Looks like it is.

Chapter 22

Use PowerShell to find out How Long it is until Christmas

By: Thomas Rayner – MVP

It's October (when I'm writing this) which means Christmas is right around the corner! Maybe not. How long is it until Christmas, anyway? Well, PowerShell can tell us if we get the date of Christmas and subtract today's date from it.

```
(Get-Date 'December 25') - (Get-Date)
```

```
PS C:\Windows\system32> (Get-Date 'December 25') - (Get-Date)



Days              : 71
Hours             : 13
Minutes           : 33
Seconds           : 10
Milliseconds      : 736
Ticks             : 61831907360482
TotalDays         : 71.5647075931505
TotalHours        : 1717.55298223561
TotalMinutes      : 103053.178934137
TotalSeconds      : 6183190.7360482
TotalMilliseconds : 6183190736.0482
```

Only 6183190736.0482 more milliseconds until Christmas!

Chapter 23

Use PowerShell to Figure out “What day of the week” x number of days from now

By: Thomas Rayner – MVP

There’s lots of fun things you can do with datetime objects in PowerShell, and using the `Get-Date`

cmdlet. Here’s one of them.

Say you want to know what day of the week it will be some arbitrary number of days from now. It’s pretty easy.

```
(Get-Date).AddDays(39).DayOfWeek
```

```
PS C:\WINDOWS\system32> (Get-Date).AddDays(39).DayOfWeek
Wednesday
```

At the time I write this, it looks like in 39 days, it’ll be Wednesday.

Chapter 24

Using Get-Member to Explore Objects

By: Thomas Rayner – MVP

I previously wrote about using Select-Object to explore PowerShell objects. Now, I am going to quickly cover using `Get-Member` to do the same.

Let's say you're using `Get-CimInstance` to get information about the operating system. You might do something like this.

```
Get-CimInstance -ClassName win32_operatingsystem
```

```
PS C:\WINDOWS\system32> Get-CimInstance -ClassName win32_operatingsystem


SystemDirectory     Organization BuildNumber RegisteredUser SerialNumber
Version

---------------     ------------ ----------- -------------- ------------
-------

C:\WINDOWS\system32              14393       DKTCLAPTOP     00329-00000-00003-
AA795 10.0.14393
```

As is the case with our example last week, there's more stuff returned and available to us than what is returned by default. Let's use `Get-Member` to see what it all is.

```
Get-CimInstance -ClassName win32_operatingsystem | get-member
```

```
PS C:\WINDOWS\system32> Get-CimInstance -ClassName win32_operatingsystem | get-
member




   TypeName:
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_OperatingSystem


Name                                        MemberType  Definition

----                                        ----------  ----------

Clone                                       Method      System.Object
ICloneable.Clone()

Dispose                                     Method      void Dispose(), void
IDisposable.Dispose()

Equals                                      Method      bool Equals(System.Object
obj)

GetCimSessionComputerName                   Method      string
GetCimSessionComputerName()

GetCimSessionInstanceId                     Method      guid
GetCimSessionInstanceId()

GetHashCode                                 Method      int GetHashCode()

GetObjectData                               Method      void
GetObjectData(System.Runtime.Serialization.SerializationInfo info,
System.Runtime.Serialization.StreamingContext context), void ...

GetType                                     Method      type GetType()

ToString                                    Method      string ToString()

BootDevice                                  Property    string BootDevice {get;}

BuildNumber                                 Property    string BuildNumber {get;}

BuildType                                   Property    string BuildType {get;}

Caption                                     Property    string Caption {get;}

CodeSet                                     Property    string CodeSet {get;}

CountryCode                                 Property    string CountryCode {get;}
```

```
CreationClassName                          Property    string CreationClassName
{get;}

<OUTPUT TRUNCATED>
```

Holy smokes, there's a lot of stuff there. As with Select-Object, you can see all the different properties that exist in this object. The big difference here is that you can see all the different methods this object comes with, too. You could store this information in a variable and then invoke the .HashCode() on it and see the output of that, like this.

```
$osInfo = Get-CimInstance -ClassName win32_operatingsystem

$osInfo.GetHashCode()
```

```
PS C:\WINDOWS\system32> $osInfo = Get-CimInstance -ClassName
win32_operatingsystem


PS C:\WINDOWS\system32> $osInfo.GetHashCode()

32638546
```

There's a lot of examples of methods that are more interesting than this, but you can play with it and make this work for you.

Chapter 25

Using Select-Object to Explore Objects

By: Thomas Rayner – MVP

When you're first getting started with PowerShell, you may not be aware that sometimes when you run a command to get data, the information returned to the screen is not ALL the information that the command actually returned.

Let me clarify with an example. If you run the `Get-ChildItem` cmdlet, you'll get a bit of information back about all the files in whichever directory you specified.

```
Get-ChildItem c:\temp\demo
```

```
PS C:\WINDOWS\system32> Get-ChildItem c:\temp\demo



    Directory: C:\temp\demo



Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----       10/14/2017  10:04 AM            133 servers-and-port-data.csv
-a----       10/14/2017  10:02 AM             77 servers.csv
```

This is not all the data that got returned, though. There are far more properties than just Mode, LastWriteTime, Length and Name to be examined. What are they? Well, we can pipe this cmdlet into `Select-Object -Property *` to see them.

```
Get-ChildItem c:\temp\demo | Select-Object -Property *
```

```
PS C:\WINDOWS\system32> Get-ChildItem c:\temp\demo | Select-Object -Property *



PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\temp\demo\servers-
and-port-data.csv
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\temp\demo
PSChildName       : servers-and-port-data.csv
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : False
Mode              : -a----
VersionInfo       : File:             C:\temp\demo\servers-and-port-data.csv
                    InternalName:
                    OriginalFilename:
                    FileVersion:
                    FileDescription:
                    Product:
                    ProductVersion:
                    Debug:            False
                    Patched:          False
                    PreRelease:       False
                    PrivateBuild:     False
                    SpecialBuild:     False
                    Language:
```

```
BaseName          : servers-and-port-data
Target            : {}
LinkType          :
Name              : servers-and-port-data.csv
Length            : 133
DirectoryName     : C:\temp\demo
Directory         : C:\temp\demo
IsReadOnly        : False
Exists            : True
FullName          : C:\temp\demo\servers-and-port-data.csv
Extension         : .csv
CreationTime      : 10/14/2017 10:04:43 AM
CreationTimeUtc   : 10/14/2017 4:04:43 PM
LastAccessTime    : 10/14/2017 10:04:43 AM
LastAccessTimeUtc : 10/14/2017 4:04:43 PM
LastWriteTime     : 10/14/2017 10:04:43 AM
LastWriteTimeUtc  : 10/14/2017 4:04:43 PM
Attributes        : Archive


PSPath            :
Microsoft.PowerShell.Core\FileSystem::C:\temp\demo\servers.csv
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\temp\demo
PSChildName       : servers.csv
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : False
Mode              : -a----
VersionInfo       : File:             C:\temp\demo\servers.csv
                    InternalName:
                    OriginalFilename:
                    FileVersion:
```

```
                    FileDescription:
                    Product:
                    ProductVersion:
                    Debug:            False
                    Patched:          False
                    PreRelease:       False
                    PrivateBuild:     False
                    SpecialBuild:     False
                    Language:

BaseName          : servers
Target            : {}
LinkType          :
Name              : servers.csv
Length            : 77
DirectoryName     : C:\temp\demo
Directory         : C:\temp\demo
IsReadOnly        : False
Exists            : True
FullName          : C:\temp\demo\servers.csv
Extension         : .csv
CreationTime      : 10/14/2017 9:58:17 AM
CreationTimeUtc   : 10/14/2017 3:58:17 PM
LastAccessTime    : 10/14/2017 10:01:21 AM
LastAccessTimeUtc : 10/14/2017 4:01:21 PM
LastWriteTime     : 10/14/2017 10:02:55 AM
LastWriteTimeUtc  : 10/14/2017 4:02:55 PM
Attributes        : Archive
```

Look at all that goodness. You can select specific properties by replacing the star with the names of the properties you want to see.

```
Get-ChildItem c:\temp\demo | Select-Object -Property Name, Attributes,
IsReadOnly
```

```
PS C:\WINDOWS\system32> Get-ChildItem c:\temp\demo | Select-Object -Property
Name, Attributes, IsReadOnly


Name                      Attributes IsReadOnly
----                      ---------- ----------
servers-and-port-data.csv    Archive      False
servers.csv                  Archive      False
```

Happy scripting!

Chapter 26

Can PowerShell Parameters Belong to Multiple Sets?

By: Thomas Rayner – MVP

Say you've got a function that takes three parameters: Username, ComputerName and SessionName, but you don't want someone to use ComputerName and SessionName at once. You decide to put them in separate parameter sets. Awesome, except you want Username to be a part of both parameter sets and it doesn't look like you can specify more than one.

This will generate an error:

```
function Do-Thing {
    [CmdletBinding()]
    param (
    [Parameter( ParameterSetName = 'Computer','Session' )][string]$Username,
    [Parameter( ParameterSetName = 'Computer' )][string]$ComputerName,
    [Parameter( ParameterSetName = 'Session' )][PSSession]$SessionName
    )
# Other code
}
```

So how do you make a parameter a member of more than one parameter set? You need more [Parameter()] qualifiers.

```
function Do-Thing {
```

```
    [CmdletBinding()]
    param (
    [Parameter( ParameterSetName = 'Computer' )]
    [Parameter( ParameterSetname = 'Session' )]
    [string]$Username,

    [Parameter( ParameterSetName = 'Computer' )][string]$ComputerName,
    [Parameter( ParameterSetName = 'Session' )][PSSession]$SessionName
    )
# Other code
}
```

They chain together and you now $Username is a part of both parameter sets.

Chapter 27

Opening an Exchange Online Protection Shell

By: Thomas Rayner – MVP

I built a PowerShell function in my profile to connect quickly to Exchange Online. That's great, but what if you also want to manage Exchange Online Protection (EOP) from a PoweShell console? Well it turns out to be pretty easy.

```
$cred = Get-Credential

$s = New-PSSession -ConfigurationName Microsoft.Exchange -ConnectionUri
https://outlook.office365.com/powershell-liveid/ -Credential $cred -
Authentication Basic -AllowRedirection

import-pssession $s
```

```
PS C:\WINDOWS\system32> $cred = Get-Credential

$s = New-PSSession -ConfigurationName Microsoft.Exchange -ConnectionUri
https://outlook.office365.com/powershell-liveid/ -Credential $cred -
Authentication Basic -AllowRedirection

import-pssession $s

cmdlet Get-Credential at command pipeline position 1

Supply values for the following parameters:

WARNING: The names of some imported commands from the module 'tmp_5pt4o42k.j34'
include unapproved verbs that might make them less discoverable. To find the
commands with unapproved verbs, ru

n the Import-Module command again with the Verbose parameter. For a list of
approved verbs, type Get-Verb.
```

```
ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     1.0        tmp_5pt4o42k.j34                    {Add-
AvailabilityAddressSpace, Add-DistributionGroupMember, Add-
MailboxFolderPermission, Add-MailboxLocation...}
```

Chapter 28

Import Active Directory Module into Windows PE

By: Mick Pletcher – MVP

One thing I have been wanting to have is access to active directory in a WinPE environment. The main reason I want it is to be able to delete systems from active directory during a build. When I first started researching, I found this blog that guided me on writing this script. The blog tells how to inject the AD module into the WIM file. That is fine, but do you really want to do that every time you generate a new WIM file? I don't. I started testing to see if the directories could be copied into the WinPE environment while it was running without the need of a reboot. It worked. Currently, this script only makes the Active Directory module available in the WinPE environment. I am going to write more scripts to take advantage of the AD module.

To use this script, you will need to place it on a network share, or if you are using WinPE, you can place it within the scripts folder of the DeploymentShare so the image will have access to it. I wrote this with four parameters so that you can use the domain username and password within a task sequence without putting it inside the script to possibly expose it. The username and password give the script access to map to the NetworkPath. The NetworkPath points to the location where the Active Directory components reside to copy over to the WinPE environment. The DriveLetter pertains to the drive letter you wish for the script to use when mapping to the NetworkPath. If you want, you could enter default values for the parameters if you want.

The next thing you will need to do is to create the source folders on the NetworkPath, which will contain all of the files.

For 32-Bit WinPE, create the following directories on your NetworkPath. This is what my source directory looks like:

ActiveDirectory	8/31/2015 1:10 PM	File folder
Microsoft.ActiveDirectory.Management	8/31/2015 1:10 PM	File folder
Microsoft.ActiveDirectory.Management.Resources	8/31/2015 1:10 PM	File folder
msil_microsoft-windows-d..ivecenter.resources_31bf3856ad364e35_10.0.10514.4_en-us_c9da70497b67e587	8/31/2015 1:09 PM	File folder
x86_microsoft.activedirectory.management_31bf3856ad364e35_10.0.10514.4_none_4947062d67e618d7	8/31/2015 1:09 PM	File folder

NOTE: The last two directories will have different names as the module is updated by Microsoft. You will have to search for the first part of the name to find them if it changes. That is why in the script I have it to search for the name of the directory knowing that it might change.

For 32-bit WinPE, copy the following directories from a Windows 10 machine to the appropriate directories created above. Make sure you copy all subdirectories along with the full contents:

- %windir%\System32\WindowsPowerShell\v1.0\Modules\ActiveDirectory
- %windir%\Microsoft.NET\assembly\GAC_32\Microsoft.ActiveDirectory.Management
- %windir%\Microsoft.NET\assembly\GAC_32\Microsoft.ActiveDirectory.Management.Resources
- %windir%\WinSxS\x86_microsoft.activedirectory.management_31bf3856ad364e35_6.3.9431.0_none_b85eb2e785c286ef
- %windir%\WinSxS\msil_microsoft-windows-d..ivecenter.resources_31bf3856ad364e35_6.3.9431.0_en-us_38f21d039944539f

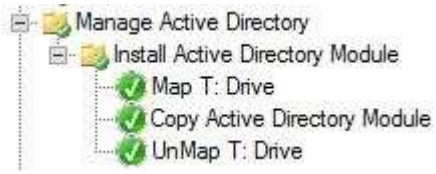
For 64-Bit WinPE, I have included an If statement, but it has not been tested, so I can't guarantee it will work. I am not sure if you still need to copy the 32-bit folders also, or if they can be

removed and just the 64-bit folders installed. Here is the list of folders to copy from a Windows 10 x64 system:

- %windir%\SysWOW64\WindowsPowerShell\v1.0\Modules\ActiveDirectory
- %windir%\Microsoft.NET\assembly\GAC_64\Microsoft.ActiveDirectory.Management
- %windir%\Microsoft.NET\assembly\GAC_64\Microsoft.ActiveDirectory.Management.Resources
- %windir%\WinSxS\amd64_microsoft.activedir..anagement.resources_31bf3856ad364e35_6.3.9431.0_en-us_fb186ae865900ae8

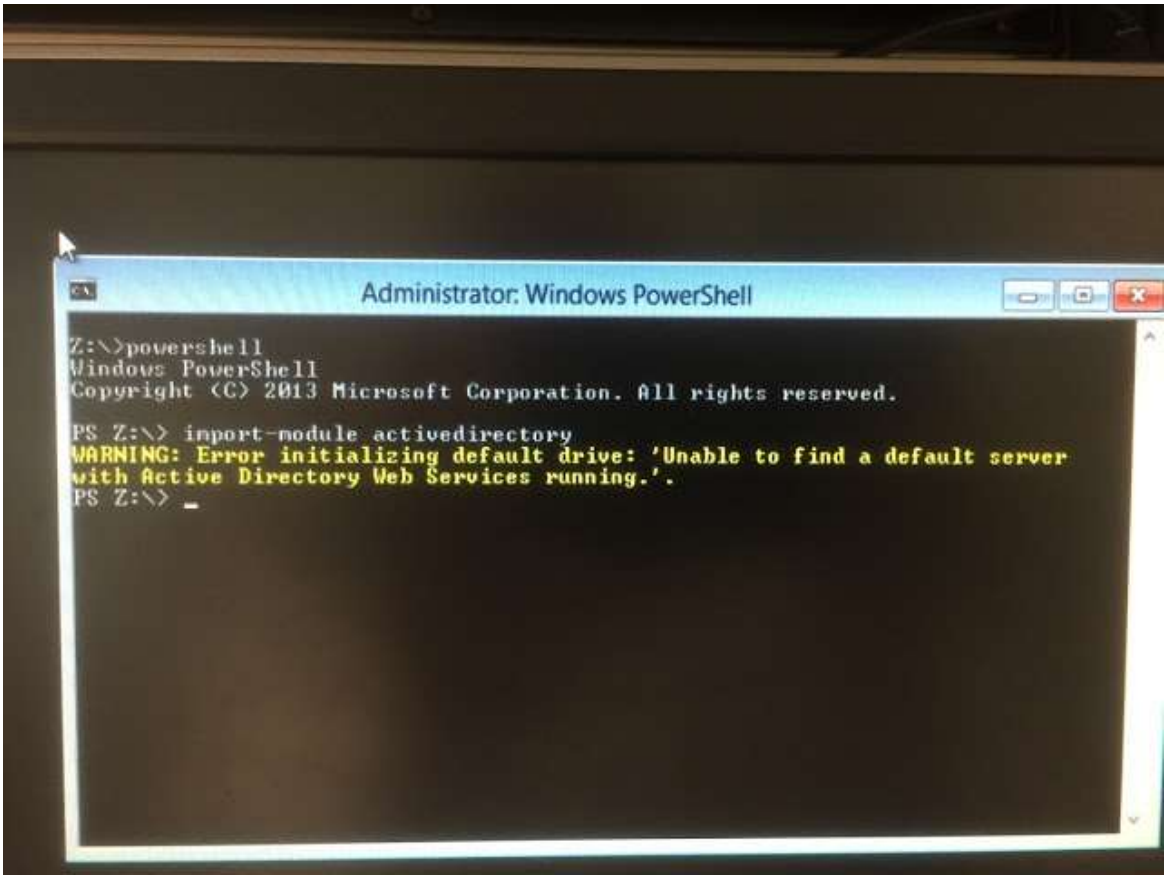
As for the last 64-bit entry above, I have included a variable in the script to grab the name of the directory as the last part will likely change upon future updates to the PowerShell AD module.

Below is how I put the script into the task sequence build. I first map a T: drive to the location of where the directories above exist. I then execute the powershell script and finally unmap the T: drive.



After you get these source files copied to a network location, you can now use the script below to run during the WinPE environment. You can see the pop-up windows in the background as it robocopies the directories over to WinPE.

One thing you will encounter when executing the script is that it will give you the following warning when you import the module:



I racked my brain last week trying to get this to go away. I was trying to use the new-psdrive to open a connection to the active directory server and I just couldn't get it to work. I finally posted to the Facebook PowerShell group and one advised me to ignore the message and use the -server parameter for each cmdlet. That works. You can ignore this message. I ran the Get-ADComputer cmdlet and specificed the AD server in the -server parameter. It worked perfectly.

```
:  <#
       .SYNOPSIS
            Install PowerShell Active Directory Module
```

```
        .DESCRIPTION

             Copies the PowerShell Active Directory Module to the WinPE
environment. This allows the use of the PowerShell module without having to
mount, inject the directories, and dismount a WIM everytime a new WIM is
generated.


        .PARAMETER DomainUserName

             Username with domain access used to map drives


        .PARAMETER DomainPassword

             Domain password used to map network drives


        .PARAMETER NetworkPath

             Network path to map where the Active Directory PowerShell module
exists


        .PARAMETER DriveLetter

             Drive letter mapping where the PowerShell Active Directory module
files exists


        .NOTES

===========================================================================

             Created with: SAPIEN Technologies, Inc., PowerShell Studio 2016
v5.2.119

             Created on:   4/8/2016 12:41 PM

             Created by:   Mick Pletcher

             Organization:

             Filename:     InstallActiveDirectoryModule.ps1

===========================================================================
  #>
  [CmdletBinding()]
```

```
param
(
	[string]
	$DomainUserName,
	[string]
	$DomainPassword,
	[string]
	$NetworkPath,
	[string]
	$DriveLetter
)

function Copy-Folder {
<#
	.SYNOPSIS
		Copy Folder

	.DESCRIPTION
		Copy folder to destination

	.PARAMETER SourceFolder
		A description of the SourceFolder parameter.

	.PARAMETER DestinationFolder
		A description of the DestinationFolder parameter.

	.EXAMPLE
		PS C:\> Copy-Folder -SourceFolder 'Value1' -
DestinationFolder 'Value2'

	.NOTES
```

```
            Additional information about the function.
  #>

        [CmdletBinding()]
        param
        (
            [string]
            $SourceFolder,
            [string]
            $DestinationFolder
        )

        $Executable = $env:windir + "\system32\Robocopy.exe"
        $Switches = $SourceFolder + [char]32 + $DestinationFolder + [char]32 +
"/e /eta /mir"
        Write-Host "Copying "$SourceFolder"....." -NoNewline
        $ErrCode = (Start-Process -FilePath $Executable -ArgumentList $Switches -
Wait -Passthru).ExitCode
        If (($ErrCode -eq 0) -or ($ErrCode -eq 1)) {
            Write-Host "Success" -ForegroundColor Yellow
        } else {
            Write-Host "Failed with error code"$ErrCode -ForegroundColor Red
        }
  }

  function Get-Architecture {
  <#
        .SYNOPSIS
            Get-Architecture

        .DESCRIPTION
```

```
            Returns whether the system architecture is 32-bit or 64-bit

        .EXAMPLE
            Get-Architecture

        .NOTES
            Additional information about the function.
  #>

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

        $OSArchitecture = Get-WmiObject -Class Win32_OperatingSystem | Select-
Object OSArchitecture
        $OSArchitecture = $OSArchitecture.OSArchitecture
        Return $OSArchitecture
        #Returns 32-bit or 64-bit
  }


  function New-NetworkDrive {
  <#
        .SYNOPSIS
            Map network drive

        .DESCRIPTION
            Map the network drive for copying down the PowerShell Active
Directory files to the WinPE environment

        .EXAMPLE
            PS C:\> New-NetworkDrive
```

```
        .NOTES
             Additional information about the function.
  #>

        [CmdletBinding()]
        param ()

        $Executable = $env:windir + "\system32\net.exe"
        $Switches = "use" + [char]32 + $DriveLetter + ":" + [char]32 +
$NetworkPath + [char]32 + "/user:" + $DomainUserName + [char]32 +
$DomainPassword
        Write-Host "Mapping"$DriveLetter":\ drive....." -NoNewline
        $ErrCode = (Start-Process -FilePath $Executable -ArgumentList $Switches -
Wait -Passthru).ExitCode
        If ((Test-Path $DriveLetter":\") -eq $true) {
             Write-Host "Success" -ForegroundColor Yellow
        } else {
             Write-Host "Failed" -ForegroundColor Yellow
        }
  }

  function Remove-NetworkDrive {
  <#
        .SYNOPSIS
             Delete the mapped network drive

        .DESCRIPTION
             Delete the mapped network drive

        .EXAMPLE
                        PS C:\> Remove-NetworkDrive
```

```
        .NOTES
             Additional information about the function.
  #>

        [CmdletBinding()]
        param ()

        $Executable = $env:windir + "\system32\net.exe"
        $Switches = "use" + [char]32 + $DriveLetter + ":" + [char]32 + "/delete"
        Write-Host "Deleting"$DriveLetter":\ drive....." -NoNewline
        $ErrCode = (Start-Process -FilePath $Executable -ArgumentList $Switches -
Wait -Passthru).ExitCode
        If ((Test-Path $DriveLetter":\") -eq $true) {
             Write-Host "Failed" -ForegroundColor Yellow
        } else {
             Write-Host "Success" -ForegroundColor Yellow
        }
  }

  cls
  #Get WinPE Architecture
  $Architecture = Get-Architecture
  #Map network drive to PowerShell active directory module
  New-NetworkDrive
  #Get msil_microsoft-windows-d..ivecenter.resources Directory Name
  $MicrosoftWindowsIvecenterResources = Get-ChildItem $DriveLetter":\" | where {
$_.Attributes -eq 'Directory' } | Where-Object { $_.FullName -like
"*msil_microsoft-windows-d..ivecenter.resources*" }
  #Get WinSxS x86_microsoft.activedirectory.management Name
  $WinSxSMicrosoftActiveDirectoryManagementResources = Get-ChildItem
$DriveLetter":\" | where { $_.Attributes -eq 'Directory' } | Where-Object {
$_.FullName -like "*x86_microsoft.activedirectory.management*" }
```

```
  #Get WinSxS amd64_microsoft.activedir..anagement.resources Name

  $WinSxSMicrosoftActiveDirectoryManagementResources_x64 = Get-ChildItem
$DriveLetter":\" | where { $_.Attributes -eq 'Directory' } | Where-Object {
$_.FullName -like "*amd64_microsoft.activedir..anagement.resources*" }


  #Copy ActiveDirectory Folder

  Copy-Folder -SourceFolder $NetworkPath"\ActiveDirectory" -DestinationFolder
$env:windir"\System32\WindowsPowerShell\v1.0\Modules\ActiveDirectory"

  #Copy Microsoft.ActiveDirectory.Management Folder

  Copy-Folder -SourceFolder $NetworkPath"\Microsoft.ActiveDirectory.Management"
-DestinationFolder
$env:windir"\Microsoft.NET\assembly\GAC_32\Microsoft.ActiveDirectory.Management"

  #Copy Microsoft.ActiveDirectory.Management.Resources Folder

  Copy-Folder -SourceFolder
$NetworkPath"\Microsoft.ActiveDirectory.Management.Resources" -DestinationFolder
$env:windir"\Microsoft.NET\assembly\GAC_32\Microsoft.ActiveDirectory.Management.
Resources"

  #Copy msil_microsoft-windows-d..ivecenter.resources Folder

  Copy-Folder -SourceFolder $NetworkPath"\"$MicrosoftWindowsIvecenterResources -
DestinationFolder $env:windir"\WinSxS\"$MicrosoftWindowsIvecenterResources

  #Copy x86_microsoft.activedirectory.management Folder

  Copy-Folder -SourceFolder
$NetworkPath"\"$WinSxSMicrosoftActiveDirectoryManagementResources -
DestinationFolder
$env:windir"WinSxS\"$WinSxSMicrosoftActiveDirectoryManagementResources


  If ($Architecture -eq "64-bit") {

       #Copy ActiveDirectory x64 Folder

       Copy-Folder -SourceFolder $NetworkPath"\ActiveDirectory" -
DestinationFolder $env:SystemDrive"\"

       #Copy Microsoft.ActiveDirectory.Management x64 Folder

       Copy-Folder -SourceFolder
$NetworkPath"\Microsoft.ActiveDirectory.Management" -DestinationFolder
$env:windir"\Microsoft.NET\assembly\GAC_64\Microsoft.ActiveDirectory.Management"

       #Copy Microsoft.ActiveDirectory.Management.Resources x64 Folder

       Copy-Folder -SourceFolder
$NetworkPath"\Microsoft.ActiveDirectory.Management.Resources" -DestinationFolder
```

```
$env:windir"\Microsoft.NET\assembly\GAC_64\Microsoft.ActiveDirectory.Management.
Resources"

       #Copy amd64_microsoft.activedir..anagement.resources x64 Folder

       Copy-Folder -SourceFolder
$NetworkPath"\"$WinSxSMicrosoftActiveDirectoryManagementResources_x64 -
DestinationFolder
$env:windir"\WinSxS\"$WinSxSMicrosoftActiveDirectoryManagementResources_x64

  }


  #Unmap Network Drive

  Remove-NetworkDrive
```

Chapter 29

Using PowerShell to report on Windows Updates installed during MDT OSD Build

By: Mick Pletcher – MVP

I found it nice to be able to get a clean, filtered report on what Windows updates got installed during the build process. This allows me to inject those updates into the MDT Packages so they get injected into the image before it is laid down to speed the process up. I had published this tool two years ago and decided to revamp it to also include email functionality. The tool has given me a report, but there were times I forgot to look at it after a build completed. This reminds me by sending the report out via email.

The way this tool works is by reading the ZTIWindowsUpdate.log file from the c:\minint\smsosd\osdlogs directory and extracting the list of installed Windows Updates. The script filters out everything that is non-windows updates, such as Dell drivers. It also filters out the windows defender updates since those are cumulative and gets updated on a regular basis.

This is a screenshot of what the logs look like when executed and output to the screen:

```
Administrator: C:\windows\system32\cmd.exe

KBArticle                                  Description
---------                                  -----------
KB2952664                                  Update for Windows 7 for x64-based S...
KB2952664                                  Update for Windows 7 for x64-based S...
KB3161102                                  Update for Windows 7 for x64-based S...
KB3164025                                  Security Update for Microsoft .NET F...
KB3170455                                  Security Update for Windows 7 for x6...
KB3170735                                  Update for Windows 7 for x64-based S...
KB3172605                                  Update for Windows 7 for x64-based S...
KB3177467                                  Update for Windows 7 for x64-based S...
KB3179573                                  Update for Windows 7 for x64-based S...
KB3181988                                  Update for Windows 7 for x64-based S...
KB3184143                                  Update for Windows 7 for x64-based S...
KB3185319                                  Cumulative Security Update for Inter...
KB3188740                                  October  2016 Security and Quality R...
KB4017094                                  Security Update for Microsoft Silver...
KB4019112                                  May  2017 Security and Quality Rollu...
KB4019264                                  2017-05 Security Monthly Quality Rol...
KB4019265                                  2017-05 Preview of Monthly Quality R...
KB4019288                                  May  2017 Preview of Quality Rollup ...

C:\Users\Administrator>
```

Here is a screenshot of what the same report looks like when opened up in Excel.

	A	B
1	KBArticle	Description
2	KB2952664	Update for Windows 7 for x64-based Systems
3	KB2952664	Update for Windows 7 for x64-based Systems
4	KB3161102	Update for Windows 7 for x64-based Systems
5	KB3164025	Security Update for Microsoft .NET Framework 4.6.1 on Windows 7 and Windows Server 2008 R2 for x64
6	KB3170455	Security Update for Windows 7 for x64-based Systems
7	KB3170735	Update for Windows 7 for x64-based Systems
8	KB3172605	Update for Windows 7 for x64-based Systems
9	KB3177467	Update for Windows 7 for x64-based Systems
10	KB3179573	Update for Windows 7 for x64-based Systems
11	KB3181988	Update for Windows 7 for x64-based Systems
12	KB3184143	Update for Windows 7 for x64-based Systems
13	KB3185319	Cumulative Security Update for Internet Explorer 11 for Windows 7 for x64-based Systems
14	KB3188740	October 2016 Security and Quality Rollup for .NET Framework 3.5.1 on Windows 7 SP1 and Windows Server 2008 R2 SP1 for x64
15	KB4017094	Security Update for Microsoft Silverlight
16	KB4019112	May 2017 Security and Quality Rollup for .NET Framework 3.5.1 4.5.2 4.6 4.6.1 4.6.2 on Windows 7 and Server 2008 R2 for x64
17	KB4019264	2017-05 Security Monthly Quality Rollup for Windows 7 for x64-based Systems
18	KB4019265	2017-05 Preview of Monthly Quality Rollup for Windows 7 for x64-based Systems
19	KB4019288	May 2017 Preview of Quality Rollup for .NET Framework 3.5.1 4.5.2 4.6 4.6.1 4.6.2 on Windows 7 and Server 2008 R2 for x64

WindowsUpdatesReport

The script extracts the KB article number and description and writes that information to an object. The object is then displayed on the screen and written to a .CSV file. It is sorted by KBArticle number.

The firm I work at uses Dell machines and in doing so I excluded all Dell drivers from the list. There is also an exclusions.txt file it can read from to input items you may want to exclude from the list. I added "*Advanced Micro Devices*" as one item in my TXT file. The exclusions.txt file should reside in the same directory as the script.

The script has been tested when a system is connected to the domain (Final Image) and when it belongs to a workgroup (Reference Image). It works in both instances.

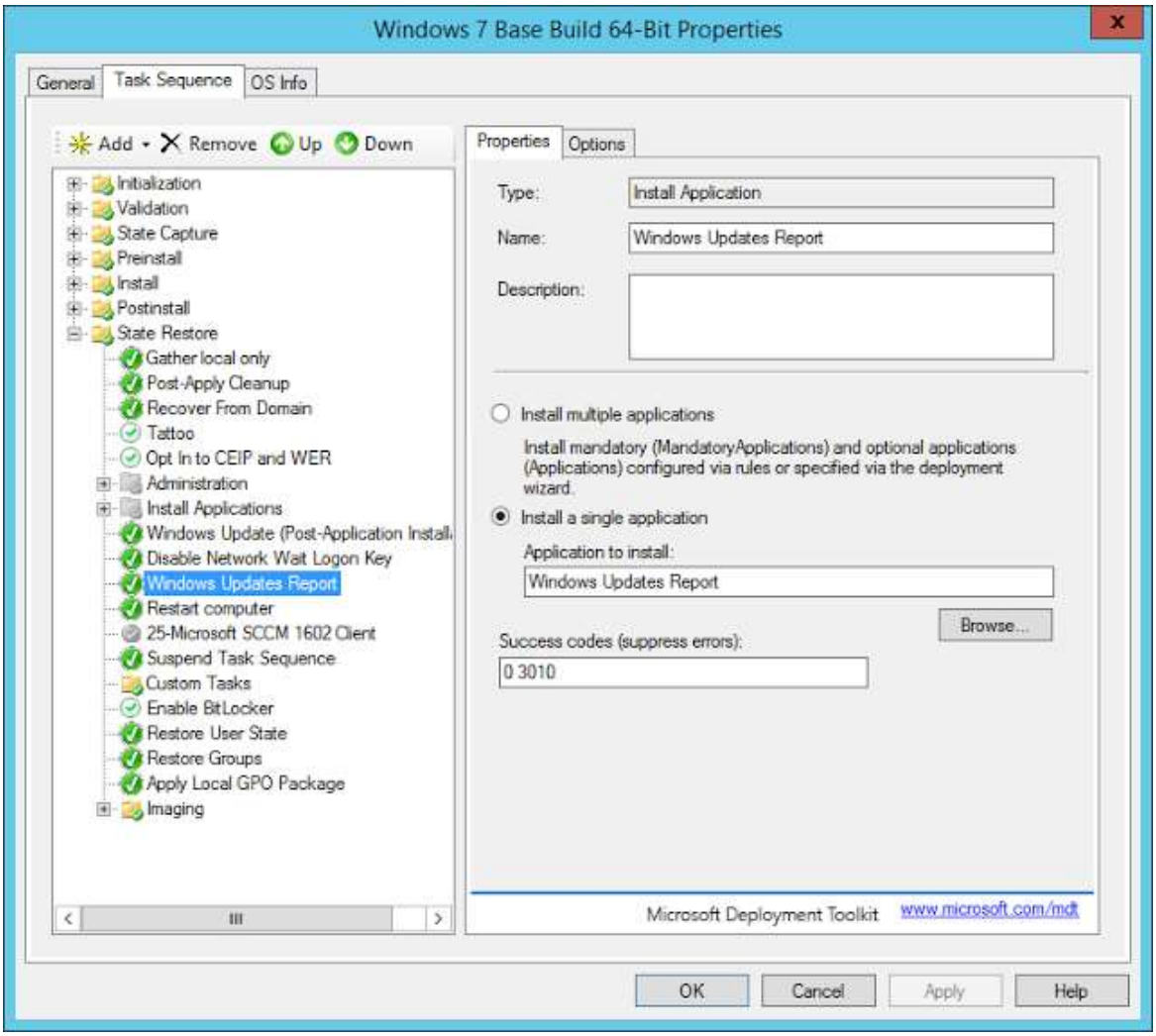
I have pre-populated all parameters, except From, To, and SMTPServer. Those were left blank since you would likely want to populate them at the command line.

Here is an example:

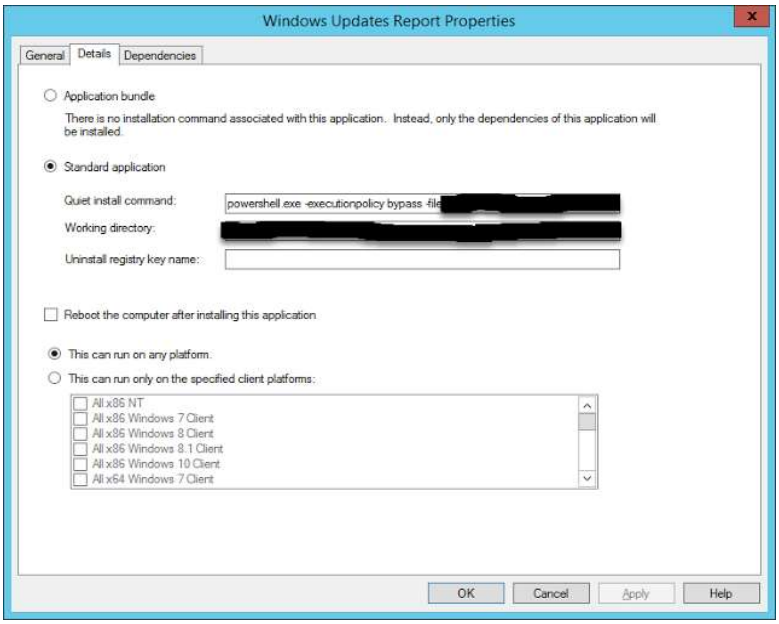
```
powershell.exe -file WindowsUpdatesReport.ps1 -email -From IT@Testcompany.com -
To mickpletcher@testcompany.com -SMTPServer smtp.testcompany.com
```

I have pre-populated the -OutputFile, -ExclusionsFile, -Subject, and -Body. You can go into the script and change those or decide to override them by defining them at the command line. You could also populate the -From, -To, and -SMTPServer if you like.

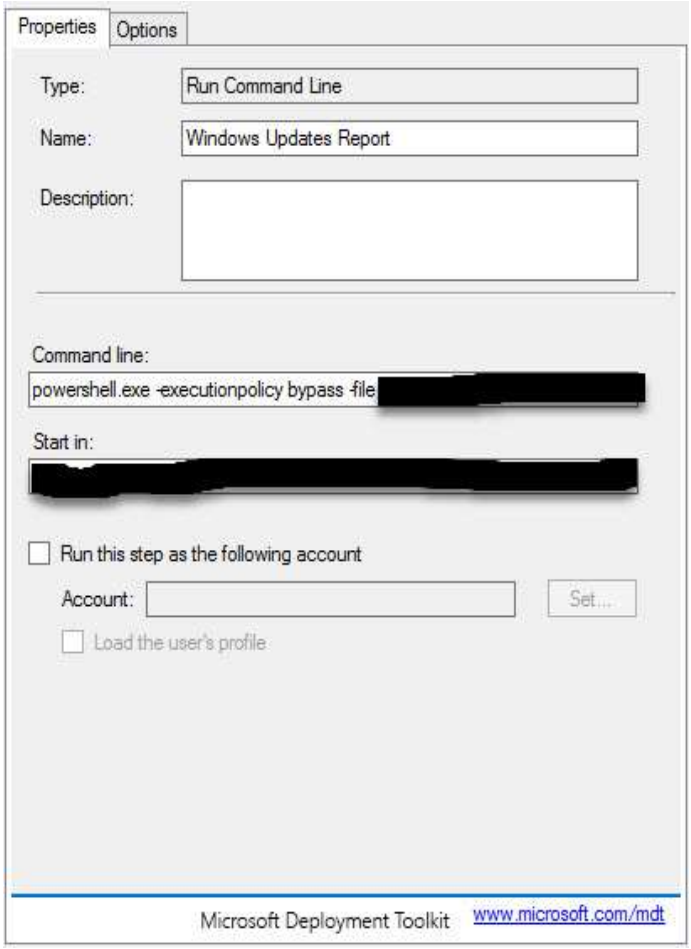
Here is a screenshot of how it is setup in the MDT task sequence the first time. This did not work.



And this is a filtered screenshot of how it is setup under as an application install:



I tried one more way to execute it and it finally worked as shown below:



The command line I used is: powershell.exe -executionpolicy bypass -file <UNC path>\WindowsUpdatesReport.ps1 -Email -From <sender's email address> -To <recipient's email address> -SMTPServer <SMTP server address>

The start in contains the <UNC path> where the script resides.

You can download the file from my GitHub location:
https://github.com/MicksITBlogs/PowerShell/blob/master/WindowsUpdatesReport.ps1

```
[CmdletBinding()]
 param
 (
      [ValidateNotNullOrEmpty()][string]$OutputFile =
'WindowsUpdatesReport.csv',
      [ValidateNotNullOrEmpty()][string]$ExclusionsFile = 'Exclusions.txt',
      [switch]$Email,
      [string]$From,
      [string]$To,
      [string]$SMTPServer,
      [string]$Subject = 'Windows Updates Build Report',
      [string]$Body = "List of windows updates installed during the build
process"
 )

 function Get-RelativePath {

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

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

 function Remove-OutputFile {

      [CmdletBinding()]
      param ()

      #Get the path this script is executing from
      $RelativePath = Get-RelativePath
```

```
        #Define location of the output file
        $File = $RelativePath + $OutputFile
        If ((Test-Path -Path $File) -eq $true) {
            Remove-Item -Path $File -Force
        }
}

function Get-Updates {

        [CmdletBinding()][OutputType([array])]
        param ()

        $UpdateArray = @()
        #Get the path this script is executing from
        $RelativePath = Get-RelativePath
        #File containing a list of exclusions
        $ExclusionsFile = $RelativePath + $ExclusionsFile
        #Get list of exclusions from exclusions file
        $Exclusions = Get-Content -Path $ExclusionsFile
        #Locate the ZTIWindowsUpdate.log file
        $FileName = Get-ChildItem -Path $env:HOMEDRIVE"\minint" -filter
ztiwindowsupdate.log -recurse
        #Get list of all installed updates except for Windows Malicious Software
Removal Tool, Definition Update for Windows Defender, and Definition Update for
Microsoft Endpoint Protection
        $FileContent = Get-Content -Path $FileName.FullName | Where-Object { ($_ -
like "*INSTALL*") } | Where-Object { $_ -notlike "*Windows Defender*" } | Where-
Object { $_ -notlike "*Endpoint Protection*" } | Where-Object { $_ -notlike
"*Windows Malicious Software Removal Tool*" } | Where-Object { $_ -notlike
"*Dell*" } | Where-Object { $_ -notlike $Exclusions }
        #Filter out all unnecessary lines
        $Updates = (($FileContent -replace (" - ", "~")).split("~") | where-object
{ ($_ -notlike "*LOG*INSTALL*") -and ($_ -notlike "*ZTIWindowsUpdate*") -and ($_
-notlike "*-*-*-*-*") })
```

```
      foreach ($Update in $Updates) {
            #Create object
            $Object = New-Object -TypeName System.Management.Automation.PSObject
            #Add KB article number to object
            $Object | Add-Member -MemberType NoteProperty -Name KBArticle -Value
($Update.split("(")[1].split(")")[0].Trim()
            #Add description of KB article to object
            $Description = $Update.split("(")[0]
            $Description = $Description -replace (",", " ")
            $Object | Add-Member -MemberType NoteProperty -Name Description -
Value $Description
            #Add the object to the array
            $UpdateArray += $Object
      }
      If ($UpdateArray -ne $null) {
            $UpdateArray = $UpdateArray | Sort-Object -Property KBArticle
            #Define file to write the report to
            $OutputFile = $RelativePath + $OutputFile
            $UpdateArray | Export-Csv -Path $OutputFile -NoTypeInformation -
NoClobber
      }
      Return $UpdateArray
 }

 Clear-Host
 #Delete the old report file
 Remove-OutputFile
 #Get list of installed updates
 Get-Updates
 If ($Email.IsPresent) {
      $RelativePath = Get-RelativePath
      $Attachment = $RelativePath + $OutputFile
```

```
      #Email Updates

      Send-MailMessage -From $From -To $To -Subject $Subject -Body $Body -
SmtpServer $SMTPServer -Attachments $Attachment

 }
```

Chapter 30

Report on Mapped Drives to understand Cryptolocker Vulnerabilities with SCCM and PowerShell

By: Mick Pletcher – MVP

Recently, we wanted to start keeping track of users with mapped drives due to cryptolocker vulnerabilities. There are a few applications we have that require mapped drives, so we have certain users with them.

This script will scan all user profiles on a machine and report users with mapped drives. This is done by parsing through the HKU registries. It has been written so that you can either have the script write the report to a text file if you do not have SCCM and/or it can write it to WMI so that SCCM can read the results. I have also included a UNCPathExclusionsFile parameter that allows you to create a text file that resides in the same directory as the script. It contains a list of UNC paths that you do not want the script to report. I recommend pre-populating the values of the $TextFileLocation and $UNCPathExclusionsFile parameters within the script. That just leaves the $OutputFile and $SCCMReporting left to specify at the command line.

If you are wanting this to write the results to SCCM, here is what you need to do. First, SCCM needs to know what to look for in order to report on it. This script will use WMI to report that data to SCCM. The first thing is to execute the script locally on any PC. Run it using the following command line: powershell.exe -file MappedDriveReport.ps1 -SCCMReporting

That command line will execute the script to scan for mapped drives write the results to WMI and then initiate a hardware inventory. Because the new WMI entry has not been added to SCCM, it will not be reported yet. Now that you have executed the script on the local machine, do the following:

1. Go into SCCM--->Administration Tab--->Client Settings---> Default Client Settings--->Hardware Inventory--->Set Classes.
2. Click Add--->Connect.
3. Enter the computer name of the system you ran the script on, check recursive, check Credentials required (Computer is not local)---> <domain>\<username> in the username field, and finally the password for the associated username.
4. Click Connect
5. Click on the Class Name tab to sort by class name
6. Scroll down to find MappedDrives and check the box
7. Click OK

You have now added the WMI class to SCCM for it to grab the data from the PCs and report it back to SCCM.

To get the systems to report the data back to SCCM, you will need to setup a package, not an application, in SCCM to deploy out to the systems. I have the package setup to re-run once a week at 12:00 pm on Wednesdays so that I can get the most users to report back. More users are online at that time here than any of the other days.

If you read the .Example in the documentation portion of the script, you will see two examples on how to execute the script.

I have also included a hardware inventory within the script so the data will be reported back to SCCM right after the script is executed.

In order to view the data in SCCM, you can do the following using the Resource Explorer:

1. Right-click on a machine in the Assets and Compliance--->Devices
2. Click Start--->Resource Explorer
3. Click the plus beside Hardware
4. If a system had mapped drives, then there will be a mapped drives field, otherwise it does not exist.

You can also use the queries to report systems with mapped drives. Here is the query I use:

```
select distinct SMS_G_System_MAPPEDDRIVES.user,
SMS_G_System_MAPPEDDRIVES.Letter, SMS_G_System_MAPPEDDRIVES.Path
from  SMS_R_System inner join SMS_G_System_MAPPEDDRIVES on
```

```
SMS_G_System_MAPPEDDRIVES.ResourceID = SMS_R_System.ResourceId
order by SMS_G_System_MAPPEDDRIVES.user
```

If you do not have SCCM and need a report, you can use the -OutputFile to have it write the results to a text file at the specified location defined in the $TextFileLocation parameter.

```
  [CmdletBinding()]
 param
 (
      [switch]
      $OutputFile,
      [string]
      $TextFileLocation =
'\\drfs1\DesktopApplications\ProductionApplications\Waller\MappedDrivesReport\Re
ports',
      [string]
      $UNCPathExclusionsFile =
"\\drfs1\DesktopApplications\ProductionApplications\Waller\MappedDrivesReport\UN
CPathExclusions.txt",
      [switch]
      $SCCMReporting
 )

 function Get-CurrentDate {

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

      $CurrentDate = Get-Date
      $CurrentDate = $CurrentDate.ToShortDateString()
      $CurrentDate = $CurrentDate -replace "/", "-"
      If ($CurrentDate[2] -ne "-") {
```

```
            $CurrentDate = $CurrentDate.Insert(0, "0")
        }
        If ($CurrentDate[5] -ne "-") {
            $CurrentDate = $CurrentDate.Insert(3, "0")
        }
        Return $CurrentDate
    }

    function Get-MappedDrives {
        [CmdletBinding()][OutputType([array])]

        #Get UNC Exclusions from UNCPathExclusions.txt file
        $UNCExclusions = Get-Content $UNCPathExclusionsFile -Force
        #Get HKEY_Users Registry Keys
        [array]$UserSIDS = (Get-ChildItem -Path REGISTRY::HKEY_Users | where-
Object { ($_ -notlike "*Classes*") -and ($_ -like "*S-1-5-21*") }).Name
        #Get Profiles from HKLM
        [array]$ProfileList = (Get-ChildItem -Path
REGISTRY::"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\ProfileList" | Where-Object { $_ -like "*S-1-5-21*" }).Name
        $UserMappedDrives = @()
        #Iterate through each HKEY_USERS profile
        foreach ($UserSID in $UserSIDS) {
            #GET SID only
            [string]$UserSID = $UserSID.Split("\")[1].Trim()
            #Find the userprofile that matches the HKEY_USERS
            [string]$UserPROFILE = $ProfileList | Where-Object { $_ -like "*" +
$UserSID + "*" }
            #Get the username associated with the SID
            $Username = ((Get-ItemProperty -Path
REGISTRY::$UserPROFILE).ProfileImagePath).Split("\")[2].trim()
            #Define registry path to mapped drives
```

```
            [string]$MappedDrives = "HKEY_USERS\" + $UserSID + "\Network"
            #Get list of mapped drives
            [array]$MappedDrives = (Get-ChildItem REGISTRY::$MappedDrives |
Select-Object name).name
            foreach ($MappedDrive in $MappedDrives) {
                $DriveLetter = (Get-ItemProperty -Path REGISTRY::$MappedDrive |
select PSChildName).PSChildName
                $DrivePath = (Get-ItemProperty -Path REGISTRY::$MappedDrive |
select RemotePath).RemotePath
                If ($DrivePath -notin $UNCExclusions) {
                    $Drives = New-Object System.Management.Automation.PSObject
                    $Drives | Add-Member -MemberType NoteProperty -Name
ComputerName -Value $env:COMPUTERNAME
                    $Drives | Add-Member -MemberType NoteProperty -Name
Username -Value $Username
                    $Drives | Add-Member -MemberType NoteProperty -Name
DriveLetter -Value $DriveLetter
                    $Drives | Add-Member -MemberType NoteProperty -Name
DrivePath -Value $DrivePath
                    $UserMappedDrives += $Drives
                }
            }
        }
        Return $UserMappedDrives
    }

    function Get-RelativePath {

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

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

```
}

function Invoke-SCCMHardwareInventory {

    [CmdletBinding()]
    param ()

    $ComputerName = $env:COMPUTERNAME
    $SMSCli = [wmiclass] "\\$ComputerName\root\ccm:SMS_Client"
    $SMSCli.TriggerSchedule("{00000000-0000-0000-0000-000000000001}") | Out-
Null
}

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

    $WMITest = Get-WmiObject $Class -ErrorAction SilentlyContinue
    If ($WMITest -ne $null) {
        $Output = "Deleting " + $WMITest.__CLASS[0] + " 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("ComputerName",
[System.Management.CimType]::String, $false)
        $newClass.Properties["ComputerName"].Qualifiers.Add("key", $true)
        $newClass.Properties["ComputerName"].Qualifiers.Add("read", $true)
        $newClass.Properties.Add("DriveLetter",
[System.Management.CimType]::String, $false)
        $newClass.Properties["DriveLetter"].Qualifiers.Add("key", $false)
        $newClass.Properties["DriveLetter"].Qualifiers.Add("read", $true)
        $newClass.Properties.Add("DrivePath",
[System.Management.CimType]::String, $false)
        $newClass.Properties["DrivePath"].Qualifiers.Add("key", $false)
        $newClass.Properties["DrivePath"].Qualifiers.Add("read", $true)
        $newClass.Properties.Add("Username", [System.Management.CimType]::String,
$false)
        $newClass.Properties["Username"].Qualifiers.Add("key", $false)
        $newClass.Properties["Username"].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 {

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

        foreach ($MappedDrive in $MappedDrives) {
            Set-WmiInstance -Class $Class -Arguments @{ ComputerName =
$MappedDrive.ComputerName; DriveLetter = $MappedDrive.DriveLetter; DrivePath =
$MappedDrive.DrivePath; Username = $MappedDrive.Username } | Out-Null
        }
    }

    function Start-ConfigurationManagerClientScan {

        [CmdletBinding()]
        param
        (
            [ValidateSet('00000000-0000-0000-0000-000000000121', '00000000-0000-
0000-0000-000000000003', '00000000-0000-0000-0000-000000000010', '00000000-0000-
0000-0000-000000000001', '00000000-0000-0000-0000-000000000021', '00000000-0000-
0000-0000-000000000022', '00000000-0000-0000-0000-000000000002', '00000000-0000-
0000-0000-000000000031', '00000000-0000-0000-0000-000000000108', '00000000-0000-
0000-0000-000000000113', '00000000-0000-0000-0000-000000000111', '00000000-0000-
```

```
0000-0000-000000000026', '00000000-0000-0000-0000-000000000027', '00000000-0000-
0000-0000-000000000032')]$ScheduleID
        )

        $WMIPath = "\\" + $env:COMPUTERNAME + "\root\ccm:SMS_Client"
        $SMSwmi = [wmiclass]$WMIPath
        $Action = [char]123 + $ScheduleID + [char]125
        [Void]$SMSwmi.TriggerSchedule($Action)
  }

  cls
  #Get list of mapped drives for each user
  $UserMappedDrives = Get-MappedDrives
  #Write output to a text file if -OutputFile is specified
  If ($OutputFile.IsPresent) {
        If (($TextFileLocation -ne $null) -and ($TextFileLocation -ne "")) {
              #Add backslash (\) to the end of the TextFileLocation if it is not
present
              If ($TextFileLocation[$TextFileLocation.Length - 1] -ne "\") {
                    $TextFileLocation += "\"
              }
              #Write list of mapped drives to the specified text file.
              [string]$OutputFile = [string]$TextFileLocation + $env:COMPUTERNAME
+ ".txt"
        } else {
              #Get the relative path this script was executed from
              $RelativePath = Get-RelativePath
              $OutputFile = $RelativePath + $env:COMPUTERNAME + ".txt"
        }
        If ((Test-Path $OutputFile) -eq $true) {
              Remove-Item $OutputFile -Force
```

```
        }
        If (($UserMappedDrives -ne $null) -and ($UserMappedDrives -ne "")) {
             $UserMappedDrives | Format-Table -AutoSize | Out-File $OutputFile -
Width 255
        }
   }
   If ($SCCMReporting.IsPresent) {
        #Create the new WMI class to write the output data to
        New-WMIClass -Class "Mapped_Drives"
        #Write the output data as an instance to the WMI class
        If ($UserMappedDrives -ne $null) {
             New-WMIInstance -MappedDrives $UserMappedDrives -Class
"Mapped_Drives"
        }
        #Invoke a hardware inventory to send the data to SCCM
        Invoke-SCCMHardwareInventory
   }
   #Display list of mapped drives for each user
   $UserMappedDrives | Format-Table
```

Chapter 31

Set Windows Features and Verify with PowerShell

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.

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

function Confirm-Feature {

    [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 {

        [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 {

        [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 {

        [CmdletBinding()]
        param ()

        $RelativePath = (split-path $SCRIPT:MyInvocation.MyCommand.Path -parent)
+ '\'
        $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
}
```

Chapter 32

Uninstall an Application by Name with PowerShell

By: Mick Pletcher – MVP

Here is a function that will uninstall an MSI installed application by the name of the app. You do not need to input the entire name either. For instance, say you are uninstalling all previous versions of Adobe Reader. Adobe Reader is always labeled Adobe Reader X, Adobe Reader XI, and so forth. This script allows you to do this without having to find out every version that is installed throughout a network and then enter an uninstaller line for each version. You just need to enter Adobe Reader as the application name and the desired switches. It will then search the name fields in the 32 and 64 bit uninstall registry keys to find the associated GUID. Finally, it will execute an msiexec.exe /x {GUID} to uninstall that version.

```
  [CmdletBinding()]
param ()

function Uninstall-MSIByName {

     [CmdletBinding()]
     param
     (
          [ValidateNotNullOrEmpty()][String]$ApplicationName,
          [ValidateNotNullOrEmpty()][String]$Switches
     )
```

```
      #MSIEXEC.EXE

      $Executable = $Env:windir + "\system32\msiexec.exe"

      #Get list of all Add/Remove Programs for 32-Bit and 64-Bit

      $Uninstall = Get-ChildItem
HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall -Recurse -ErrorAction
SilentlyContinue

      If (((Get-WmiObject -Class Win32_OperatingSystem | Select-Object
OSArchitecture).OSArchitecture) -eq "64-Bit") {

           $Uninstall += Get-ChildItem
HKLM:\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall -Recurse -
ErrorAction SilentlyContinue

      }

      #Find the registry containing the application name specified in
$ApplicationName

      $Key = $uninstall | foreach-object { Get-ItemProperty REGISTRY::$_ } |
where-object { $_.DisplayName -like "*$ApplicationName*" }

      If ($Key -ne $null) {

           Write-Host "Uninstall"$Key.DisplayName"....." -NoNewline

           #Define msiexec.exe parameters to use with the uninstall

           $Parameters = "/x " + $Key.PSChildName + [char]32 + $Switches

           #Execute the uninstall of the MSI

           $ErrCode = (Start-Process -FilePath $Executable -ArgumentList
$Parameters -Wait -Passthru).ExitCode

           #Return the success/failure to the display

           If (($ErrCode -eq 0) -or ($ErrCode -eq 3010) -or ($ErrCode -eq 1605))
{

                Write-Host "Success" -ForegroundColor Yellow

           } else {

                Write-Host "Failed with error code "$ErrCode -ForegroundColor
Red

           }

      }

 }


 Clear-Host
```

```
Uninstall-MSIByName -ApplicationName "Cisco Jabber" -Switches "/qb- /norestart"
```

Chapter 33

Azure Automatic Account Creation and Adding Modules using PowerShell

By: Will Anderson – MVP

Microsoft's Operations Management Suite provides some exceptional tools for monitoring and maintaining your environments in both the cloud and in your datacenter. One of it's best features, however, is its ability to leverage the tools that you've already developed to perform tasks and remediate issues using PowerShell, Azure Automation Runbooks, and OMS Alert triggers. In this series, we'll be discussing how you can configure these tools to take care of problems in your own environment. Today, we'll be talking about how you can take your own PowerShell Modules and upload them to Azure Automation.

Creating the Azure Automation Account

In order to create the Azure Automation Account, you'll need to have create the automation account object in the target resource group, and the ability to create an AzureRunAs account in AzureAD. It's also important to be mindful that not every Azure region has the Microsoft.Automation resource provider registered to it, so you'll want the resource group to exist in the appropriate locale. You can check this with the Get-AzureRmResourceProvider cmdlet:

```
Get-AzureRmResourceProvider -ProviderNamespace 'Microsoft.Automation'
```

```
PS C:\windows\system32> Get-AzureRmResourceProvider -ProviderNamespace 'Microsoft.Automation'


ProviderNamespace : Microsoft.Automation
RegistrationState : Registered
ResourceTypes     : {automationAccounts}
Locations         : {Japan East, East US 2, West Europe, Southeast Asia...}

ProviderNamespace : Microsoft.Automation
RegistrationState : Registered
ResourceTypes     : {automationAccounts/runbooks}
Locations         : {Japan East, East US 2, West Europe, Southeast Asia...}

ProviderNamespace : Microsoft.Automation
RegistrationState : Registered
ResourceTypes     : {automationAccounts/webhooks}
Locations         : {Japan East, East US 2, West Europe, Southeast Asia...}

ProviderNamespace : Microsoft.Automation
RegistrationState : Registered
ResourceTypes     : {operations}
Locations         : {South Central US}

ProviderNamespace : Microsoft.Automation
RegistrationState : Registered
ResourceTypes     : {automationAccounts/softwareUpdateConfigurations}
Locations         : {Japan East, East US 2, West Europe, Southeast Asia...}
```

For our purposes, we'll be deploying a resource group to East US 2. Once the resource group has been created, we'll use New-AzureRmAutomationAccount

```
$BaseName = 'testautoacct'

$Location = 'eastus2'

$ResGrp = New-AzureRmResourceGroup -Name $BaseName -Location $Location -Verbose

$AutoAcct = New-AzureRmAutomationAccount -ResourceGroupName
$ResGrp.ResourceGroupName -Name ($BaseName + $Location) -Location
$ResGrp.Location
```

It's good to note that while -Verbose is available for New-AzureRmAutomationAccount, it will not return any verbose output.

```
PS C:\windows\system32> $ResGrp = New-AzureRmResourceGroup -Name $BaseName -Location $Location -Verbose
VERBOSE: Performing the operation "Replacing resource group ..." on target "".
VERBOSE: 9:30:21 AM - Created resource group 'testautoacct' in location 'eastus2'

PS C:\windows\system32> New-AzureRmAutomationAccount -ResourceGroupName $ResGrp.ResourceGroupName -Name ($BaseName + $Location) -Location $ResGrp.Location -Verbose


SubscriptionId        : f2007bbf-f802-4a47-9336-cf7c6b89b378
ResourceGroupName     : testautoacct
AutomationAccountName : testautoaccteastus2
Location              : eastus2
State                 : Ok
Plan                  : Free
CreationTime          : 7/12/2017 9:31:25 AM -04:00
LastModifiedTime      : 7/12/2017 9:32:16 AM -04:00
LastModifiedBy        : live.com#will.anderson@gamerliving.net
Tags                  : {}
```

Creating a Blob Container in AzureRM

Now that we have our automation account created, we can begin uploading our modules to be available for Azure Automation to use. In order to do so, we'll need to create a blob store that we can upload our modules to so that the Azure Automation Account can import them; unlike in the Azure UI, you cannot currently upload your modules directly from your local machine, so you'll need to supply a URI for Azure Automation to access.

Another 'gotcha' is that there is no AzureRm cmdlet for creating a blob container, or for uploading content to that container, so you'll need to do so using the Azure storage commands and passing the Storage Context Key from AzureRM to Azure. Here is how you can create the storage account, get the storage account key, create a context, and pass it to Azure:

```
$Stor = New-AzureRmStorageAccount -ResourceGroupName $ResGrp.ResourceGroupName -
Name modulestor -SkuName Standard_LRS -Location $ResGrp.Location -Kind
BlobStorage -AccessTier Hot


Add-AzureAccount

$Subscription = ((Get-AzureSubscription).where({$PSItem.SubscriptionName -eq
'LastWordInNerd'}))

Select-AzureSubscription -SubscriptionName $Subscription.SubscriptionName -
Current
```

```
$StorKey = (Get-AzureRmStorageAccountKey -ResourceGroupName
$Stor.ResourceGroupName -Name $Stor.StorageAccountName).where({$PSItem.KeyName -
eq 'key1'})

$StorContext = New-AzureStorageContext -StorageAccountName
$Stor.StorageAccountName -StorageAccountKey $StorKey.Value
```

Once we've run our storage commands, you'll have captured the storage context object like so:

```
PS C:\windows\system32> $StorKey = (Get-AzureRmStorageAccountKey -ResourceGroupName $Stor.ResourceGroupName -Name $St
$StorContext = New-AzureStorageContext -StorageAccountName $Stor.StorageAccountName -StorageAccountKey $StorKey.Valu

PS C:\windows\system32> $StorContext


StorageAccountName : modulestor
BlobEndPoint       : https://modulestor.blob.core.windows.net/
TableEndPoint      : https://modulestor.table.core.windows.net/
QueueEndPoint      : https://modulestor.queue.core.windows.net/
FileEndPoint       : https://modulestor.file.core.windows.net/
Context            : Microsoft.WindowsAzure.Commands.Storage.AzureStorageContext
Name               :
StorageAccount     : BlobEndpoint=https://modulestor.blob.core.windows.net/;QueueEndpoint=https://modulestor.queue.co
                     https://modulestor.table.core.windows.net/;FileEndpoint=https://modulestor.file.core.windows.net
                     ntKey=[key hidden]
EndPointSuffix     : core.windows.net/
ConnectionString   : BlobEndpoint=https://modulestor.blob.core.windows.net/;QueueEndpoint=https://modulestor.queue.co
                     https://modulestor.table.core.windows.net/;FileEndpoint=https://modulestor.file.core.windows.net
                     ntKey=xSxuFilWT5W9D3W1ga4K3Rx9leOFGKYHuwl4iTPgi6d1g2Bp/NaHgr9rMRrTQ2y3uoYj1ngReDk3nKhlYKCMxA==
ExtendedProperties : {}
```

Now that we've got access to our AzureRm storage account in Azure, we can now create our blob container:

```
$Container = New-AzureStorageContainer -Name 'modules' -Permission Blob -Context
$StorContext -Permission Blob
```

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


CloudBlobContainer : Microsoft.WindowsAzure.Storage.Blob.CloudBlobContainer
Permission         : Microsoft.WindowsAzure.Storage.Blob.BlobContainerPermissions
PublicAccess       : Blob
LastModified       : 7/12/2017 5:12:38 PM +00:00
ContinuationToken  :
Context            : Microsoft.WindowsAzure.Commands.Storage.AzureStorageContext
Name               : modules
```

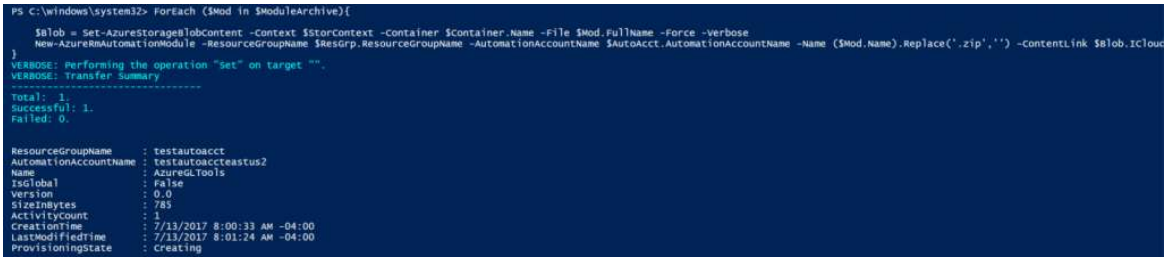
NOTE - I have my container permission set to Blob, which makes this directory publicly available. At some time in the near future, I'll walk you through how you can use SAS Tokens to access secure blobs at runtime. Just be mindful of this if you use this code in production.

Upload a Blob Container

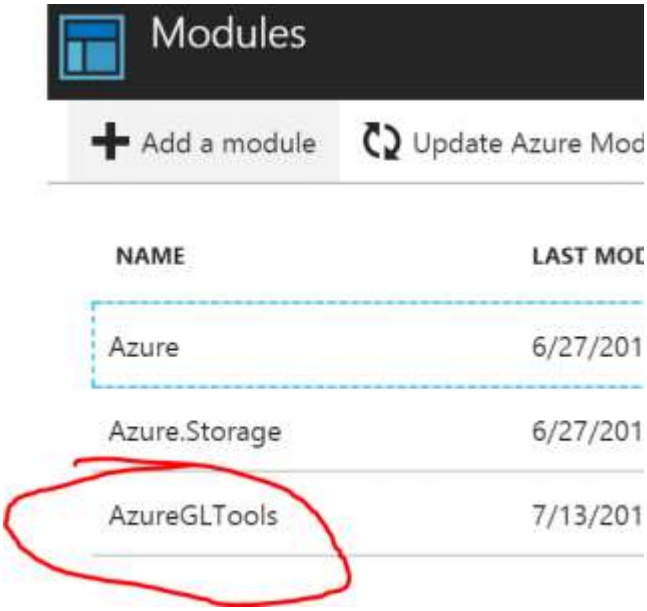
Now we can finally upload our modules to the blob store, and register them in Azure Automation! What we're going to do here is take our custom module, compress it into a .zip file, and then use the Set-AzureStorageBlobContent cmdlet to ship it up to our blob store. Once the content is shipped, we use the $Blob.ICloudBlob.Uri.AbsoluteUri to feed the New-AzureRmAutomationModule the URI required for the ContentLink parameter.

```
$ModuleLoc = 'C:\Scripts\Presentations\OMSAutomation\Modules\'
$Modules = Get-ChildItem -Directory -Path $ModuleLoc

    ForEach ($Mod in $Modules){

        Compress-Archive -Path $Mod.PSPath -DestinationPath ($ModuleLoc + '\' +
$Mod.Name + '.zip') -Force
```

```
    }

$ModuleArchive = Get-ChildItem -Path $ModuleLoc -Filter "*.zip"

ForEach ($Mod in $ModuleArchive){

    $Blob = Set-AzureStorageBlobContent -Context $StorContext -Container
$Container.Name -File $Mod.FullName -Force -Verbose

    New-AzureRmAutomationModule -ResourceGroupName $ResGrp.ResourceGroupName -
AutomationAccountName $AutoAcct.AutomationAccountName -Name
($Mod.Name).Replace('.zip','') -ContentLink $Blob.ICloudBlob.Uri.AbsoluteUri

}
```



Now that we've done all that, we can validate that we have our module in Azure Automation through the UI:



Now that we've uploaded our modules into Azure Automation, we can start using them to perform tasks in Azure.

Chapter 34

Configuring Azure Automation Runbooks and Understanding Webhook Data using PowerShell

By: Will Anderson – MVP

So, last time we learned how to upload our custom modules into Azure Automation so we can start using them in Azure Automation Runbooks. This week we're going to take a look at configuring a runbook to see what kind of data we can ingest from OMS Webhook data, and how we can leverage that data to pass into our functions.

Creating the Runbook Script

So first off, let's talk about basic runbooks and running them against objects in Azure. As previously discussed, when your automation account is created, it creates with it an AzureRunAsAccount. This account is configured to act on behalf of the user that has access to the automation account and the runbooks in order to perform the runbook task. In order to leverage this account, you need to invoke it in the runbook itself. You can actually find an example of this snippet in the AzureAutomationTutorialScript runbook in your automation account.

```
$connectionName = "AzureRunAsConnection"
try
{
    # Get the connection "AzureRunAsConnection "
    $servicePrincipalConnection=Get-AutomationConnection -Name $connectionName

    "Logging in to Azure..."
    Add-AzureRmAccount `
        -ServicePrincipal `
        -TenantId $servicePrincipalConnection.TenantId `
        -ApplicationId $servicePrincipalConnection.ApplicationId `
        -CertificateThumbprint $servicePrincipalConnection.CertificateThumbprint
```

```
}
catch {
    if (!$servicePrincipalConnection)
    {
        $ErrorMessage = "Connection $connectionName not found."
        throw $ErrorMessage
    } else{
        Write-Error -Message $_.Exception
        throw $_.Exception
    }
}
```

So now that we've got our opening snippet, we'll add that into a new .ps1 script file in our preferred integrated scripting environment tool and get to work.

Now, in order to be able to ingest data from an OMS Alert, we need to be able to pass the data to our Azure Automation runbook. In order to do so, we only need to add a $WebHookData parameter to the runbook and specify the data type as object.

```
Param (

    [Parameters()][object]$WebHookData

)
```

Now, we need to convert that data from a JSON object into something readable in our output. Webhook data is presented with three primary datasets - WebhookName, RequestHeader, and RequestBody. WebhookName, obviously is the name of the incoming webhook. RequestHeader is a hash table containing all of the header data for the incoming requestion. And finally, RequestBody is the body of the incoming request. This is where the data we want to parse will reside. Specifically, it will reside under the SearchResults property of the RequestHeader dataset.

```
$WebhookData.WebhookName
$WebhookData.RequestHeader
$WebhookData.RequestBody
```

Let's configure our runbook to display the incoming data to examine what we have to play with.

```
$SearchResults = (ConvertFrom-Json $WebhookData.RequestBody).SearchResults.value

$SearchResults
```

Publish the Runbook

Now, we'll go ahead and save our script as a .ps1 file and upload it to our automation account with the Import-AzureRmAutomationRunbook cmdlet.

```
Import-AzureRmAutomationRunbook -Path
'C:\Scripts\Presentations\OMSAutomation\ExampleRunbookScript.ps1' -Name
WebhookNSGRule -Type PowerShell -ResourceGroupName $AutoAcct.ResourceGroupName -
AutomationAccountName $AutoAcct.AutomationAccountName -Published
```

And now we can see our return.

```
PS C:\windows\system32> Import-AzureRmAutomationRunbook -Path 'C:\Scripts\Presentatio


Location              : eastus2
Tags                  : {}
JobCount              : 0
RunbookType           : PowerShell
Parameters            : {}
LogVerbose            : False
LogProgress           : False
LastModifiedBy        : live.com#will.anderson@gamerliving.net
State                 : New
ResourceGroupName     : testautoacct
AutomationAccountName : testautoaccteastus2
Name                  : ExampleRunbook
CreationTime          : 7/17/2017 8:40:56 AM -04:00
LastModifiedTime      : 7/17/2017 8:40:56 AM -04:00
Description           :
```

And if we check through the UI, we can see a brand-new, shiny runbook sitting in our automation account! Now, we can configure a basic alert to monitor in OMS.

Create an Alert

For the purposes of this example, I've create a couple of virtual machines with network security group rules for HTTP:80 and RDP:3389 accepting connections from anywhere. I do not recommend doing this for a production virtual machine. /endDisclaimer

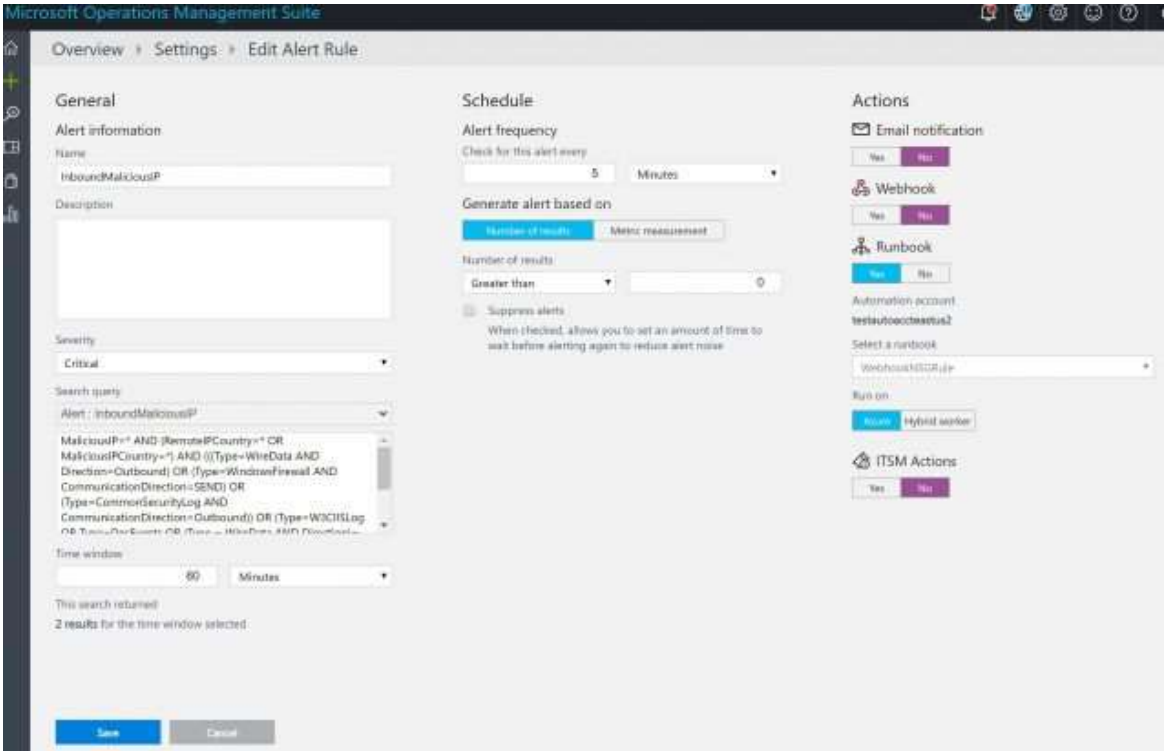
As you can well expect, these machines are throwing MaliciousIP traffic alerts in Operations Management Suite's console:

If we click on the MaliciousIP flag, it'll take us to the Log Search screen. This includes the query data that we can use for the alert. However, you'll want to clean up the query data a bit to generalize it. In this example, the query is specific to the country that is displayed in the given flag. But if we remove the country specific portion of the query, it'll allow us to cast a wider net and get data on potentially malicious traffic from any given country.

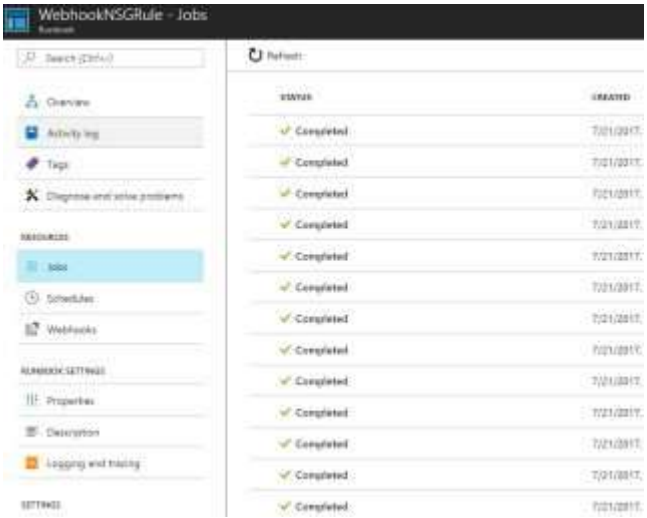
```
Canned Query:

MaliciousIP=* AND (RemoteIPCountry=* OR MaliciousIPCountry=*) AND
(((Type=WireData AND Direction=Outbound) OR (Type=WindowsFirewall
AND CommunicationDirection=SEND) OR (Type=CommonSecurityLog AND
CommunicationDirection=Outbound)) OR (Type=W3CIISLog OR
Type=DnsEvents OR (Type = WireData AND Direction!= Outbound) OR
(Type=WindowsFirewall AND CommunicationDirection!=SEND) OR (Type
```

```
= CommonSecurityLog AND CommunicationDirection!= Outbound)))
(RemoteIPCountry="People's Republic of China" OR
MaliciousIPCountry="People's Republic of China")



Modified Query:

MaliciousIP=* AND (RemoteIPCountry=* OR MaliciousIPCountry=*) AND
(((Type=WireData AND Direction=Outbound) OR (Type=WindowsFirewall
AND CommunicationDirection=SEND) OR (Type=CommonSecurityLog AND
CommunicationDirection=Outbound)) OR (Type=W3CIISLog OR
Type=DnsEvents OR (Type = WireData AND Direction!= Outbound) OR
(Type=WindowsFirewall AND CommunicationDirection!=SEND) OR (Type
= CommonSecurityLog AND CommunicationDirection!= Outbound)))
```

After testing our query to make sure it's valid, we can now hit the alert button and configure the alert. Here you'll need to give it an alert name, a schedule, and number of results before it triggers the alert. You'll also want to select the Runbook option under actions and select the test runbook we created. Then we hit save, and wait for our alert to trigger and the runbook to fire.



And as you can see, I didn't have to wait long:



Validating our Data

If we click on one of the completed instances, and navigate to the output blade, we can now see the data we're receiving from our triggered alert. This particular data shows that inbound traffic from Colombia is attempting an RDP connection to my virtual machine. With the inbound IP Address and target system name, we now have enough data to be able to create a full-blown auto-remediation solution.

```
Logging in to Azure...




Environments
Context
```

```
------------
-------

{[AzureCloud, AzureCloud], [AzureChinaCloud, AzureChinaCloud],
[AzureUSGovernment, AzureUSGovernment]} Microsoft.Azur...




Computer                    : server1

MG                          : 00000000-0000-0000-0000-000000000001

ManagementGroupName         : AOI-cb0eefe8-b88f-47ce-ae91-
dbc46df99751

SourceSystem                : OpsManager

TimeGenerated               : 2017-07-21T12:17:37.45Z

SessionStartTime            : 2017-07-21T12:16:52Z

SessionEndTime              : 2017-07-21T12:16:52Z

LocalIP                     : 10.119.192.10

LocalSubnet                 : 10.119.192.0/21

LocalMAC                    : 00-0d-3a-03-ea-a6

LocalPortNumber             : 3389

RemoteIP                    : 200.35.53.121

RemoteMAC                   : 12-34-56-78-9a-bc

RemotePortNumber            : 4935
```

```
SessionID                   :
10.119.192.10_3389_200.35.53.121_4935_2184_2017-07-
21T12:16:52.000Z

SequenceNumber              : 0

SessionState                : Listen

SentBytes                   : 20

ReceivedBytes               : 40

TotalBytes                  : 60

ProtocolName                : TCP

IPVersion                   : IPv4

SentPackets                 : 1

ReceivedPackets             : 2

Direction                   : Inbound

ApplicationProtocol         : RDP

ProcessID                   : 888

ProcessName                 : C:\Windows\System32\svchost.exe

ApplicationServiceName      : ms-wbt-server

LatencyMilliseconds         : 116

LatencySamplingTimeStamp    : 2017-07-21T12:16:52Z

LatencySamplingFailureRate : 0.0%

MaliciousIP                 : 200.35.53.121
```

```
IndicatorThreatType        : Botnet

Confidence                 : 75

Severity                   : 2

FirstReportedDateTime      : 2017-07-20T20:10:32Z

LastReportedDateTime       : 2017-07-21T11:25:11.0661909Z

IsActive                   : true

ReportReferenceLink        : https://interflowinternal.azure-
api.net/api/reports/download/generic/webbot.json

RemoteIPLongitude          : -75.88

RemoteIPLatitude           : 8.77

RemoteIPCountry            : Colombia

id                         : 149270bc-74fc-13d0-34a9-3fd665a457b2

Type                       : WireData

__metadata                 : @{Type=WireData; TimeGenerated=2017-
07-21T12:17:37.45Z}
```

It's a long road, and we're almost there! In the next chapter I'll take you through my process of modifying my module to directly ingest webhook data, and how we can take our OMS queries and deploy them to other Operations Management Suite solutions using PowerShell. See you then!

Chapter 35

Utilizing Webhook Data in Functions and Validate Results using PowerShell

By: Will Anderson – MVP

It's been a long road, but we're almost there! A couple of weeks ago we looked at how we can create an Azure Automation Account and add our own custom modules to the solution to be used in Azure Automation. Last week, we took a deeper dive into configuring a runbook to take in webhook data from an alert using Microsoft's Operations Management Suite. Then we looked into the data itself to see how we can leverage it against our runbook to fix problems for us on the fly.

In this chapter, we're going to modify an existing function to use that webhook data directly.

Building on Webhook Data

We could actually build our logic directly into the runbook to parse the webhook data and then pass the formatted information to our function that we've made available in Azure. But I prefer to keep my runbooks as simple as possible and do the heavy lifting in my function. This makes the runbook look a little bit cleaner, and allows me to minimize my code management a little more. Also, Azure Automation Runbooks, as of this writing, don't play nicely with parameter sets in them, so I might as well pass my data along to a command that does.

Originally, I had built a one-liner that allowed me to create an NSG rule on the fly to block and incoming traffic from a specific IPAddress. It was a fairly simple command. But today, we're

going to make it a little more robust, and give it the ability to use webhook data. Here's my original code:

```
Function Set-AzureRmNSGMaliciousRule {

    [cmdletbinding()]

    Param(

        [Parameter(Mandatory=$true)][string]$ComputerName,
        [Parameter(Mandatory=$true)][string]$IPAddress



    )



    $ResGroup = (Get-AzureRmResource).where({$PSItem.Name -eq $Sys})
    $VM = Get-AzureRmVM -ResourceGroupName $ResGroup.ResourceGroupName -Name
$Sys


    $VmNsg = (Get-AzureRmNetworkSecurityGroup -ResourceGroupName
$VM.ResourceGroupName).where({$PSItem.NetworkInterfaces.Id -eq
$VM.NetworkProfile.NetworkInterfaces.Id})


    $Priority = ($VmNsg.SecurityRules) | Where-Object -Property Priority -LT 200
| Select-Object -Last 1


    If ($Priority -eq $null){


        $Pri = 100


    }
```

```
    Else {

        $Pri = ($Priority + 1)

    }



    $Name = ('BlockedIP_' + $IPAddress)



    $NSGArgs = @{

        Name = $Name
        Description = ('Malicious traffic from ' + $IPAddress)
        Protocol = '*'
        SourcePortRange = '*'
        DestinationPortRange = '*'
        SourceAddressPrefix = $IPAddress
        DestinationAddressPrefix = '*'
        Access = 'Deny'
        Direction = 'Inbound'
        Priority = $Pri

    }


    $VmNsg | Add-AzureRmNetworkSecurityRuleConfig @NSGArgs | Set-
AzureRmNetworkSecurityGroup
}
```

I want to keep my mandatory parameters for my original one-liner solution in-case I need to do something tactically. So we'll go ahead and split the parameters for on-prem vs. webhook into different parameter sets. As webhook data is formatted as a JSON object, we'll need to specify the data type for the WebhookData parameter as object.

```
Param(

    [Parameter(ParameterSetName='ConsoleInput')][string]$ComputerName,
    [Parameter(ParameterSetName='ConsoleInput')][string]$MaliciousIP,
    [Parameter(ParameterSetName="WebhookInput")][object]$WebhookData


)
```

Now, we're going to add some logic to parse out the data that we're looking to use:

```
 If($PSCmdlet.ParameterSetName -eq 'WebhookInput'){

    $SearchResults = (ConvertFrom-Json
$WebhookData.RequestBody).SearchResults.value

    Write-Output ("Target computer is " + $SearchResults.Computer)
```

```
        Write-Output ("Malicious IP is " + $SearchResults.RemoteIP)

        $ComputerName = (($SearchResults.Computer).split(' ') | Select-Object -
First 1)
        $MaliciousIP = (($SearchResults.RemoteIP).split(' ') | Select-Object -
First 1)

    }



    If ($ComputerName -like "*.*"){

        $Sys = $ComputerName.Split('.') | Select-Object -First 1

    }
    Else {
        $Sys = $ComputerName
    }
```

You'll notice that I'm doing some string formatting with our data here. Webhook data can concatenate multiple alerts together and separate the array by using spaces, so we're splitting that up and grabbing the first entry for each input we need. The additional splitting on the ComputerName is to accomodate for systems that are domain joined, as Azure isn't necessarily aware of a system's FQDN. Mind you, this is a rough example, and continuously growing; So as my use cases evolve, so will my code.

Now that we have our data formatted, we can update our module and upload it to our Azure Automation Account using the same process outlined in Part I, but with the -Force parameter added so we can overwrite the existing instance.

```
    Param(

    [Parameter(Mandatory=$true)]
    [object]$WebhookData

)


$connectionName = "AzureRunAsConnection"
try
{
    # Get the connection "AzureRunAsConnection "
    $servicePrincipalConnection=Get-AutomationConnection -Name $connectionName

    "Logging in to Azure..."
    Add-AzureRmAccount `
        -ServicePrincipal `
        -TenantId $servicePrincipalConnection.TenantId `
        -ApplicationId $servicePrincipalConnection.ApplicationId `
        -CertificateThumbprint $servicePrincipalConnection.CertificateThumbprint
}
catch {
    if (!$servicePrincipalConnection)
    {
        $ErrorMessage = "Connection $connectionName not found."
        throw $ErrorMessage
    } else{
        Write-Error -Message $_.Exception
        throw $_.Exception
    }
```

```
}

Set-AzureRmNSGMaliciousRule -WebHookData $WebhookData
```

Now, in a few minutes, our runbook should trigger and we can monitor the result.

```
$Job = (Get-AzureRmAutomationJob -RunbookName WebhookNSGRule -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName
$AutoAcct.AutomationAccountName)
$Job[0] | Select-Object -Property *
```

```
ResourceGroupName       : mms-eus

AutomationAccountName   : testautoaccteastus2

JobId                   : 339601cd-14e9-4002-8fcd-7d2008726445

CreationTime            : 7/24/2017 10:11:43 AM -04:00

Status                  : Completed

StatusDetails           :

StartTime               : 7/24/2017 10:12:21 AM -04:00

EndTime                 : 7/24/2017 10:13:31 AM -04:00

Exception               :

LastModifiedTime        : 7/24/2017 10:13:31 AM -04:00

LastStatusModifiedTime  : 1/1/0001 12:00:00 AM +00:00

JobParameters           : {}
```

```
RunbookName             : WebhookNSGRule

HybridWorker            :

StartedBy               :
```

We can start digging into the outputs of the runbook after completion to gather a little more data.

```
$Job = (Get-AzureRmAutomationJob -RunbookName WebhookNSGRule -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName
$AutoAcct.AutomationAccountName)


$JobOut = Get-AzureRmAutomationJobOutput -Id $Job[0].JobId -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName
$AutoAcct.AutomationAccountName


ForEach ($JobCheck in $JobOut){


    $JobCheck.Summary
}


ForEach ($JobCheck in $JobOut){


    $JobCheck.Summary
}
```

```
Logging in to Azure...

Target computer is server1 server1 server1

Malicious IP is 183.129.160.229 183.129.160.229
```

```
Target system is server1

Incoming MaliciousIP is 183.129.160.229

Creating rule...
```

And now if I check against my system, we will see that OMS is auto-generating rules for us!

```
$VM = (Get-AzureRmResource).where({$PSItem.Name -like 'server1'})

$Machine = Get-AzureRmVM -ResourceGroupName $VM[0].ResourceGroupName -Name
$VM[0].Name

$NSG = (Get-AzureRmNetworkSecurityGroup -ResourceGroupName
$Machine.ResourceGroupName).where({$PSItem.NetworkInterfaces.Id -eq
$Machine.NetworkProfile.NetworkInterfaces.Id})

(Get-AzureRmNetworkSecurityRuleConfig -NetworkSecurityGroup
$NSG[0]).where({$PSItem.Name -like "BlockedIP_*"})
```

```
Name                      : BlockedIP_206.190.36.45

Id                        : /subscriptions/f2007bbf-f802-4a47-
9336-
cf7c6b89b378/resourceGroups/test/providers/Microsoft.Network/netw
orkSecurityGroups/server1nsgeus2domain


Controller/securityRules/BlockedIP_206.190.36.45

Etag                      : W/"279e0fee-05c6-43ef-b897-
19f927dd9a40"

ProvisioningState         : Succeeded

Description               : Auto-Generated rule - OMS detected
malicious traffic from 206.190.36.45
```

```
Protocol                  : *

SourcePortRange           : *

DestinationPortRange      : *

SourceAddressPrefix       : 206.190.36.45

DestinationAddressPrefix : *

Access                    : Deny

Priority                  : 100

Direction                 : Inbound



Name                      : BlockedIP_183.129.160.229

Id                        : /subscriptions/f2007bbf-f802-4a47-
9336-
cf7c6b89b378/resourceGroups/test/providers/Microsoft.Network/netw
orkSecurityGroups/server1nsgeus2domain


Controller/securityRules/BlockedIP_183.129.160.229

Etag                      : W/"279e0fee-05c6-43ef-b897-
19f927dd9a40"

ProvisioningState         : Succeeded

Description               : Auto-Generated rule - OMS detected
malicious traffic from 183.129.160.229

Protocol                  : *

SourcePortRange           : *

DestinationPortRange      : *
```

```
SourceAddressPrefix      : 183.129.160.229

DestinationAddressPrefix : *

Access                   : Deny

Priority                 : 101

Direction                : Inbound
```

After letting my system go for about 24 hours, my OMS Alert triggered the runbook an additional five times. Each time generating an additional network security group rule in response to traffic that OMS had recognized as potentially malicious, and thus remediating my problem while I slept.

```
PS C:\WINDOWS\system32> $NSG.SecurityRules | Select-Object Name,Description,Access,Priority,SourceAddressPrefix | Format-Table

Name                      Description                                                                                Access Priority SourceAddressPrefix
----                      -----------                                                                                ------ -------- -------------------
RDPTemp_Inbound           Used as rule placeholder and to be able to jump into the environment for validation checks. Allow       200 *
HTTP_Inbound                                                                                                         Allow       210 *
BlockedIP_206.190.36.45   Auto-Generated rule - OMS detected malicious traffic from 206.190.36.45                     Deny        100 206.190.36.45
BlockedIP_183.129.160.229 Auto-Generated rule - OMS detected malicious traffic from 183.129.160.229                   Deny        101 183.129.160.229
BlockedIP_82.221.105.7    Auto-Generated rule - OMS detected malicious traffic from 82.221.105.7                    Deny        102 82.221.105.7
BlockedIP_61.153.61.50    Auto-Generated rule - OMS detected malicious traffic from 61.153.61.50                    Deny        103 61.153.61.50
BlockedIP_79.77.19.201    Auto-Generated rule - OMS detected malicious traffic from 79.77.19.201                    Deny        104 79.77.19.201
BlockedIP_141.212.122.96  Auto-Generated rule - OMS detected malicious traffic from 141.212.122.96                   Deny        105 141.212.122.96
BlockedIP_139.162.119.197 Auto-Generated rule - OMS detected malicious traffic from 139.162.119.197                   Deny        106 139.162.119.197
```

Using a monitoring tool that can tightly integrate with your automation tools is a necessity in the age of the Cloud.

Chapter 36

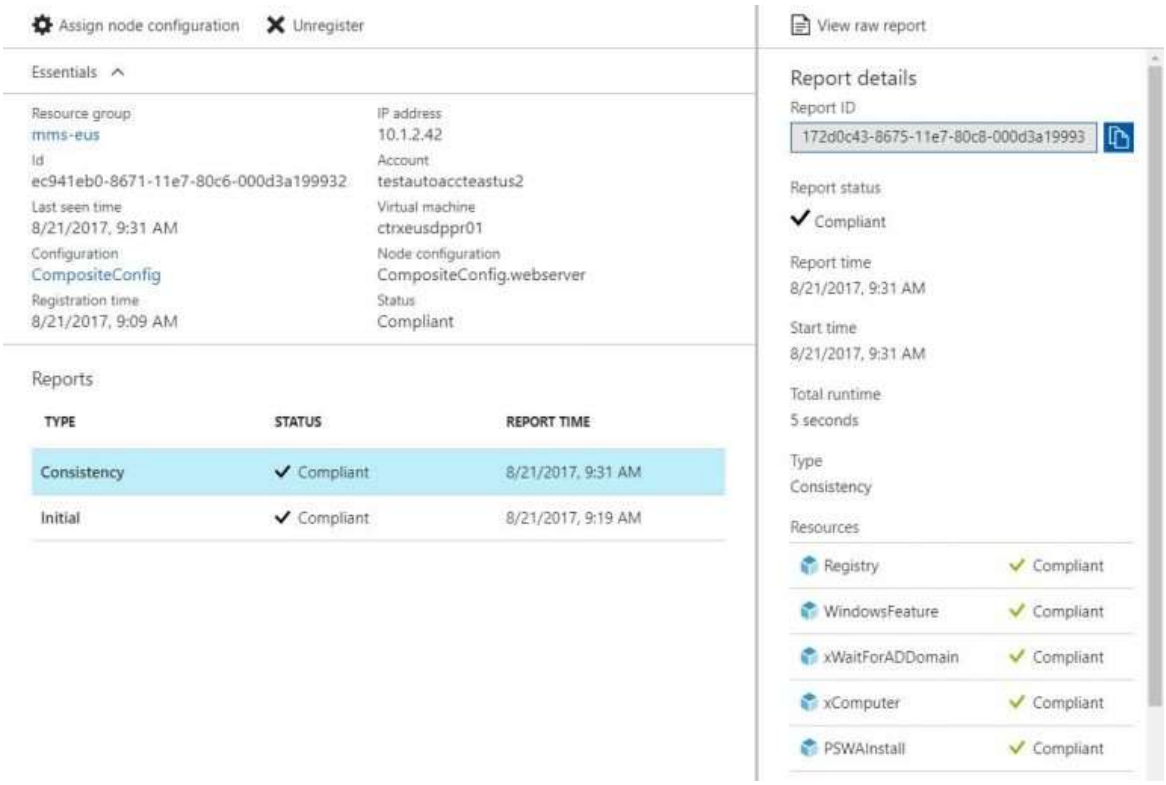
Adding Configuration to your Azure Automation Account using Azure DSC

By: Will Anderson – MVP

I've been wanting write about this while, and with some of the recent changes in Azure Automation DSC, I feel like we can now do a truly complete series. So, let's get started!

Compliance is hard as it is. And as companies start moving more workloads into the cloud, they struggle with compliance even more so. Many organizations are moving to Infrastructure-as-a-Service for a multitude of reasons (both good and bad). As these workloads become more numerous, IT departments are struggling with keeping up with auditing and management needs. Desired State Configuration, as we all know, can provide a path to not only configuring your environments as they deploy as new workloads, but can maintain compliancy, and give you rich reporting.

Yes. Rich reporting from Desired State Configuration, out of the box. You read it right. You can get rich graphical reporting out of Azure Automation Desired State Configuration out of the box. And you can even use it on-prem!



In this series, we're going to be discussing the push and pull methods for Desired State Configuration in Azure. We'll be going over some of the 'gotchas' that you have to keep in mind while deploying your configurations in the Azure environment. And we'll be talking about how we can use hybrid workers to manage systems on-prem using the same tools.

Push vs. Pull

Desired State Configuration, like a datacenter implementation, can be handled via push or pull method. Push method in Azure does not give you reporting, but allows you to deploy your configurations to a new or existing environment. These configurations, and the modules necessary to perform the configuration, are stored in a private blob that you create, and then the

Azure Desired State Configuration extension can be assigned that package. It is then downloaded to the target machine, decompressed, modules installed, and the configuration .mof file generated locally on the system.

Pull method fully uses the capabilities of the Azure Automation Account for storing modules, configurations, and .mof compilations to deploy to systems. The target DSC nodes are registered and monitored through the Azure Automation Account and reporting is generated and made available through the UI. This reporting can also be forwarded to OMS Log Analytics for dashboarding and alerting purposes.

Pros and Cons to Each

So, let's talk about some of the upsides and downsides to each method. These may affect your decisions as you architect your DSC solution.

- **Pricing** - Azure DSC is essentially free. Azure Automation DSC is free for Azure nodes, while there is a cost associated with managed on-prem nodes. This charged per month and is dependent on how often the machines are checking in. You can get more information on the particulars here.
- **Reporting** - If you're looking for rich reporting, Azure Automation DSC is definitely the way to go. You can still get statuses from your Azure DSC nodes via PowerShell, but this leaves the onus on you to format that data and make it look pretty. We'll be taking a look at how we can do this a bit later.
- **Flexibility** - Azure Automation DSC allows you to use modules stored in your Azure Automation Account. If you wish to use a new module, you simply add that module, update your configuration file, and recompile. With Azure DSC, you need to repackage your configuration with all of the modules, re-publish them, and re-push them to your target machines.
- **Side-by-Side Module Versioning Tolerance** - Currently, Azure DSC actually has an advantage over Azure Automation DSC in this respect. You cannot currently have multiple module versions in your module repository. So if you're using Automation DSC and calling the same DSC resources in multiple configs, they need to all be on that same module version.

- **On-Prem Management Capabilities** - Azure Automation DSC has the ability to manage on-prem virtual machines, either directly or via Hybrid Workers. This gives you the ability to manage all of your virtual machines and monitor their configuration status from a single pane of glass. Azure DSC does not have this capability.
- **Managing Systems in AWS** - Yes. You can also manage your virtual machines in AWS using the AWS DSC Toolkit via Azure Automation DSC!

So that's the overview of what we're going to be talking about through this series. Tomorrow, we'll be getting into how to add configurations into Azure Automation DSC and compiling your configs.

Things to Consider

When building configurations for Azure DSC (or anything where we are pulling pre-created .mof files from), there are some things that we need to keep in mind.

Don't embed PowerShell scripts in your configurations. - I spent a lot of time cleaning up my own configurations when learning Azure Automation DSC. When configurations are compiled, they're done so on a virtual machine hidden under the covers and can cause some unexpected behaviours. Some of the issues that I ran into were:

- Using environment variables like $env:COMPUTERNAME - This actually caused me a lot of headaches when I started building systems that were being joined to a domain. The name of the instance that compiles the .mof will be used for $env:COMPUTERNAME instead of the target computer name and you'll be banging your head on the table wondering what happened. Some of the resources that have been published in the gallery have been updated to use a 'localhost' option as a computer name input, such as xActiveDirectory. This takes care of a lot of those headaches.
- Using Parenthetical Commands to establish values - Using something like Get-NetAdapter in a parenthetical argument to get a network adapter of your target system and pass the needed values on to your DSC Resource Providers won't work for the same reasons as above. In this instance, I received a vague error indicating that I was passing an invalid property, and took a little bit of time before I understood what was going on.

- I also ran into an issue with compiling a configuration because I had been using Set-Item to configure the WSMan maxEnvelopeSize in my configs because they can get really big. The error that I received was that WSMan wasn't installed on the machine. It took me a bit to realize that this was because the machine compiling the .mof didn't have WSMan running on the box and it was blowing up on the config.

Instead, if you need to run PowerShell scripts ahead of your deployment, you can use the custom script extension to perform those tasks in Azure, or just put the script into your image on-prem. There is one exception to this, and that's what we'll be talking about next.

Leverage Azure Automation Credential storage where possible - Passing credentials in as a parameter can cause all kinds of issues.

- First and foremost, anyone that is building or deploying those configurations will know those credentials.
- Second of all, it brings the possibility of someone tripping over the keyboard and entering a credential in improperly.

Allowing Azure Automation to tap the credential store during .mof compilation allows to credentials to stay in a secured bubble through the entire process. To pass a credential from Azure Automation to your config, you need to modify the configuration. Simply call Get-AutomationPSCredential to a variable inside your configuration, and then set that variable wherever those credentials are required. Like so:

```
$AdminCreds = Get-AutomationPSCredential -Name $AdminName

Node ($AllNodes.Where{$_.Role -eq "WebServer"}).NodeName
{

        JoinDomain DomainJoin
        {
            DependsOn = "[WindowsFeature]RemoveUI"
            DomainName = $DomainName
```

```
                Admincreds = $Admincreds
                RetryCount = 20
                RetryIntervalSec = 60
            }


    }
```

Azure Automation under the covers will authenticate to the Credentials store with the RunAs account, and then pass those credentials as PSCredential to your DSC resource provider.

Stop Using localhost (or a specific computer name) as the Node Name - Azure Automation DSC allows you to use genericized, but meaningful names to configurations instead of just assigning things to localhost. So now you can use webServer, or domainController, or something that describes the role instead of a machine name. This makes it much easier to decide which configuration should go to what machine.

```
$ConfigData =
@{
    AllNodes =
    @(
        @{
            NodeName = "*"
            PSDscAllowPlainTextPassword = $true
        },


        @{
            NodeName     = "webServer"
            Role         = "WebServer"
        }

        @{
            NodeName = "domainController"
            Role = "domaincontroller"
        }

    )
}
```

Upload the Configuration

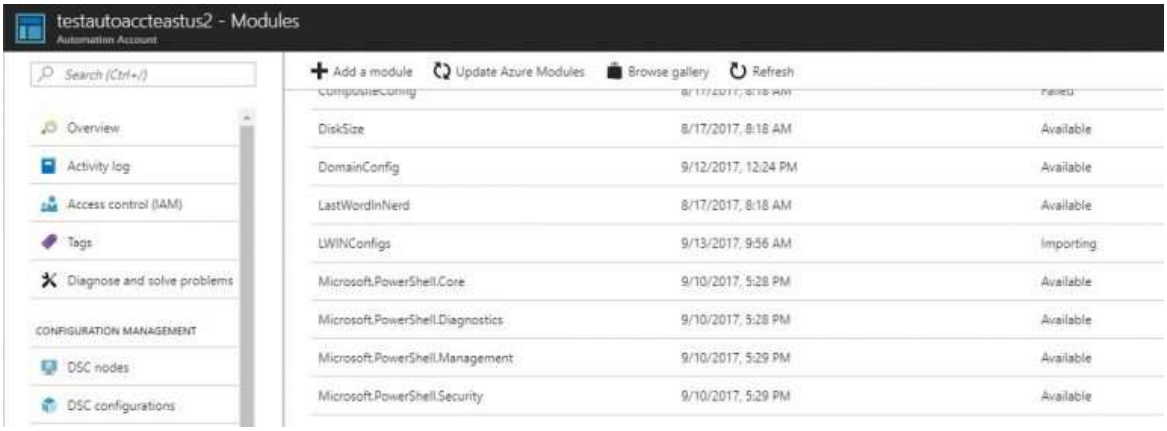
So much like in my previous series on Azure Automation and OMS, we're going to upload our DSC resources to our Automation Account's modules directory. This requires getting the automation account, zipping up our local module files, sending them to a blob store, and importing those modules from the blob store. I've sectioned out the code into different regions to better break it down for your own purposes.

```
#region GetAutomationAccount


$AutoResGrp = Get-AzureRmResourceGroup -Name 'mms-eus'

$AutoAcct = Get-AzureRmAutomationAccount -ResourceGroupName
$AutoResGrp.ResourceGroupName


#endregion


#region compress configurations


    Set-Location C:\Scripts\Presentations\AzureAutomationDSC\ResourcesToUpload

    $Modules = Get-ChildItem -Directory


    ForEach ($Mod in $Modules){


        Compress-Archive -Path $Mod.PSPath -DestinationPath ((Get-Location).Path
+ '\' + $Mod.Name + '.zip') -Force


    }



#endregion


#region Access blob container
```

```
$StorAcct = Get-AzureRmStorageAccount -ResourceGroupName
$AutoAcct.ResourceGroupName


Add-AzureAccount

$AzureSubscription = ((Get-AzureSubscription).where({$PSItem.SubscriptionName -
eq $Sub.Name}))

Select-AzureSubscription -SubscriptionName $AzureSubscription.SubscriptionName -
Current

$StorKey = (Get-AzureRmStorageAccountKey -ResourceGroupName
$StorAcct.ResourceGroupName -Name
$StorAcct.StorageAccountName).where({$PSItem.KeyName -eq 'key1'})

$StorContext = New-AzureStorageContext -StorageAccountName
$StorAcct.StorageAccountName -StorageAccountKey $StorKey.Value

$Container = Get-AzureStorageContainer -Name ('modules') -Context $StorContext




#endregion


#region upload zip files


$ModulesToUpload = Get-ChildItem -Filter "*.zip"


ForEach ($Mod in $ModulesToUpload){


        $Blob = Set-AzureStorageBlobContent -Context $StorContext -Container
$Container.Name -File $Mod.FullName -Force


        New-AzureRmAutomationModule -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName
$AutoAcct.AutomationAccountName -Name ($Mod.Name).Replace('.zip','') -
ContentLink $Blob.ICloudBlob.Uri.AbsoluteUri


}
```

```
#endregion
```

Once we've uploaded our files, we can monitor them to ensure that they've imported successfully via the UI, or by using the Get-AzureRmAutomationModule command.




```
Get-AzureRmAutomationModule -Name LWINConfigs -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName $AutoAcct.AutomationAccountNa
me
```

```
ResourceGroupName      : mms-eus

AutomationAccountName : testautoaccteastus2

Name                   : LWINConfigs

IsGlobal               : False

Version                : 1.0.0.0
```

```
SizeInBytes           : 5035

ActivityCount         : 1

CreationTime          : 9/13/2017 9:56:10 AM -04:00

LastModifiedTime      : 9/13/2017 9:57:26 AM -04:00

ProvisioningState     : Succeeded
```

Compile the Configuration

Once we've uploaded our modules, we can then upload and compile our configuration. For this, we'll use the Import-AzureRmAutomationDscConfiguration command. But before we do, there's two things to note when formatting a configuration for deployment to Azure Automation DSC.

- The configuration name has to match the name of the configuration file. So if your configuration is called SqlServerConfig, your config file has to be called SqlServerConfig.ps1.
- The sourcepath parameter errors out with an 'invalid argument specified' error if you use a string path. Instead, it works if you use (Get-Item).FullName

We'll be casting this command to a variable, as we'll be using it later on when we compile the configuration. You'll also want to use the publish parameter to publish the configuration after importation, and if you're overwriting a configuration you'll want to leverage the force parameter.

```
$Config = Import-AzureRmAutomationDscConfiguration -SourcePath (Get-Item
C:\Scripts\Presentations\AzureAutomationDSC\TestConfig.ps1).FullName -
AutomationAccountName $AutoAcct.AutomationAccountName -ResourceGroupName
$AutoAcct.ResourceGroupName -Description DemoConfiguration -Published -Force
```

```
ResourceGroupName     : mms-eus
AutomationAccountName : testautoaccteastus2
Location              : eastus2
State                 : Published
Name                  : TestConfig
Tags                  : {}
CreationTime          : 9/13/2017 10:43:24 AM -04:00
LastModifiedTime      : 9/13/2017 10:43:24 AM -04:00
Description           : DemoConfiguration
Parameters            : {AutomationAccountName, AdminName, ResourceGroupName, domainName}
LogVerbose            : False
```

Now that our configuration is published, we can compile it. So let's add our parameters and configuration data:

```
$Parameters = @{

            'DomainName' = 'lwinerd.local'
            'ResourceGroupName' = $AutoAcct.ResourceGroupName
            'AutomationAccountName' = $AutoAcct.AutomationAccountName
            'AdminName' = 'lwinadmin'

}

$ConfigData =
@{
    AllNodes =
    @(
        @{
            NodeName = "*"
            PSDscAllowPlainTextPassword = $true
        },
```

```
        @{
            NodeName      = "webServer"
            Role          = "WebServer"
        }

        @{
            NodeName = "domainController"
            Role = "domaincontroller"
        }

    )
}
```

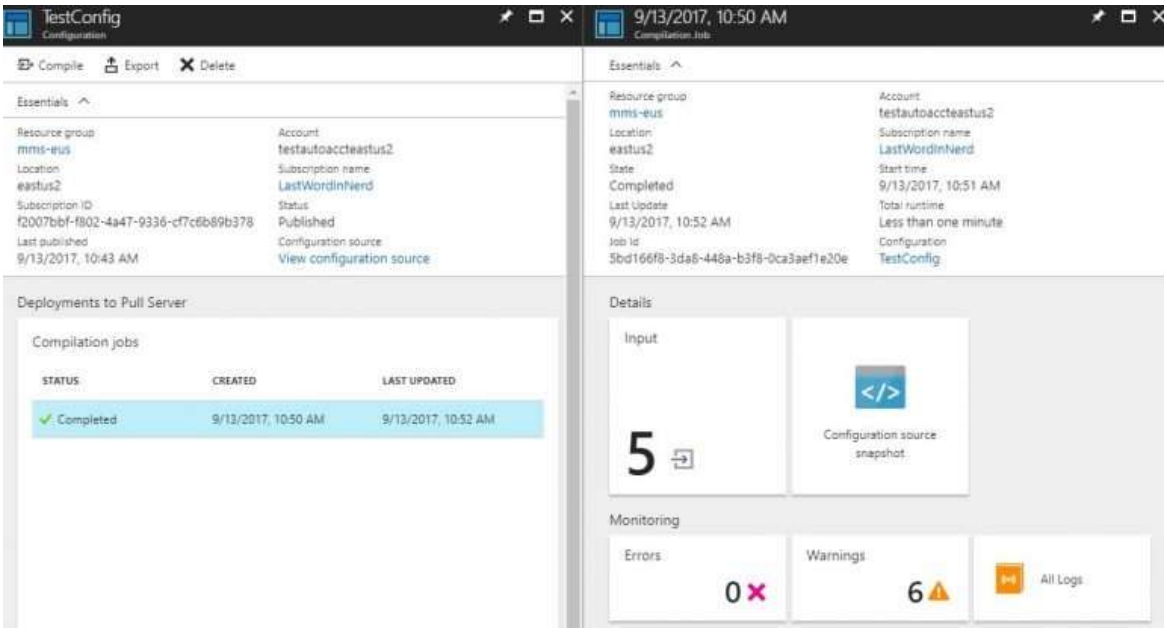
You'll notice that I have PSDscAllowPlainTextPassword set to true for all of my nodes. This is to allow the PowerShell instance on the compilation node to compile the configuration with credentials being passed into it. This PowerShell instance isn't aware that once the .mof is compiled, it is encrypted by Azure Automation before it's stored in the Automation Account.

Now that we have our parameters and configuration data set, we can pass this to our Start-AzureRmAutomationDscCompilationJob command to kick off the .mof compilation.

```
$DSCComp = Start-AzureRmAutomationDscCompilationJob -AutomationAccountName
$AutoAcct.AutomationAccountName -ConfigurationName $Config.Name -
ConfigurationData $ConfigData -Parameters $Parameters -ResourceGroupName
$AutoAcct.ResourceGroupName
```

And now we can use the Get-AzureRmAutomationDscCompilationJob command to check the status of the compilation, or check through the UI.

```
Get-AzureRmAutomationDscCompilationJob -Id $DSCComp.Id -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName
$AutoAcct.AutomationAccountName
```



The compilation itself can take up to around five minutes, so grab yourself a cup of coffee. Once it returns as complete, we can get to registering our endpoints and delivering our configurations to them.

Chapter 37

Onboarding Automation DSC Endpoints and Reporting

By: Will Anderson - MVP

In the last chapter we talked about modifying and uploading our configurations to Azure Automation DSC. We were able to import credentials from Azure's Automation Account Credential store, and then compile the .mof files in the automation account for deployment. In this chapter, we'll be looking at how we apply those configurations to existing systems via PowerShell. Then we'll take a look at some of the reporting available via Azure Automation DSC and send those reports over to Operations Management Suite for dashboarding.

So, when we left off. We successfully published our configurations in Automation DSC. If we run Get-AzureRmAutomationDscNodeConfiguration against the configuration I published, we get the following:

```
Get-AzureRmAutomationDscNodeConfiguration -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName
$AutoAcct.AutomationAccountName -ConfigurationName TestConfig
```

```
PS C:\Scripts\Presentations\AzureAutomationDSC\ResourcesToUpload>
ConfigurationName TestConfig


ResourceGroupName     : mms-eus
AutomationAccountName : testautoaccteastus2
Name                  : TestConfig.domainController
CreationTime          : 9/13/2017 10:52:31 AM -04:00
LastModifiedTime      : 9/13/2017 10:52:31 AM -04:00
ConfigurationName     : TestConfig
RollupStatus          : Good

ResourceGroupName     : mms-eus
AutomationAccountName : testautoaccteastus2
Name                  : TestConfig.webServer
CreationTime          : 9/13/2017 10:52:31 AM -04:00
LastModifiedTime      : 9/13/2017 10:52:31 AM -04:00
ConfigurationName     : TestConfig
RollupStatus          : Good
```

As you can see, when we published the configuration, it generated two configuration .mofs based on our node names - domainController and webServer. Now of course, we're not going to be calling our servers webServer and domainController, rather, these are generalized names for our configurations. We get the root configuration (TestConfig), and then the node specific configuration based on the root document (webServer or domainController). This gives us a lot of flexibility as we can now statefully name our configurations, and assign them to machines without dealing with guids or having all of the mofs defined by a computer name or any other nonsense! We just assign what named configuration goes to what system, and away we go.

We don't even really care what the computer name is, as long as the correct config gets assigned. This is really helpful when working on Azure Resource Manager templates, because I don't even really know what the system name will be until runtime. I just designate a set of systems as 'webServer', assign the config and deploy.

Register the Virtual Machine

So, let's go ahead and get a system that we want to target. I just so happen to have one in Azure right here:

```
$TargetResGroup = 'nrdtste'
$VMName = 'ctrxeusdbnp01'

$VM = Get-AzureRmVM -ResourceGroupName $TargetResGroup -Name $VMName
```

Now that we have our VM object, we're going to create a hash-table with some configuration items for the DSC Local Configuration Manager on the target system.

```
$DSCLCMConfig = @{

    'ConfigurationMode' = 'ApplyAndAutocorrect'
    'RebootNodeIfNeeded' = $true
    'ActionAfterReboot' = 'ContinueConfiguration'

}
```

Once we have all of this, we can now go ahead and register our target node in Automation DSC using the Register-AzureRmAutomationDscNode command.

```
Register-AzureRmAutomationDscNode -AzureVMName $VM.Name -AzureVMResourceGroup
$VM.ResourceGroupName -AzureVMLocation $VM.Location -AutomationAccountName
$AutoAcct.AutomationAccountName -ResourceGroupName $AutoAcct.ResourceGroupName
@DSCLCMConfig
```

You might note with this command that you can also assign it a configuration as you register the node. However, I've had occasional issues with this method. So we're going to go ahead and register the node first, then assign the configuration. As another note, while the system is being registered, the command will hold your session until it returns a success or failure. So grab another cup of coffee and enjoy it for a few minutes while we wait.

```
PS C:\Scripts\Presentations\AzureAutomationDSC\ResourcesToUpload> Register-AzureRmAutomationDscNode -AzureVMName $VM.Name -AzureVMResourceGroup $VM.ResourceGroupName -AzureVMLo
untName $AutoAcct.AutomationAccountName -ResourceGroupName $AutoAcct.ResourceGroupName @DSCLCMConfig


DeploymentName          : 20170913125950
ResourceGroupName       : nrdtste
ProvisioningState       : Succeeded
Timestamp               : 9/13/2017 5:04:25 PM
Mode                    : Incremental
TemplateLink            :
                          Uri            : https://eus2oaasibizamarketprod1.blob.core.windows.net/automationdscpreview/azuredeployV2.json
                          ContentVersion : 1.0.0.0

Parameters              :
                          Name              Type                       Value
                          ==============    ========================   ==========
                          vmName            String                     ctrxeusdbnp01
                          location          String                     eastus
                          modulesUrl        String                     https://eus2oaasibizamarketprod1.blob.core.windows.net/automationdscpreview/RegistrationMetaConfigV2.zip
                          configurationFunction  String                     RegistrationMetaConfigV2.ps1\RegistrationMetaConfigV2
                          registrationKey   SecureString
                          registrationUrl   String                     https://eus2-agentservice-prod-1.azure-automation.net/accounts/e0a66fa1-071f-41c1-9521-960a9b24d0da
                          nodeConfigurationName  String
                          configurationMode  String                     ApplyAndAutocorrect
                          configurationModeFrequencyMins  Int                     15
                          refreshFrequencyMins  Int                     30
                          rebootNodeIfNeeded  Bool                     True
                          actionAfterReboot  String                     ContinueConfiguration
                          allowModuleOverwrite  Bool                     False
                          timestamp         String                     9/13/2017 4:59:50 PM

Outputs                 :
DeploymentDebugLogLevel :
```

Apply a Configuration

Now we can see our machine has registered successfully. But if we run the Get-AzureRmAutomationDscNode command, we can see that the NodeConfigurationName property is empty. So, let's fix that.

```
PS C:\Scripts\Presentations\AzureAutomationDSC\ResourcesToUpload> Get-AzureRmAutomationDscNode -Name $V


ResourceGroupName     : mms-eus
AutomationAccountName : testautoaccteastus2
Name                  : ctrxeusdbnp01
RegistrationTime      : 9/13/2017 1:03:51 PM -04:00
LastSeen              : 9/13/2017 1:04:04 PM -04:00
IpAddress             : 10.1.2.6;127.0.0.1;fe80::b0c1:132b:c417:4fd6%13;::2000:0:0:0;::1;::2000:0:0:0
Id                    : 7c84a881-98a5-11e7-80c5-000d3a1155ab
NodeConfigurationName :
Status                : Compliant
```

What we need to do is capture the configuration we want to apply, so we do this by grabbing it with Get-AzureRmAutomationDscNodeConfiguration. Then, we'll capture the target DSC endpoint with the Get command we previously used, and cast both objects to our Set-AzureRmAutomationDscNode command to apply the configuration to the appropriate node.

```
$Configuration = Get-AzureRmAutomationDscNodeConfiguration -
AutomationAccountName $AutoAcct.AutomationAccountName -ResourceGroupName
$AutoAcct.ResourceGroupName -Name 'CompositeConfig.webServer'
```

```
$TargetNode = Get-AzureRmAutomationDscNode -Name $VM.Name -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName
$AutoAcct.AutomationAccountName

Set-AzureRmAutomationDscNode -Id $TargetNode.Id -NodeConfigurationName
$Configuration.Name -AutomationAccountName $AutoAcct.AutomationAccountName -
ResourceGroupName $AutoAcct.ResourceGroupName -Verbose -Force
```

After a couple of seconds, we can see that the configuration has been assigned to our node. Once the LCM hits it's next review cycle, it'll pick up the configuration and start applying:

```
$TargetNode = Get-AzureRmAutomationDscNode -Name $VM.Name -ResourceGroupName $AutoAcct.ResourceGroupNa
Set-AzureRmAutomationDscNode -Id $TargetNode.Id -NodeConfigurationName $Configuration.Name -Automation
VERBOSE: Performing the operation "Updating the node configuration assignment for this node" on target


ResourceGroupName     : mms-eus
AutomationAccountName : testautoaccteastus2
Name                  : ctrxeusdbnp01
RegistrationTime      : 9/13/2017 1:03:51 PM -04:00
LastSeen              : 9/13/2017 1:04:04 PM -04:00
IpAddress             : 10.1.2.6;127.0.0.1;fe80::b0c1:132b:c417:4fd6%13;::2000:0:0:0;::1;::2000:0:0:0
Id                    : 7c84a881-98a5-11e7-80c5-000d3a1155ab
NodeConfigurationName : CompositeConfig.webServer
Status                :
```

We can check on the status of our target node by using the Get-AzureRmAutomationDscNodeReport command like so to get some useful information:

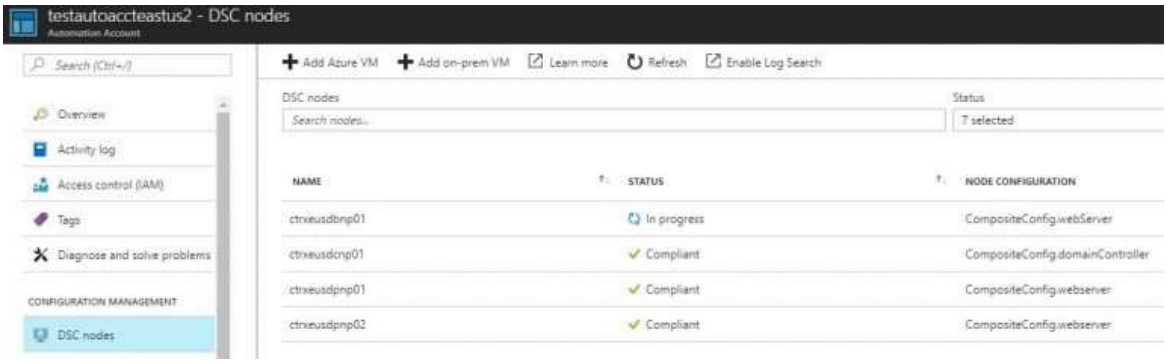
```
Get-AzureRmAutomationDscNodeReport -NodeId $TargetNode.Id -ResourceGroupName
$AutoAcct.ResourceGroupName -AutomationAccountName
$AutoAcct.AutomationAccountName -Latest
```

And it will output some pretty useful information.

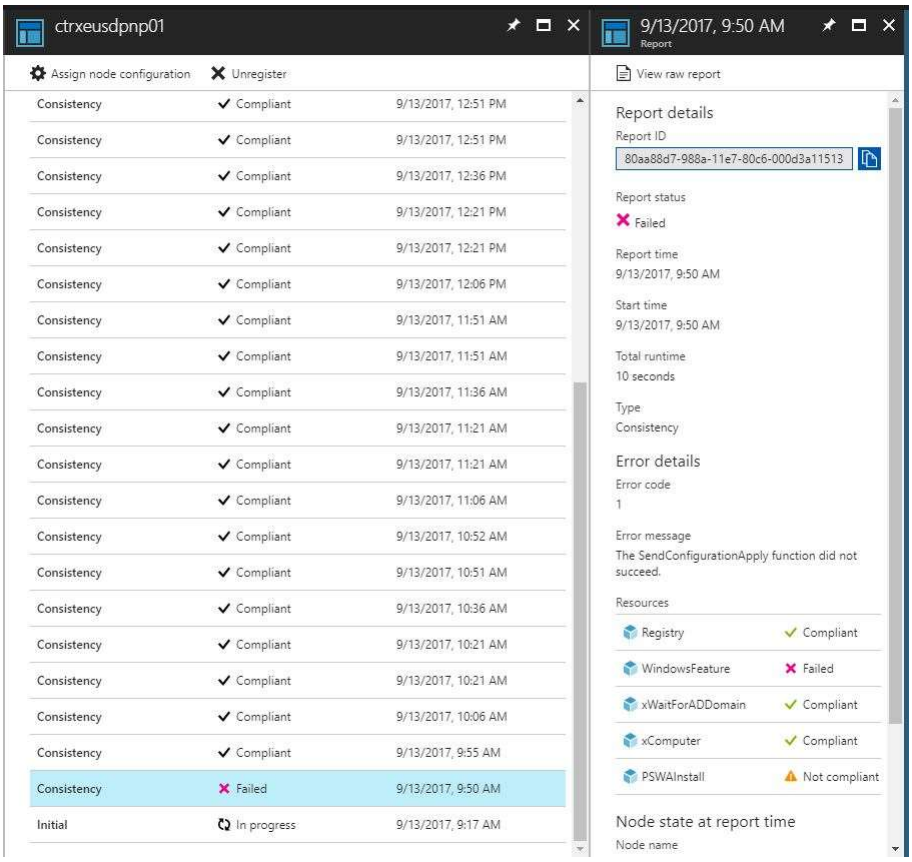
```
ResourceGroupName     : mms-eus
AutomationAccountName : testautoaccteastus2
StartTime             : 9/13/2017 1:33:48 PM -04:00
LastModifiedTime      : 9/13/2017 1:33:49 PM -04:00
EndTime               : 9/13/2017 1:33:49 PM -04:00
ReportType            : Consistency
Id                    : b2e2e788-98a9-11e7-80c5-000d3a1155ab
NodeId                : 7c84a881-98a5-11e7-80c5-000d3a1155ab
Status                : InProgress
RefreshMode           :
RebootRequested       :
ReportFormatVersion   : 2.0
```

Azure Automation DSC Reports

This is where I have to admit that the UI really shines. You can see all of your systems at a glance, with what configuration is assigned and it's current state.



Furthermore, you can actually drill down through the nodes to see what resources are being applied, what their dependencies are, and what the state of the particular configuration item is.



There is a wealth of data that you can find here in an easy to read dashboard. Furthermore, you can connect this to a Log Analytics instance (or other products that support restful API), and ship it up for alerting and more dashboarding.

Connecting to Log Analytics

So connecting your Azure Automation DSC is pretty straightforward. To be able to use it, you need to have an OMS tier that includes the Automation and Control offering to start. If you do, then all you have to do is follow a couple of simple commands.

First, we have to get the resourceIds for the Automation Account and the Log Analytics workspace.

```
#Get the resourceId of the automation account.

$AutoAcctResource = Find-AzureRmResource -ResourceType
"Microsoft.Automation/automationAccounts" -ResourceNameContains
'testautoaccteastus2'


#Get the resourceId of the Log Analytics Workspace

$LogAnalyticsResource = Find-AzureRmResource -ResourceType
"Microsoft.OperationalInsights/workspaces" -ResourceNameContains 'LWINerd'
```

Then we can use those resourceIds to pass to Set-AzureRmDiagnosticSetting and specify our DSCNodeStatus category.

```
Set-AzureRmDiagnosticSetting -ResourceId $AutoAcctResource.ResourceId -
WorkspaceId $LogAnalyticsResource.ResourceId -Enabled $true -Categories
"DscNodeStatus" -Verbose
```

Then you'll get a return similar to this:

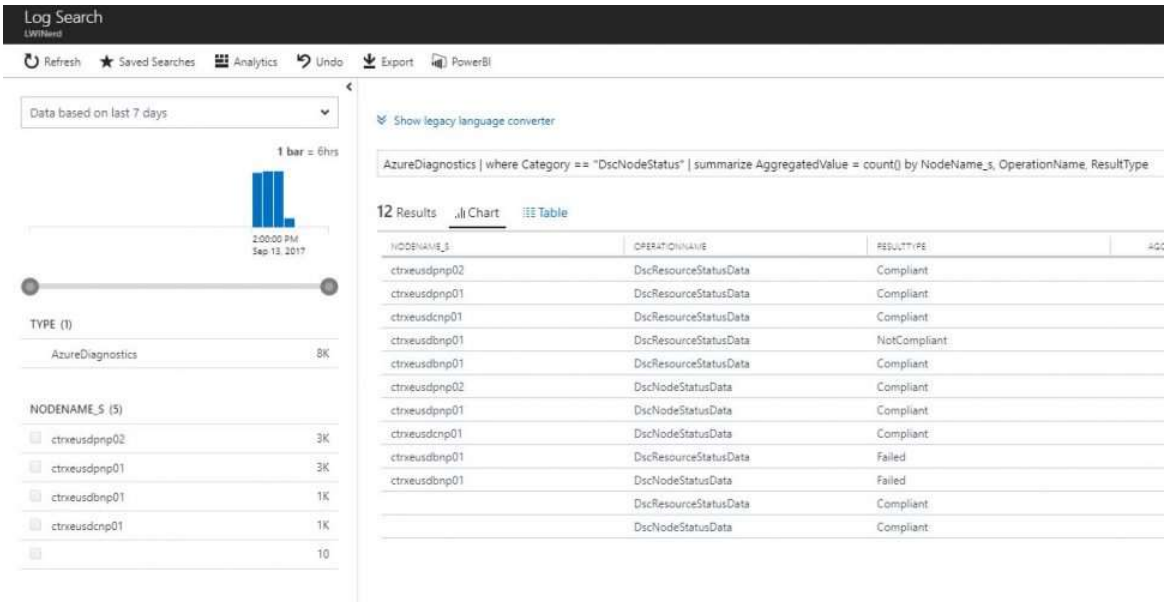
```
Set-AzureRmDiagnosticSetting -ResourceId $AutoAcctResource.ResourceId -
WorkspaceId $LogAnalyticsResource.ResourceId -Enabled $true -Categories "D

scNodeStatus" -Verbose
```

```
StorageAccountId             :
```

```
ServiceBusRuleId            :

EventHubAuthorizationRuleId :

Metrics

    TimeGrain       : PT1M

    Enabled         : False

    RetentionPolicy

    Enabled : False

    Days    : 0




Logs

    Category        : JobLogs

    Enabled         : False

    RetentionPolicy

    Enabled : False

    Days    : 0




    Category        : JobStreams

    Enabled         : False

    RetentionPolicy
```

```
    Enabled : False

    Days    : 0




    Category        : DscNodeStatus

    Enabled         : True

    RetentionPolicy

    Enabled : False

    Days    : 0




WorkspaceId                 : /subscriptions/f2007bbf-f802-4a47-
9336-cf7c6b89b378/resourceGroups/mms-
eus/providers/Microsoft.OperationalInsights/workspaces/LWINerd

Id                          :

/subscriptions/f2007bbf-f802-4a47-9336-
cf7c6b89b378/resourcegroups/mms-
eus/providers/microsoft.automation/automationaccounts/testautoacc
teastus2/providers/microsoft.insights/diagnosticSettings/service

Name                        : service

Type                        :

Location                    :

Tags                        :
```

After a little while, we can check back to our log search and start performing queries and configuring alerts.



So that's Azure Automation DSC in a nutshell! But don't worry, I haven't forgotten about Azure DSC's push method. We will be talking about that in the next chapter.

Chapter 38

Publishing Configurations and Pushing them with Azure DSC

By: Will Anderson – MVP

So, we've talked about Azure Automation DSC and the extensive reporting we can get from it. With the pricing as it is, it would be hard to argue as to why you would want to use anything else. But I'm a completionist, and there may be some edge cases that might come up where you wouldn't be able to use the pull method for configurations. So let's talk about how you can use Azure DSC to push a configuration to a virtual machine.

So, let's get started!

Publish the Configuration

In order to push a configuration, we need to publish it to a blob store. When you use Publish-AzureRmVmDscConfiguration, the command bundles all of the required modules along with the configuration into a .zip file. It does this by pulling the modules from your local machine that you're running the command from, so you'll need to make sure that you have the appropriate modules installed on your system.

First, we'll go ahead and grab a storage account where these binaries can be published. In the storage account, we have a blob store for our configurations. This blob store is a private store.

```
$AutoResGrp = Get-AzureRmResourceGroup -Name 'mms-eus'

$StorAcct = Get-AzureRmStorageAccount -ResourceGroupName
$AutoResGrp.ResourceGroupName -Name 'modulestor'
```

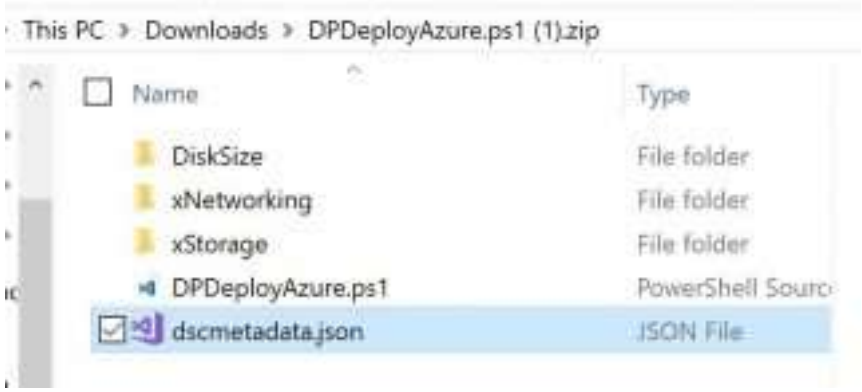
Now that we have our private store, we're going to publish our configuration using the Publish-AzureRmVMDscConfiguration command.

```
$DSCBlob = Publish-AzureRmVMDscConfiguration -ConfigurationPath
C:\Scripts\Configs\cmdpconfig.ps1 -ResourceGroupName $StorAcct.ResourceGroupName
-ContainerName 'dscpushconfig' -StorageAccountName $StorAcct.StorageAccountName
-Force
```

```
$Archive = $DSCBlob.Split('/') | Select-Object -Last 1
```

As previously mentioned, the command reads your configuration, and then grabs the necessary modules from your local machine and adds them to the package when it publishes the configuration. This way, the machine has all of the necessary bits to perform the configuration. You can actually validate this by downloading the packaged .zip file from the blob store and seeing for yourself.

Along with the modules and configuration, you'll also find a dscmetadata.json file that is essentially a manifest of the required modules.



Install the VM Extensions

Now that our binaries have been published, we can get our target machine and deploy the Azure DSC VM extension to it while assigning the configuration. When you deploy the extension, it's best to use the latest version available. If you want to check which version is the latest, you can check out the release history on the PowerShell Team Blog.

```
$ArmVmRsg = Get-AzureRmResourceGroup -Name 'nrdtste'

$ArmVm = Get-Azurermvm -ResourceGroupName $ArmVmRsg.ResourceGroupName -Name
'ctrxeusdbnp01'

Set-AzureRmVMDscExtension -ArchiveResourceGroupName $StorAcct.ResourceGroupName
-ArchiveBlobName $Archive -ResourceGroupName $ArmVm.ResourceGroupName -
ArchiveStorageAccountName $StorAcct.StorageAccountName -ArchiveContainerName
'dscpushconfig' -Version '2.26' -VMName $ArmVm.Name -ConfigurationName
'CMDPConfig' -Verbose
```

Like with Azure Automation DSC, when you register the VM extension, your PowerShell session will be held open until the extension returns a success or failure status. Once it returns, you can check the status of the configuration using Get-AzureRmVmDscExtensionStatus.

```
Get-AzureRmVMDscExtensionStatus -ResourceGroupName $ArmVm.ResourceGroupName -
VMName $ArmVm.Name
```

```
ResourceGroupName   : nrdtst3

VmName              : ctrxeusdbnp01

Version             : 2.26

Status              : Provisioning succeeded

StatusCode          : ProvisioningState/succeeded

Timestamp           : 10/9/2017 1:12:22 PM

StatusMessage       : DSC configuration was applied successfully.

DscConfigurationLog : {[2017-10-09 13:11:18Z] [VERBOSE]
[ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] The operation 'Get-WindowsFeature'
succeeded: Server-Gui-Shell, [2017-10-09

                      13:11:18Z] [VERBOSE] [ctrxeusdbnp01]: LCM:
[ End    Test     ]  [[WindowsFeature]RemoveUI]  in 9.5980
seconds., [2017-10-09 13:11:18Z] [VERBOSE] [ctrxeusdbnp01]: LCM:
[ Start  Set

                      ]  [[WindowsFeature]RemoveUI], [2017-10-09
13:11:19Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] Uninstallation started......}
```

If you want to dive a little deeper, we can of course grab the specific DscConfigurationLog information:

```
(Get-AzureRmVMDscExtensionStatus -ResourceGroupName $ArmVm.ResourceGroupName -
VMName $Armvm.Name).DscConfigurationLog
```

```
[2017-10-09 13:11:18Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] The operation 'Get-WindowsFeature'
succeeded: Server-Gui-Shell

[2017-10-09 13:11:18Z] [VERBOSE] [ctrxeusdbnp01]: LCM:  [ End
Test     ]  [[WindowsFeature]RemoveUI]  in 9.5980 seconds.

[2017-10-09 13:11:18Z] [VERBOSE] [ctrxeusdbnp01]: LCM:  [ Start
Set      ]  [[WindowsFeature]RemoveUI]

[2017-10-09 13:11:19Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] Uninstallation started...

[2017-10-09 13:11:19Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] Continue with removal?

[2017-10-09 13:11:19Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] Prerequisite processing started...

[2017-10-09 13:11:24Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] Prerequisite processing succeeded.

[2017-10-09 13:12:21Z] [WARNING] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] You must restart this server to finish
the removal process.

[2017-10-09 13:12:21Z] Settings handler status to 'transitioning'
(C:\Packages\Plugins\Microsoft.Powershell.DSC\2.26.1.0\Status\0.s
tatus)

[2017-10-09 13:12:21Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] Uninstallation succeeded.

[2017-10-09 13:12:21Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] Successfully uninstalled the feature
Server-Gui-Shell.
```

```
[2017-10-09 13:12:21Z] [VERBOSE] [ctrxeusdbnp01]:
[[WindowsFeature]RemoveUI] The Target machine needs to be
restarted.

[2017-10-09 13:12:21Z] [VERBOSE] [ctrxeusdbnp01]: LCM:  [ End
Set      ]  [[WindowsFeature]RemoveUI]  in 62.7090 seconds.

[2017-10-09 13:12:21Z] [VERBOSE] [ctrxeusdbnp01]: LCM:  [ End
Resource ]  [[WindowsFeature]RemoveUI]

[2017-10-09 13:12:21Z] [VERBOSE] [ctrxeusdbnp01]:
[] A reboot is required to progress further. Please reboot the
system.

[2017-10-09 13:12:21Z] [WARNING] [ctrxeusdbnp01]:
[] A reboot is required to progress further. Please reboot the
system.

[2017-10-09 13:12:21Z] [VERBOSE] [ctrxeusdbnp01]: LCM:  [ End
Set      ]

[2017-10-09 13:12:21Z] [VERBOSE] [ctrxeusdbnp01]: LCM:  [ End
Set      ]    in  74.8080 seconds.

[2017-10-09 13:12:21Z] [VERBOSE] Operation 'Invoke CimMethod'
complete.

[2017-10-09 13:12:21Z] [VERBOSE] Time taken for configuration job
to complete is 75.071 seconds
```

As you can see, the configuration is complete pending a reboot. This brings us to a few of the caveats associated with the push method for Azure DSC.

- Unfortunately, unlike with the Register-AzurRmAutomationDscNodeConfiguration command available for Azure Automation, you cannot currently configure the LCM direct from the command. Instead, you'll want to add a LocalConfigurationManager block to your top level config to set any attributes for the LCM.

- As the system is downloading the packaged modules and configuration files, the mof file is configured locally on the machine. While the current.mof file is encrypted, there is a copy of the mof that is generated in the C:\Packages\Plugins\Microsoft.Powershell.DSC\<pluginVersion>\<configuration>\ directory. You'll want to be careful as to what you're passing in plain text in that regard.
- You can retrieve the DscConfigurationLog data for validation of your configs and the state of the machines, but this process requires automation and can take some time to compile.

So, now we've explore Azure Desired State Configuration using the available push and pull methods. And we've explored the rich reporting capabilities that are available to you in Azure Automation DSC. It's been a long journey, but I hope you've found this content to be useful to you!

Chapter 39

Testing RDMA Connectivity with PowerShell

By: Dave Kawula – MVP

I have been doing a lot of deployments of Microsoft Hyper Converged Storage solution called Storage Spaces Direct. Part of this configuration is setting up the network stack properly. I found this lovely little script from Microsoft to help us with just that

This script from Microsoft can be downloaded from here:
https://github.com/Microsoft/SDN/blob/master/Diagnostics/Test-Rdma.ps1

It includes some great little options to run not only all your core RDMA Tests and gain a better understanding of the RDMA PowerShell Commands but also includes a cool DiskSpd.exe test to validate connectivity to a remote host.

```
PS C:\Post-Install> .\RDMA_Test_Config.ps1

cmdlet RDMA_Test_Config.ps1 at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
IfIndex: 7
IsRoCE: true
VERBOSE: Diskspd.exe found at C:\Post-Install\diskspd.exe
VERBOSE: The adapter vEthernet (Storage 1) is a virtual adapter
VERBOSE: Retrieving vSwitch bound to the virtual adapter
VERBOSE: Found vSwitch: Embedded_vSwitch_Team
VERBOSE: Found the following physical adapter(s) bound to vSwitch: Ethernet 6, Ethernet 5
VERBOSE: Underlying adapter is RoCE. Checking if QoS/DCB/PFC is configured on each physical adapter(s)
VERBOSE: QoS/DCB/PFC configuration is correct.
VERBOSE: RDMA configuration is correct.
VERBOSE: Checking if remote IP address, 172.19.0.5, is reachable.
VERBOSE: Remote IP 172.19.0.5 is reachable.
VERBOSE: Disabling RDMA on adapters that are not part of this test. RDMA will be enabled on them later.
VERBOSE: Testing RDMA traffic now for. Traffic will be sent in a parallel job. Job details:
VERBOSE: 206275123 RDMA bytes written per second
VERBOSE: 13510401 RDMA bytes sent per second
VERBOSE: 308248809 RDMA bytes written per second
VERBOSE: 11479742 RDMA bytes sent per second
VERBOSE: 380304922 RDMA bytes written per second
VERBOSE: 11690536 RDMA bytes sent per second
VERBOSE: 347692235 RDMA bytes written per second
VERBOSE: 13262423 RDMA bytes sent per second
VERBOSE: 371318461 RDMA bytes written per second
VERBOSE: 13199097 RDMA bytes sent per second
VERBOSE: 384073440 RDMA bytes written per second
VERBOSE: 14139847 RDMA bytes sent per second
VERBOSE: 397201136 RDMA bytes written per second
VERBOSE: 13951179 RDMA bytes sent per second
VERBOSE: 409656704 RDMA bytes written per second
VERBOSE: 14411121 RDMA bytes sent per second
VERBOSE: 395865859 RDMA bytes written per second
VERBOSE: 14512234 RDMA bytes sent per second
VERBOSE: 402716795 RDMA bytes written per second
VERBOSE: 14263627 RDMA bytes sent per second
VERBOSE: 400683005 RDMA bytes written per second
VERBOSE: 14920635 RDMA bytes sent per second
```

```
VERBOSE: 13053387 RDMA bytes sent per second
VERBOSE: 314830009 RDMA bytes written per second
VERBOSE: 11476856 RDMA bytes sent per second
VERBOSE: 368139962 RDMA bytes written per second
VERBOSE: 13162535 RDMA bytes sent per second
VERBOSE: 372233062 RDMA bytes written per second
VERBOSE: 14614994 RDMA bytes sent per second
VERBOSE: 415916960 RDMA bytes written per second
VERBOSE: 3749799 RDMA bytes sent per second
VERBOSE: Enabling RDMA on adapters that are not part of this test. RDMA was disabled on them prior to sending RDMA traff
ic.
VERBOSE: RDMA traffic test SUCCESSFUL: RDMA traffic was sent to 172.19.0.5
PS C:\Post-Install>
```

Here is the script itself:

```
[CmdletBinding()]
```

```
Param(
  [Parameter(Mandatory=$True, Position=1, HelpMessage="Interface index of the
adapter for which RDMA config is to be verified")]
  [string] $IfIndex,
  [Parameter(Mandatory=$True, Position=2, HelpMessage="True if underlying fabric
type is RoCE. False for iWarp or IB")]
  [bool] $IsRoCE,
  [Parameter(Position=3, HelpMessage="IP address of the remote RDMA adapter")]
  [string] $RemoteIpAddress,
  [Parameter(Position=4, HelpMessage="Full path to the folder containing
diskspd.exe")]
  [string] $PathToDiskspd
)


if ($RemoteIpAddress -ne $null)
{
    if (($PathToDiskspd -eq $null) -Or ($PathToDiskspd -eq ''))
    {
        $PathToDiskspd = "C:\Windows\System32"
    }

    $FullPathToDiskspd = $PathToDiskspd + "\diskspd.exe"
    if ((Test-Path $FullPathToDiskspd) -eq $false)
    {
        Write-Host "ERROR: Diskspd.exe not found at" $FullPathToDiskspd ".
Please download diskspd.exe and place it in the specified location. Exiting." -
ForegroundColor Red
        return
    }
    else
    {
        Write-Host "VERBOSE: Diskspd.exe found at" $FullPathToDiskspd
    }
```

```
}

$rdmaAdapter = Get-NetAdapter -IfIndex $IfIndex

if ($rdmaAdapter -eq $null)
{
    Write-Host "ERROR: The adapter with interface index $IfIndex not found" -
ForegroundColor Red
    return
}

$rdmaAdapterName = $rdmaAdapter.Name
$virtualAdapter = Get-VMNetworkAdapter -ManagementOS | where DeviceId -eq
$rdmaAdapter.DeviceID

if ($virtualAdapter -eq $null)
{
    $isRdmaAdapterVirtual = $false
    Write-Host "VERBOSE: The adapter $rdmaAdapterName is a physical adapter"
}
else
{
    $isRdmaAdapterVirtual = $true
    Write-Host "VERBOSE: The adapter $rdmaAdapterName is a virtual adapter"
}

$rdmaCapabilities = Get-NetAdapterRdma -InterfaceDescription
$rdmaAdapter.InterfaceDescription

if ($rdmaCapabilities -eq $null -or $rdmaCapabilities.Enabled -eq $false)
{
```

```
    Write-Host "ERROR: The adapter $rdmaAdapterName is not enabled for RDMA" -
ForegroundColor Red

    return

}


if ($rdmaCapabilities.MaxQueuePairCount -eq 0)

{

    Write-Host "ERROR: RDMA capabilities for adapter $rdmaAdapterName are not
valid : MaxQueuePairCount is 0" -ForegroundColor Red

    return

}


if ($rdmaCapabilities.MaxCompletionQueueCount -eq 0)

{

    Write-Host "ERROR: RDMA capabilities for adapter $rdmaAdapterName are not
valid : MaxCompletionQueueCount is 0" -ForegroundColor Red

    return

}


$smbClientNetworkInterfaces = Get-SmbClientNetworkInterface


if ($smbClientNetworkInterfaces -eq $null)

{

    Write-Host "ERROR: No network interfaces detected by SMB (Get-
SmbClientNetworkInterface)" -ForegroundColor Red

    return

}


$rdmaAdapterSmbClientNetworkInterface = $null

foreach ($smbClientNetworkInterface in $smbClientNetworkInterfaces)

{

    if ($smbClientNetworkInterface.InterfaceIndex -eq $IfIndex)
```

```
    {
        $rdmaAdapterSmbClientNetworkInterface = $smbClientNetworkInterface
    }
}

if ($rdmaAdapterSmbClientNetworkInterface -eq $null)
{
    Write-Host "ERROR: No network interfaces found by SMB for adapter
$rdmaAdapterName (Get-SmbClientNetworkInterface)" -ForegroundColor Red
    return
}

if ($rdmaAdapterSmbClientNetworkInterface.RdmaCapable -eq $false)
{
    Write-Host "ERROR: SMB did not detect adapter $rdmaAdapterName as RDMA
capable. Make sure the adapter is bound to TCP/IP and not to other protocol like
vmSwitch." -ForegroundColor Red
    return
}

$rdmaAdapters = $rdmaAdapter
if ($isRdmaAdapterVirtual -eq $true)
{
    Write-Host "VERBOSE: Retrieving vSwitch bound to the virtual adapter"
    $switchName = $virtualAdapter.SwitchName
    Write-Host "VERBOSE: Found vSwitch: $switchName"
    $vSwitch = Get-VMSwitch -Name $switchName
    $rdmaAdapters = Get-NetAdapter -InterfaceDescription
$vSwitch.NetAdapterInterfaceDescriptions
    $vSwitchAdapterMessage = "VERBOSE: Found the following physical adapter(s)
bound to vSwitch: "
    $index = 1
    foreach ($qosAdapter in $rdmaAdapters)
```

```
    {
        $qosAdapterName = $qosAdapter.Name
        $vSwitchAdapterMessage = $vSwitchAdapterMessage +
[string]$qosAdapterName
        if ($index -lt $rdmaAdapters.Length)
        {
                $vSwitchAdapterMessage = $vSwitchAdapterMessage + ", "
        }
        $index = $index + 1
    }
    Write-Host $vSwitchAdapterMessage
}



if ($IsRoCE -eq $true)
{
    Write-Host "VERBOSE: Underlying adapter is RoCE. Checking if QoS/DCB/PFC is
configured on each physical adapter(s)"
    foreach ($qosAdapter in $rdmaAdapters)
    {
        $qosAdapterName = $qosAdapter.Name
        $qos = Get-NetAdapterQos -Name $qosAdapterName
        if ($qos.Enabled -eq $false)
        {
            Write-Host "ERROR: QoS is not enabled for adapter $qosAdapterName" -
ForegroundColor Red
            return
        }

        if ($qos.OperationalFlowControl -eq "All Priorities Disabled")
        {
```

```
            Write-Host "ERROR: Flow control is not enabled for adapter
$qosAdapterName" -ForegroundColor Red
            return
        }
    }
    Write-Host "VERBOSE: QoS/DCB/PFC configuration is correct."
}

Write-Host "VERBOSE: RDMA configuration is correct."

if ($RemoteIpAddress -ne '')
{
    Write-Host "VERBOSE: Checking if remote IP address, $RemoteIpAddress, is
reachable."
    $canPing = Test-Connection $RemoteIpAddress -Quiet
    if ($canPing -eq $false)
    {
        Write-Host "ERROR: Cannot reach remote IP $RemoteIpAddress" -
ForegroundColor Red
        return
    }
    else
    {
        Write-Host "VERBOSE: Remote IP $RemoteIpAddress is reachable."
    }
}

if ($RemoteIpAddress -eq '')
{
    Write-Host "VERBOSE: Remote IP address was not provided. If RDMA does not
work, make sure that remote IP address is reachable."
```

```
}
else
{
    Write-Host "VERBOSE: Disabling RDMA on adapters that are not part of this
test. RDMA will be enabled on them later."
    $adapters = Get-NetAdapterRdma
    $InstanceIds = $rdmaAdapters.InstanceID;

    $adaptersToEnableRdma = @()
    foreach ($adapter in $adapters)
    {
        if ($adapter.Enabled -eq $true)
        {
            if (($adapter.InstanceID -notin $InstanceIds) -And
($adapter.InstanceID -ne $rdmaAdapter.InstanceID))
            {
                $adaptersToEnableRdma += $adapter
                Disable-NetAdapterRdma -Name $adapter.Name
            }
        }
    }

    Write-Host "VERBOSE: Testing RDMA traffic now for. Traffic will be sent in a
parallel job. Job details:"

    $ScriptBlock = {
        param($RemoteIpAddress, $PathToDiskspd)
        cd $PathToDiskspd
        .\diskspd.exe -b4K -c10G -t4 -o16 -d100000 -L -Sr -d30
\\$RemoteIpAddress\C$\testfile.dat
      }
```

```
    $thisJob = Start-Job $ScriptBlock -ArgumentList
$RemoteIpAddress,$PathToDiskspd


    $RdmaTrafficDetected = $false


    # Check Perfmon counters while the job is running
    While ((Get-Job -id $($thisJob).Id).state -eq "Running")
    {
        $written = Get-Counter -Counter "\SMB Direct Connection(_Total)\Bytes
RDMA Written/sec" -ErrorAction Ignore
        $sent = Get-Counter -Counter "\SMB Direct Connection(_Total)\Bytes
Sent/sec" -ErrorAction Ignore
        if ($written -ne $null)
        {
            $RdmaWriteBytesPerSecond = [uint64]($written.Readings.split(":")[1])
            if ($RdmaWriteBytesPerSecond -gt 0)
            {
                 $RdmaTrafficDetected = $true


            }
            Write-Host "VERBOSE:" $RdmaWriteBytesPerSecond "RDMA bytes written
per second"
        }
        if ($sent -ne $null)
        {
             $RdmaWriteBytesPerSecond = [uint64]($sent.Readings.split(":")[1])
            if ($RdmaWriteBytesPerSecond -gt 0)
            {
                 $RdmaTrafficDetected = $true
            }
            Write-Host "VERBOSE:" $RdmaWriteBytesPerSecond "RDMA bytes sent per
second"
        }
```

```
    }

    del \\$RemoteIpAddress\C$\testfile.dat

    Write-Host "VERBOSE: Enabling RDMA on adapters that are not part of this
test. RDMA was disabled on them prior to sending RDMA traffic."
    foreach ($adapter in $adaptersToEnableRdma)
    {
        Enable-NetAdapterRdma -Name $adapter.Name
    }

    if ($RdmaTrafficDetected)
    {
        Write-Host "VERBOSE: RDMA traffic test SUCCESSFUL: RDMA traffic was sent
to" $RemoteIpAddress -ForegroundColor Green
    }
    else
    {
        Write-Host "VERBOSE: RDMA traffic test FAILED: Please check physical
switch port configuration for Priorty Flow Control." -ForegroundColor Yellow
    }

}
```

Chapter 40

Storage Spaces Direct Network Reporting HTML Script for Mellanox Adapters via PowerShell

By: Dave Kawula – MVP

Hey Storage Spaces Direct fans, I know we have had a lot of chatter going on regarding Mellanox's recent bad firmware release. As I banged my head up against the wall I discovered that Mellanox actually provides some really nice PowerShell Cmdlets with their WinOF drivers.

When looking into them I figured why not build out a nice little reporting script that would grab the Mellanox NIC Configs from my Storage Spaces Direct Environment.

This will help me discover driver, firmware, and settings drift quite easily.

Here is a list of all the Mellanox PowerShell Commands currently available:

```
#Powershell SET commands Sets

Set-MlnxDriverCoreSetting
Set-MlnxPCIDevicePortTypeSetting
Set-MlnxPCIDeviceSriovSetting
```

```
#Powershell GET commands Sets

Get-MlnxDriver
Get-MlnxFirmware
Get-MlnxIBPort
Get-MlnxNetAdapter
Get-MlnxPCIDevice
Get-MlnxSoftware

#Get-MlnxDriver Command Set

Get-MlnxDriverCapabilities
Get-MlnxDriverCoreCapabilities
Get-MlnxDriverCoreSetting
Get-MlnxDriverService
Get-MlnxDriverSetting


Get-MlnxDriverCapabilities |FL
Get-MlnxDriverCoreCapabilities |FL
Get-MlnxDriverCoreSetting | FL
Get-MlnxFirmwareIdentity | FL
Get-MlnxIBPort
Get-MlnxIBPortCounters | FL
Get-MlnxNetAdapter | FL
Get-MlnxNetAdapterEcnSetting | FL
Get-MlnxNetAdapterFlowControlSetting | FL
Get-MlnxNetAdapterRoceSetting | FL
Get-MlnxNetAdapterSetting |FL
```

```
Get-MLNXPCIDevice | fl
Get-MLNXPCIDeviceCapabilities | fl
Get-MlnxPCIDevicePortTypeSetting |fl
Get-MlnxPCIDeviceSetting | fl
Get-MlnxSoftwareIdentity
```

What I did was take this and build them into a pretty little HTML Report for you using PowerShell.

```
$Header = @"
<style>
TABLE {border-width: 1px; border-style: solid; border-color: black; border-collapse: collapse;}
TH {border-width: 1px; padding: 3px; border-style: solid; border-color: black; background-color: #6495ED;}
TD {border-width: 1px; padding: 3px; border-style: solid; border-color: black;}
</style>
"@

$servers = @('S2DNODE1','S2DNODE2')
$resultComputerInfo = Invoke-Command -ComputerName $servers -ScriptBlock { Get-ComputerInfo | Select-Object -Property CSDNSHostName,WindowsEditionId,OSServerLevel,OSUptime,OsFreePhysicalMemory,CSModel,CSManufacturer,CSNumberOfLogicalProcessors,CSNumberofProcessors,HyperVisorPresent }

$resultMLNXPCIDevice = Invoke-Command -ComputerName $servers -ScriptBlock { Get-MLNXPCIDevice | Select-Object -Property Systemname,Caption,Description,DeviceID,LastErrorCode,DriverVersion,FirmwareVersion }
```

```
$resultMlnxPCIDeviceSetting = Invoke-Command -ComputerName $servers -ScriptBlock
{ Get-MlnxPCIDeviceSetting | Select-Object -Property
Systemname,Caption,Description,InstanceID }


$resultMLNXPCIDeviceCapabilities = Invoke-Command -ComputerName $servers -
ScriptBlock { Get-MLNXPCIDeviceCapabilities | Select-Object -Property
Systemname,Caption,Description,PortOneAutoSense,PortOneDefault,PortOneAutoSenseA
llowed,PortOneEth,PorttwoIb,PortTwoAutoSenseCap,PortTwoDefault,PortTwoDoSenseAll
owed,PortTwoEth }


$resultMlnxNetAdapter = Invoke-Command -ComputerName $servers -ScriptBlock {
Get-MlnxNetAdapter | Select-Object -Property
Systemname,Caption,Description,Name,ErrorDescription,MaxSpeed,MaxTransmissionUni
t,AutoSense,FullDuplex,LinkTechnology,PortNumber,DroplessMode }


$resultMlnxNetAdapterRoceSetting = Invoke-Command -ComputerName $servers -
ScriptBlock { Get-MlnxNetAdapterRoceSetting | Select-Object -Property
Systemname,Caption,Description,InterfaceDescription,PortNumber,RoceMode,Enabled
}


$resultMlnxIBPort = Invoke-Command -ComputerName $servers -ScriptBlock { Get-
MlnxIBPort | Select-Object -Property
Systemname,Caption,Description,MaxSpeed,PortType,Speed,ActiveMaximumTransmission
Unit,PortNumberSupportedMaximumTransmissionUnit,MaxMsgSize,MaxVls,NumGids,NumPke
ys,Transport }


$resultMlnxIBPortCounters = Invoke-Command -ComputerName $servers -ScriptBlock {
Get-MlnxIBPortCounters | Select-Object -Property
Systemname,Caption,Description,StatisticTime,BytesReceived,BytesTransmitted,Pack
etsReceived,PacketsTransmitted,ExcessiveBufferOverflows,LinkDownCounter,LinterEr
rorRecoveryCounter,PortRcvErrors }


$resultMlnxFirmwareIdentity = Invoke-Command -ComputerName $servers -ScriptBlock
{ Get-MlnxFirmwareIdentity | Select-Object -Property
Caption,Description,Name,Manufacturer,VersionString  }



ConvertTo-Html -Body "<H1>CheckyourLogs.Net Mellanox Storage Spaces Direct S2D
Node Configuration Report </H1><H1> S2D System Information </H3>
$($resultComputerInfo | Convertto-Html -Property * -Fragment) <H1> Mellanox
Software  </H1> $($resultMLNXPCIDevice | Convertto-Html -Property * -Fragment))
```

```
<h1>Mellanox PCI Device Settings</h1> $($resultMLNXPCIDeviceDeviceSetting |
Convertto-Html -Property * -Fragment) <H1> Mellanox Device Capabilities </H1>
$($resultMLNXPCIDeviceCapabilities | Convertto-Html -Property * -Fragment) <H1>
Mellanox NetAdapter Info </H1>$($resultMlnxNetAdapter | Convertto-Html -Property
* -Fragment) <H1> Mellanox ROCE Settings </H1>
$($resultMlnxNetAdapterRoceSetting | Convertto-Html -Property * -Fragment) <H1>
Mellanox IB Port Configuration </H1> $($resultMlnxIBPort | Convertto-Html -
Property * -Fragment) <H1> Mellanox IB Port Counters
</H1>$($resultMlnxIBPortCounters | Convertto-Html -Property * -Fragment) <H1>
Mellanox Adapter Firmware </H1> $($resultMlnxFirmwareIdentity | Convertto-Html -
Property * -Fragment)" -Title "Mellanox Adapter Configuraiton" -Head $Header
|Out-File mellanoxreport.html
```

Mellanox Adapter Firmware

Caption	Description	Name	Manufacturer	VersionString	PSComputerNam
MLNX_FirmwareIdentity 'Firmware for device 4103 with PSID MT_1200111023'	Firmware for device 4103 with PSID MT_1200111023	MLNX VPI Adapter MT1652K02108 firmware	Mellanox Technologies	2.40.7000	S2DNODE2
MLNX_FirmwareIdentity 'Firmware for device 4103 with PSID MT_1200111023'	Firmware for device 4103 with PSID MT_1200111023	MLNX VPI Adapter MT1652K02114 firmware	Mellanox Technologies	2.40.5032	S2DNODE1

In the screenshot above we can see that we have a mismatched firmware. Good thing we had this little script to help us figure that out 😊

Dave

Chapter 41

Using PowerShell and DSC to build out an RDSH Farm from Scratch

By: Dave Kawula – MVP

So, I have a new project coming up where I will be required to manage, maintain, and support an RDS Deployment for a local engineering firm.

And after working with some of the brightest PowerShell experts in the world on the Master PowerShell Tricks series I decided to cut ties to the GUI and build it 100 % using PowerShell.

The requirements for me to test this are actually a bit complicated because I wanted to have a test lab to play with.

Luckily, I had already build my BigDemo PowerShell Script that included all the functions I would need to get started. You can grab a copy for yourself at https://www.github.com/dkawula. It was also features in Master PowerShell Tricks V2 and Master Storage Spaces Direct.

Let's commence the work at around 3:00 PM I started modifying the code in my existing script.

If you recall I use this same script to build out my Storage Spaces Direct Farms.

Now I have a couple of functions that I use to build the base VM's from the Base Virtual Disks and then do their post configurations.

```
function Invoke-DemoVMPrep
{
    param
    (
        [string] $VMName,
        [string] $GuestOSName,
        [switch] $FullServer
    )

    Write-Log $VMName 'Removing old VM'
    get-vm $VMName -ErrorAction SilentlyContinue |
    stop-vm -TurnOff -Force -Passthru |
    remove-vm -Force
    Clear-File "$($VMPath)\$($GuestOSName).vhdx"
```

```
    Write-Log $VMName 'Creating new differencing disk'

    if ($FullServer)

    {

        $null = New-VHD -Path "$($VMPath)\$($GuestOSName).vhdx" -ParentPath
"$($BaseVHDPath)\VMServerBase.vhdx" -Differencing

    }


    else

    {

        $null = New-VHD -Path "$($VMPath)\$($GuestOSName).vhdx" -ParentPath
"$($BaseVHDPath)\VMServerBaseCore.vhdx" -Differencing

    }


    Write-Log $VMName 'Creating virtual machine'

    new-vm -Name $VMName -MemoryStartupBytes 16GB -SwitchName $virtualSwitchName
`

    -Generation 2 -Path "$($VMPath)\" | Set-VM -ProcessorCount 2


    Set-VMFirmware -VMName $VMName -SecureBootTemplate
MicrosoftUEFICertificateAuthority

    Set-VMFirmware -Vmname $VMName -EnableSecureBoot off

    Add-VMHardDiskDrive -VMName $VMName -Path "$($VMPath)\$($GuestOSName).vhdx"
-ControllerType SCSI

    Write-Log $VMName 'Starting virtual machine'

    Enable-VMIntegrationService -Name 'Guest Service Interface' -VMName $VMName

    start-vm $VMName

}


function Create-DemoVM

{

    param

    (
```

```
        [string] $VMName,
        [string] $GuestOSName,
        [string] $IPNumber = '0'
    )

    Wait-PSDirect $VMName -cred $localCred

    Invoke-Command -VMName $VMName -Credential $localCred {
        param($IPNumber, $GuestOSName,  $VMName, $domainName, $Subnet)
        if ($IPNumber -ne '0')
        {
            Write-Output -InputObject "[$($VMName)]:: Setting IP Address to
$($Subnet)$($IPNumber)"
            $null = New-NetIPAddress -IPAddress "$($Subnet)$($IPNumber)" -
InterfaceAlias 'Ethernet' -PrefixLength 24
            Write-Output -InputObject "[$($VMName)]:: Setting DNS Address"
            Get-DnsClientServerAddress | ForEach-Object -Process {
                Set-DnsClientServerAddress -InterfaceIndex $_.InterfaceIndex -
ServerAddresses "$($Subnet)1"
            }
        }
        Write-Output -InputObject "[$($VMName)]:: Renaming OS to
`"$($GuestOSName)`""
        Rename-Computer -NewName $GuestOSName
        Write-Output -InputObject "[$($VMName)]:: Configuring WSMAN Trusted
hosts"
        Set-Item -Path WSMan:\localhost\Client\TrustedHosts -Value
"*.$($domainName)" -Force
        Set-Item WSMan:\localhost\client\trustedhosts "$($Subnet)*" -Force -
concatenate
        Enable-WSManCredSSP -Role Client -DelegateComputer "*.$($domainName)" -
Force
    } -ArgumentList $IPNumber, $GuestOSName, $VMName, $domainName, $Subnet
```

```
    Restart-DemoVM $VMName

    Wait-PSDirect $VMName -cred $localCred
}
```

After the Servers are build using Invoke-DemoVMPrep we use the Create-DemoVM to do their final configs… here is what it looks like inside the script.

Now in this example I build a Domain Controller, MGMT Server, and DHCP Server, and the basic VM's build for the RDS Farm.

```
Invoke-DemoVMPrep 'DHCP1-RDS' 'DHCP1-RDS' -FullServer
Invoke-DemoVMPrep 'MGMT1-RDS' 'MGMT1-RDS' -FullServer
Invoke-DemoVMPrep 'RDSH01-RDS' 'RDSH01-RDS' -FullServer
Invoke-DemoVMPrep 'RDSH02-RDS' 'RDSH02-RDS' -FullServer
Invoke-DemoVMPrep 'RDGW01-RDS' 'RDGW01-RDS' -FullServer
Invoke-DemoVMPrep 'RDAPP01-RDS' 'RDAPP01-RDS' -FullServer
Invoke-DemoVMPrep 'DC1-RDS' 'DC1-RDS' -FullServer


$VMName = 'DC1-RDS'
$GuestOSName = 'DC1-RDS'
$IPNumber = '1'

Create-DemoVM $VMName $GuestOSName $IPNumber

Invoke-Command -VMName $VMName -Credential $localCred {
    param($VMName, $domainName, $domainAdminPassword)

    Write-Output -InputObject "[$($VMName)]:: Installing AD"
    $null = Install-WindowsFeature AD-Domain-Services -IncludeManagementTools
```

```
    Write-Output -InputObject "[$($VMName)]:: Enabling Active Directory and
promoting to domain controller"

    Install-ADDSForest -DomainName $domainName -InstallDNS -NoDNSonNetwork -
NoRebootOnCompletion `

    -SafeModeAdministratorPassword (ConvertTo-SecureString -String
$domainAdminPassword -AsPlainText -Force) -confirm:$false

} -ArgumentList $VMName, $domainName, $domainAdminPassword


Restart-DemoVM $VMName




$VMName = 'DHCP1-RDS'

$GuestOSName = 'DHCP1-RDS'

$IPNumber = '3'


Create-DemoVM $VMName $GuestOSName $IPNumber


Invoke-Command -VMName $VMName -Credential $localCred {

    param($VMName, $domainCred, $domainName)

    Write-Output -InputObject "[$($VMName)]:: Installing DHCP"

    $null = Install-WindowsFeature DHCP -IncludeManagementTools

    Write-Output -InputObject "[$($VMName)]:: Joining domain as
`"$($env:computername)`""

    while (!(Test-Connection -ComputerName $domainName -BufferSize 16 -Count 1 -
Quiet -ea SilentlyContinue))

    {

        Start-Sleep -Seconds 1

    }

    do

    {

        Add-Computer -DomainName $domainName -Credential $domainCred -ea
SilentlyContinue

    }
```

```
    until ($?)
} -ArgumentList $VMName, $domainCred, $domainName

Restart-DemoVM $VMName
Wait-PSDirect $VMName -cred $domainCred

Invoke-Command -VMName $VMName -Credential $domainCred {
    param($VMName, $domainName, $Subnet, $IPNumber)

    Write-Output -InputObject "[$($VMName)]:: Waiting for name resolution"

    while ((Test-NetConnection -ComputerName $domainName).PingSucceeded -eq
$false)
    {
        Start-Sleep -Seconds 1
    }

    Write-Output -InputObject "[$($VMName)]:: Configuring DHCP Server"
    Set-DhcpServerv4Binding -BindingState $true -InterfaceAlias Ethernet
    Add-DhcpServerv4Scope -Name 'IPv4 Network' -StartRange "$($Subnet)10" -
EndRange "$($Subnet)200" -SubnetMask 255.255.255.0
    Set-DhcpServerv4OptionValue -OptionId 6 -value "$($Subnet)1"
    Add-DhcpServerInDC -DnsName "$($env:computername).$($domainName)"
    foreach($i in 1..99)
    {
        $mac = '00-b5-5d-fe-f6-' + ($i % 100).ToString('00')
        $ip = $Subnet + '1' + ($i % 100).ToString('00')
        $desc = 'Container ' + $i.ToString()
        $scopeID = $Subnet + '0'
        Add-DhcpServerv4Reservation -IPAddress $ip -ClientId $mac -Description
$desc -ScopeId $scopeID
    }
```

```
} -ArgumentList $VMName, $domainName, $Subnet, $IPNumber

Restart-DemoVM $VMName
```

Now that I had my configurations started I finished up by running Create-DemoVM on the RDS Farm instances which basically just joined them to the domain and restarted them.

```
$VMName = 'MGMT1-RDS'
$GuestOSName = 'MGMT1-RDS'

Create-DemoVM $VMName $GuestOSName

Invoke-Command -VMName $VMName -Credential $localCred {
    param($VMName, $domainCred, $domainName)
    Write-Output -InputObject "[$($VMName)]:: Management tools"
    $null = Install-WindowsFeature RSAT-Clustering, RSAT-Hyper-V-Tools
    Write-Output -InputObject "[$($VMName)]:: Joining domain as
`"$($env:computername)`""
    while (!(Test-Connection -ComputerName $domainName -BufferSize 16 -Count 1 -
Quiet -ea SilentlyContinue))
    {
        Start-Sleep -Seconds 1
    }
    do
    {
        Add-Computer -DomainName $domainName -Credential $domainCred -ea
SilentlyContinue
    }
    until ($?)
} -ArgumentList $VMName, $domainCred, $domainName
```

```
Restart-DemoVM $VMName

$VMName = 'RDSH01-RDS'
$GuestOSName = 'RDSH01-RDS'

Create-DemoVM $VMName $GuestOSName

Invoke-Command -VMName $VMName -Credential $localCred {
    param($VMName, $domainCred, $domainName)
    Write-Output -InputObject "[$($VMName)]:: Management tools"
   # $null = Install-WindowsFeature RSAT-Clustering, RSAT-Hyper-V-Tools
    Write-Output -InputObject "[$($VMName)]:: Joining domain as
`"$($env:computername)`""
    while (!(Test-Connection -ComputerName $domainName -BufferSize 16 -Count 1 -
Quiet -ea SilentlyContinue))
    {
        Start-Sleep -Seconds 1
    }
    do
    {
        Add-Computer -DomainName $domainName -Credential $domainCred -ea
SilentlyContinue
    }
    until ($?)
} -ArgumentList $VMName, $domainCred, $domainName

Restart-DemoVM $VMName

$VMName = 'RDSH02-RDS'
$GuestOSName = 'RDSH02-RDS'
```

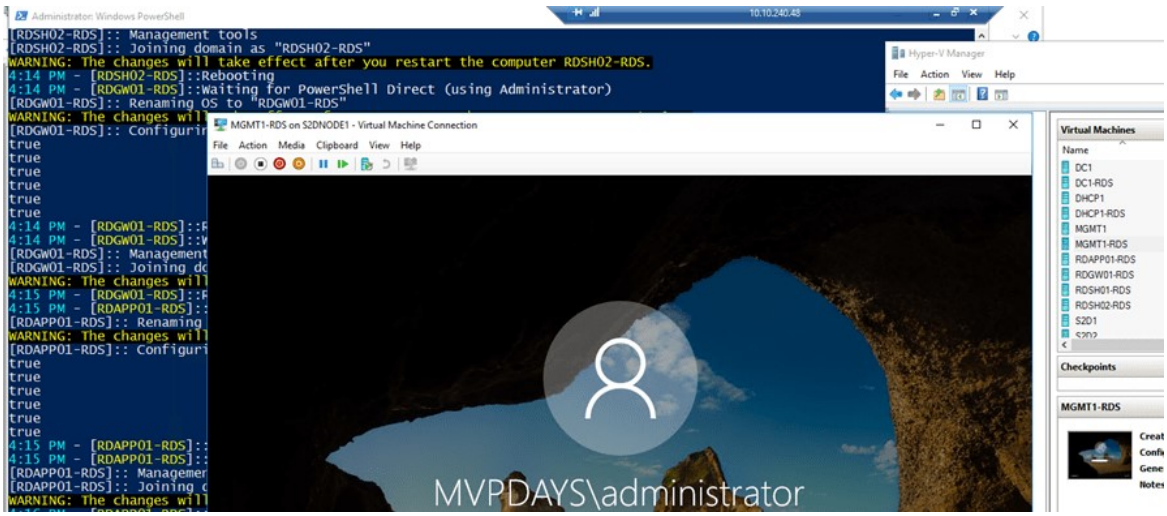
```
Create-DemoVM $VMName $GuestOSName

Invoke-Command -VMName $VMName -Credential $localCred {
    param($VMName, $domainCred, $domainName)
    Write-Output -InputObject "[$($VMName)]:: Management tools"
    #$null = Install-WindowsFeature RSAT-Clustering, RSAT-Hyper-V-Tools
    Write-Output -InputObject "[$($VMName)]:: Joining domain as
`"$($env:computername)`""
    while (!(Test-Connection -ComputerName $domainName -BufferSize 16 -Count 1 -
Quiet -ea SilentlyContinue))
    {
        Start-Sleep -Seconds 1
    }
    do
    {
        Add-Computer -DomainName $domainName -Credential $domainCred -ea
SilentlyContinue
    }
    until ($?)
} -ArgumentList $VMName, $domainCred, $domainName

Restart-DemoVM $VMName

$VMName = 'RDGW01-RDS'
$GuestOSName = 'RDGW01-RDS'

Create-DemoVM $VMName $GuestOSName

Invoke-Command -VMName $VMName -Credential $localCred {
    param($VMName, $domainCred, $domainName)
    Write-Output -InputObject "[$($VMName)]:: Management tools"
```

```
    #$null = Install-WindowsFeature RSAT-Clustering, RSAT-Hyper-V-Tools
    Write-Output -InputObject "[$($VMName)]:: Joining domain as
`"$($env:computername)`""
    while (!(Test-Connection -ComputerName $domainName -BufferSize 16 -Count 1 -
Quiet -ea SilentlyContinue))
    {
        Start-Sleep -Seconds 1
    }
    do
    {
        Add-Computer -DomainName $domainName -Credential $domainCred -ea
SilentlyContinue
    }
    until ($?)
} -ArgumentList $VMName, $domainCred, $domainName

Restart-DemoVM $VMName

$VMName = 'RDAPP01-RDS'
$GuestOSName = 'RDAPP01-RDS'

Create-DemoVM $VMName $GuestOSName

Invoke-Command -VMName $VMName -Credential $localCred {
    param($VMName, $domainCred, $domainName)
    Write-Output -InputObject "[$($VMName)]:: Management tools"
    $null = Install-WindowsFeature RSAT-Clustering, RSAT-Hyper-V-Tools
    Write-Output -InputObject "[$($VMName)]:: Joining domain as
`"$($env:computername)`""
    while (!(Test-Connection -ComputerName $domainName -BufferSize 16 -Count 1 -
Quiet -ea SilentlyContinue))
    {
        Start-Sleep -Seconds 1
```

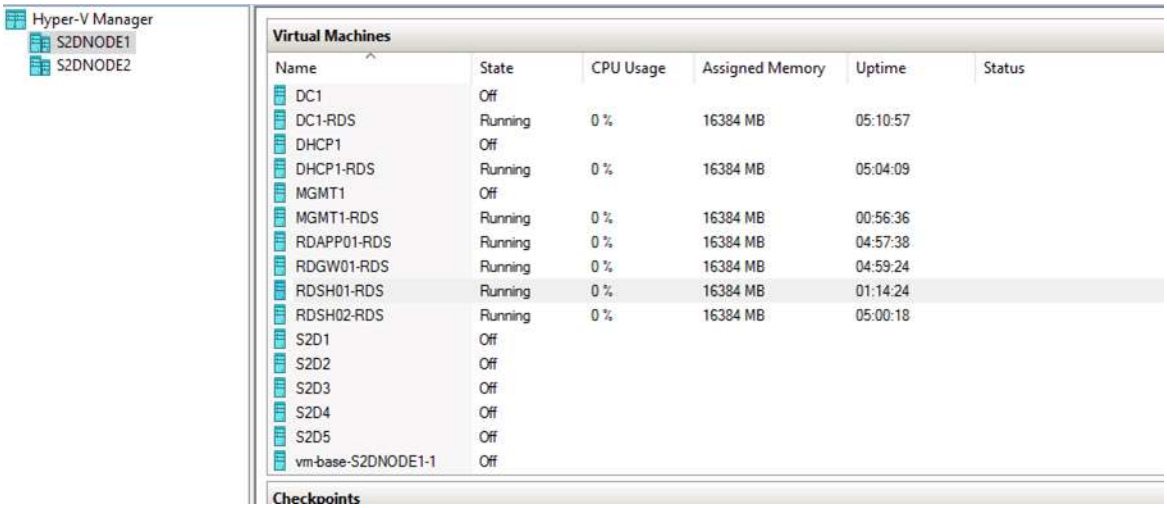
```
    }
    do
    {
        Add-Computer -DomainName $domainName -Credential $domainCred -ea
SilentlyContinue
    }
    until ($?)
} -ArgumentList $VMName, $domainCred, $domainName

Restart-DemoVM $VMName
```

The coolest part about all of this is that I am running all of this infrastructure on my 2-node Storage Spaces Direct All Flash Array and it only took 20 minutes to build this start to finish.

Here is the Script building finished product looked like this: This final run was done at around 4:14 PM



Here are the VM's Built in Hyper-V



Name	State	CPU Usage	Assigned Memory	Uptime	Status
DC1	Off				
DC1-RDS	Running	0 %	16384 MB	05:10:57	
DHCP1	Off				
DHCP1-RDS	Running	0 %	16384 MB	05:04:09	
MGMT1	Off				
MGMT1-RDS	Running	0 %	16384 MB	00:56:36	
RDAPP01-RDS	Running	0 %	16384 MB	04:57:38	
RDGW01-RDS	Running	0 %	16384 MB	04:59:24	
RDSH01-RDS	Running	0 %	16384 MB	01:14:24	
RDSH02-RDS	Running	0 %	16384 MB	05:00:18	
S2D1	Off				
S2D2	Off				
S2D3	Off				
S2D4	Off				
S2D5	Off				
vm-base-S2DNODE1-1	Off				

Now the coolest part of what I wanted to do was to automate the build of the RDS Farm with PowerShell DSC.

To accomplish this I used a PSGallery Item called xRemoteDesktopSessionHost v.1.4.0.0 which can be found here:

https://www.powershellgallery.com/packages/xRemoteDesktopSessionHost/1.4.0.0

Now with the help of Will Anderson one of the amazing Honorary Scripting Guys at Microsoft I was able to install this DSCResource without having to do much other than execute this one line of PowerShell on my

target machine:

```
Find-Module xRemoteDesktopSessionHost | Install-Module
```

Once done I had the PowerShell DSC module that would be required for me to proceed.

For tonight's testing, I decided to do a single server configuration to see how hard it would be.

Here is the DSC Configuration I used to build out my base configuration for testing:

```
param (
[string]$brokerFQDN,
[string]$webFQDN,
[string]$collectionName,
[string]$collectionDescription
)

$localhost = [System.Net.Dns]::GetHostByName((hostname)).HostName

if (!$collectionName) {$collectionName = "DK Collection"}
if (!$collectionDescription) {$collectionDescription = "Remote Desktop instance
for accessing an isolated network environment."}

Configuration RemoteDesktopSessionHost
{
    param
    (

        # Connection Broker Name
        [Parameter(Mandatory)]
        [String]$collectionName,

        # Connection Broker Description
```

```
    [Parameter(Mandatory)]
    [String]$collectionDescription,

    # Connection Broker Node Name
    [String]$connectionBroker,

    # Web Access Node Name
    [String]$webAccessServer
)
Import-DscResource -Module xRemoteDesktopSessionHost
if (!$connectionBroker) {$connectionBroker = $localhost}
if (!$connectionWebAccessServer) {$webAccessServer = $localhost}

Node "localhost"
{

    LocalConfigurationManager
    {
        RebootNodeIfNeeded = $true
    }

    WindowsFeature Remote-Desktop-Services
    {
        Ensure = "Present"
        Name = "Remote-Desktop-Services"
    }

    WindowsFeature RDS-RD-Server
    {
        Ensure = "Present"
```

```
        Name = "RDS-RD-Server"
    }

    WindowsFeature Desktop-Experience
    {
        Ensure = "Present"
        Name = "Desktop-Experience"
    }

    WindowsFeature RSAT-RDS-Tools
    {
        Ensure = "Present"
        Name = "RSAT-RDS-Tools"
        IncludeAllSubFeature = $true
    }

    if ($localhost -eq $connectionBroker) {
        WindowsFeature RDS-Connection-Broker
        {
            Ensure = "Present"
            Name = "RDS-Connection-Broker"
        }
    }

    if ($localhost -eq $webAccessServer) {
        WindowsFeature RDS-Web-Access
        {
            Ensure = "Present"
            Name = "RDS-Web-Access"
        }
```

```
        }

        WindowsFeature RDS-Licensing
        {
            Ensure = "Present"
            Name = "RDS-Licensing"
        }

        xRDSessionDeployment Deployment
        {
            SessionHost = $localhost
            ConnectionBroker = if ($ConnectionBroker) {$ConnectionBroker} else
{$localhost}
            WebAccessServer = if ($WebAccessServer) {$WebAccessServer} else
{$localhost}
            DependsOn = "[WindowsFeature]Remote-Desktop-Services",
"[WindowsFeature]RDS-RD-Server"
        }

        xRDSessionCollection Collection
        {
            CollectionName = $collectionName
            CollectionDescription = $collectionDescription
            SessionHost = $localhost
            ConnectionBroker = if ($ConnectionBroker) {$ConnectionBroker} else
{$localhost}
            DependsOn = "[xRDSessionDeployment]Deployment"
        }
        xRDSessionCollectionConfiguration CollectionConfiguration
        {
        CollectionName = $collectionName
        CollectionDescription = $collectionDescription
```

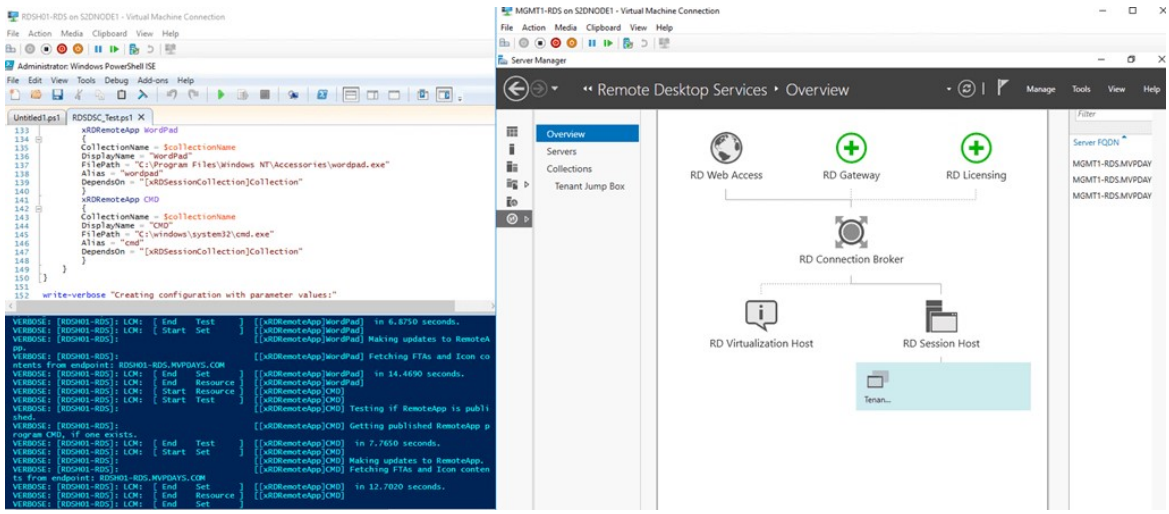
```
        ConnectionBroker = if ($ConnectionBroker) {$ConnectionBroker} else
{$localhost}
        TemporaryFoldersDeletedOnExit = $false
        SecurityLayer = "SSL"
        DependsOn = "[xRDSessionCollection]Collection"
        }
        xRDRemoteApp Calc
        {
        CollectionName = $collectionName
        DisplayName = "Calculator"
        FilePath = "C:\Windows\System32\calc.exe"
        Alias = "calc"
        DependsOn = "[xRDSessionCollection]Collection"
        }
        xRDRemoteApp Mstsc
        {
        CollectionName = $collectionName
        DisplayName = "Remote Desktop"
        FilePath = "C:\Windows\System32\mstsc.exe"
        Alias = "mstsc"
        DependsOn = "[xRDSessionCollection]Collection"
        }

        xRDRemoteApp WordPad
        {
        CollectionName = $collectionName
        DisplayName = "WordPad"
        FilePath = "C:\Program Files\Windows NT\Accessories\wordpad.exe"
        Alias = "wordpad"
        DependsOn = "[xRDSessionCollection]Collection"
        }
```

```
        xRDRemoteApp CMD
        {
        CollectionName = $collectionName
        DisplayName = "CMD"
        FilePath = "C:\windows\system32\cmd.exe"
        Alias = "cmd"
        DependsOn = "[xRDSessionCollection]Collection"
        }
    }
}

write-verbose "Creating configuration with parameter values:"
write-verbose "Collection Name: $collectionName"
write-verbose "Collection Description: $collectionDescription"
write-verbose "Connection Broker: $brokerFQDN"
write-verbose "Web Access Server: $webFQDN"

RemoteDesktopSessionHost -collectionName $collectionName -collectionDescription
$collectionDescription -connectionBroker $brokerFQDN -webAccessServer $webFQDN -
OutputPath .\RDSDSC\

Set-DscLocalConfigurationManager -verbose -path .\RDSDSC\

Start-DscConfiguration -wait -force -verbose -path .\RDSDSC\
```

Here was a snip of the script in action building the single node RDS Test Server:

```
program Calculator, if one exists.
VERBOSE: [RDSH01-RDS]: LCM:  [ End    Test     ]  [[xRDRemoteApp]Calc]  in 6.1880 seconds.
VERBOSE: [RDSH01-RDS]: LCM:  [ Start  Set      ]  [[xRDRemoteApp]Calc]
VERBOSE: [RDSH01-RDS]:                            [[xRDRemoteApp]Calc] Making updates to RemoteApp.
VERBOSE: [RDSH01-RDS]: LCM:  [ End    Set      ]  [[xRDRemoteApp]Calc]  in 8.3590 seconds.
VERBOSE: [RDSH01-RDS]: LCM:  [ End    Resource ]  [[xRDRemoteApp]Calc]
VERBOSE: [RDSH01-RDS]: LCM:  [ Start  Resource ]  [[xRDRemoteApp]Mstsc]
VERBOSE: [RDSH01-RDS]: LCM:  [ Start  Test     ]  [[xRDRemoteApp]Mstsc]
VERBOSE: [RDSH01-RDS]:                            [[xRDRemoteApp]Mstsc] Testing if RemoteApp is pub
lished.
VERBOSE: [RDSH01-RDS]:                            [[xRDRemoteApp]Mstsc] Getting published RemoteApp
 program Remote Desktop, if one exists.
VERBOSE: [RDSH01-RDS]: LCM:  [ End    Test     ]  [[xRDRemoteApp]Mstsc]  in 7.0620 seconds.
VERBOSE: [RDSH01-RDS]: LCM:  [ Start  Set      ]  [[xRDRemoteApp]Mstsc]
VERBOSE: [RDSH01-RDS]:                            [[xRDRemoteApp]Mstsc] Making updates to RemoteApp
.
VERBOSE: [RDSH01-RDS]:                            [[xRDRemoteApp]Mstsc] Fetching FTAs and Icon cont
ents from endpoint: RDSH01-RDS.MVPDAYS.COM
VERBOSE: [RDSH01-RDS]: LCM:  [ End    Set      ]  [[xRDRemoteApp]Mstsc]  in 12.0460 seconds.
VERBOSE: [RDSH01-RDS]: LCM:  [ End    Resource ]  [[xRDRemoteApp]Mstsc]
VERBOSE: [RDSH01-RDS]: LCM:  [ Start  Resource ]  [[xRDRemoteApp]WordPad]
VERBOSE: [RDSH01-RDS]: LCM:  [ Start  Test     ]  [[xRDRemoteApp]WordPad]
VERBOSE: [RDSH01-RDS]:                            [[xRDRemoteApp]WordPad] Testing if RemoteApp is p
```

Here was a screenshot of the completely installed farm.



Here is a Screenshot of the view from the client's perspective

I did do some testing by removing some of the applications and then re-running the DSC Configuration and as expected the just got re-published.

From myself and all the authors that are part of this series we want to thank you for taking the time for reading it. All of us are looking forward to seeing you in Master PowerShell Tricks V4.

Thanks from ,

The MVP Days Publishing Authors, Editors, and Volunteers.

Chapter 42

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.

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
- Mick Pletcher: @Mick_Pletcher
- Will Anderson: @GamerLivingWill
- Cary Sun: @SifuSun