# 505.1

# PowerShell Automation and Security

SANS

Copyright © 2016, The SANS Institute. All rights reserved. The entire contents of this publication are the property of the SANS Institute.

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND THE SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With the CLA, the SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by the SANS Institute to the User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, YOU AGREE THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO THE SANS INSTITUTE, AND THAT THE SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND), SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If you do not agree, you may return the Courseware to the SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of the SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of the SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

SEC505.1 | Securing Windows and PowerShell Automation

SANS

# PowerShell Automation and Security

© Jason Fossen, Enclave Consulting LLC | All Rights Reserved | Version # B02_01

## Document Legalities

All reasonable and good faith efforts have been exerted to verify that the information in this document is accurate and up-to-date. However, new software releases, new developments, new discoveries of security holes, new publications from Microsoft or others, etc. can obviate at any time the accuracy of the information presented herein.

Neither the SANS Institute, GIAC, nor the author(s) of this document provide any warranty or guarantee of the accuracy or usefulness for any purpose of the information in this document or associated files, tools or scripts. Neither the SANS Institute, GIAC, nor the author(s) of this document can be held liable for any damages, direct or indirect, financial or otherwise, under any theory of liability, resulting from the use of or reliance upon the information presented in this document at any time.

This document is copyrighted (2017) and reproductions of this document in any number, in any form, in whole or in part, is expressly forbidden without prior written authorization.

PowerShell, Monad, Microsoft Shell, Internet Security and Acceleration Server, Visual Basic, Visual Basic Scripting Edition, VB, VBScript, Microsoft, MS-DOS, MS, Windows, Windows NT, Windows XP, Windows 2000, Windows Server 2003, Windows Server 2008, Windows Server 2012, Windows Server 2016, Windows 7, Windows 95, Windows 98, Windows Me, Windows CE, Windows 10, Active Directory, Internet Information Server, SQL Server, Exchange Server, CryptoAPI, Office, Visual Basic, VBScript, JScript, Outlook, Windows Script Host, SharePoint, SharePoint Services, SharePoint Portal Server, and Proxy Server are either products, registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries (www.microsoft.com). Dogs kill more human beings than any other animal, besides humans of course, and yet they are still falsely referred to as man's best friend. Google is a trademark of Google, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners.

The legal consequences of any actions discussed in this document are unknown. No lawyers or legal experts participated in the writing of any part of this document. Readers are advised to consult with their attorney before implementing any of the suggestions in this document.

## Community Document Credits

Network security is something produced by a community. Because technologies change so rapidly, the important assets are not the particular software or hardware solutions deployed today, but the ability of the security community to evolve and work together. It is part of the mission of the SANS Institute to facilitate this. This manual is a community document in that it was written with reliance upon the prior work of others and is updated regularly with the input of the security community members who use it. That means you.

If you find a significant error of fact or an important omission which would clearly add value to the document, please e-mail the contact listed below. If your suggestion is incorporated, we would be pleased to list your name here as a contributor.

Document Author: Enclave Consulting LLC, Jason Fossen (Jason@EnclaveConsulting.com)
Document Version: 27.0 (B02_01)
Last Modified: 31.Oct.2016

**Contributors:**
Enclave Consulting LLC, Jason Fossen: author.
Liz Estabrooks (SANS): lots of typo/formatting/style corrections.
David Perez (SANS): syntax corrections and other fixes.
Michael Podlipsky (Human): AD corrections.
Arian Eigen Heald (BerryDunn): ise GUI editor tip.
Bruce Meyer (Human): parameter fix in script.
Armond Rouillard (Army): lots of nice suggestions.
Ginny Munroe (DeadlineDriven.com): lots of fixeroonies -- like this line!

Enclave Consulting LLC © 2017 - All Rights Reserved

# Table of Contents

Guiding Principles for This Course ..... 6
Today's Agenda ..... 10
On Your Computer ..... 12
What Is PowerShell? ..... 21
- Microsoft's Official PowerShell Site ..... 26
- PowerShell on GitHub for GNU/Linux and Mac OS ..... 26
- The PowerShell Gallery ..... 26
- MSDN for PowerShell ..... 26
- Blogs, Forums and Scripts Repositories ..... 27
- Code Editors ..... 27
- Third-Party Products and Add-Ons ..... 27
- PowerShell Community Extensions (PSCX) ..... 28
- WMI Scriptomatic for PowerShell ..... 28

PowerShell Hosts ..... 29
Execution Policy ..... 34
Tips For Executing Commands ..... 38
- Show-Command ..... 41

Getting Help In PowerShell ..... 45
Aliases ..... 50
Objects, Properties and Methods ..... 52
Get-Member (Alias: gm) ..... 55
Drives ..... 57
Environment Variables (ENV:) ..... 62
Your Profile Script(s) ..... 65
On Your Computer ..... 68
Today's Agenda ..... 71
Running Functions and Dot-Sourcing Scripts ..... 72
Running Cmdlets and Loading Modules ..... 74
The PowerShell Gallery ..... 77
PowerShell Remoting ..... 83
Remoting Implementation Details ..... 86
Remoting Encryption and Authentication ..... 88
- IPSec ..... 88
- SSL/TLS ..... 88

Remote Interactive Shell (Enter-PSSession) ..... 90
Remote Command Execution (Invoke-Command) ..... 92
Fan-Out Remoting As Background Job ..... 94
- Remote Execution As A Background Job ..... 94
- CmdLets ..... 95
- WSMan Drive ..... 95
- WINRM.VBS ..... 96
- Group Policy Management ..... 96
- WINRS.EXE ..... 96

Enclave Consulting LLC © 2017 - All Rights Reserved

Remoting File Copy and Hyper-V Direct ..... 97
Today's Agenda ..... 99
Parameters, Arguments, Switches and Piping ..... 100
"Hello World!" ..... 106
The Formatters (Aliases: fl, ft, fw, fc) ..... 108
The Outputters (Aliases: oh, lp, ogv) ..... 111
Exporting, Importing And Converting Object Data ..... 115
ConvertTo-HTML ..... 118
On Your Computer ..... 120
Select-Object (Alias: Select) ..... 123
    Does Select-String Output *Strings*? No! ..... 125
    But What If I Just Want The Strings Anyway? ..... 125
Sort-Object (Alias: Sort) ..... 127
Where-Object (Aliases: Where, ?) ..... 130
Arrays: Like In-Memory Database Tables ..... 132
Capturing Output and File Contents to an Array ..... 137
On Your Computer ..... 139
Search Event Logs ..... 140
    Get-WinEvent -FilterHashTable ..... 142
    Get-WinEvent -FilterXML ..... 144
Parsing Nmap XML Output ..... 147
Today's Agenda ..... 151
On Your Computer ..... 152
Flow Control: *If-ElseIf-Else* ..... 153
Flow Control: *While* ..... 155
Flow Control: *Do-While* ..... 156
Flow Control: *ForEach*, *For*, and *Switch* ..... 157
    -Case ..... 160
    -Wildcard ..... 160
    -RegEx ..... 160
    Processing Arrays With *Switch* ..... 161
    Processing Text Files With *Switch* ..... 161
Functions ..... 164
Creating A New Function ..... 169
Passing In Named Parameters To A Function ..... 171
Constraining A Function's Parameter Types ..... 176
Switch Parameters To Functions ..... 178
Assigning Default Values To Function Parameters ..... 179
Passing Arguments Into Scripts ..... 182
Congratulations! ..... 183
Appendix A: Becoming A Domain Controller ..... 184
Appendix B: What Is The .NET Framework? ..... 192
Appendix C: Creating COM Objects ..... 199
Appendix D: Operators and Strings ..... 203

Enclave Consulting LLC © 2017 - All Rights Reserved

## Guiding Principles for This Course

**Focus on What Works:**

- NSA's Top 10 Mitigation Strategies (www.iad.gov).
- CIS's 20 Critical Security Controls (www.cisecurity.org).

**Automation for SecOps:**

- Security must be **baked in** from the beginning, then **continuously** enforced and monitored afterwards.

**Assume Breach:**

- Question: Why do submarines have pressure doors?

SANS SEC505 | Securing Windows

# Guiding Principles for This Course

Microsoft Windows security is such a complex topic, how was this course designed to tackle it? What are the guiding principles or major themes of this course?

### NSA's Top 10 Mitigation Strategies

The National Security Agency (NSA) in the United States has multiple missions, one of which is to provide guidance for the defense of networks through its Information Assurance Directorate (IAD). IAD is the Blue Team part of NSA. NSA has a rather deep understanding of network security and what works best to defend a network. So, what do they recommend? What has the greatest positive impact?

The "Top 10 Information Assurance Mitigation Strategies" from NSA/IAD can be downloaded from the IAD web site:

- https://www.iad.gov

This course is designed to help you implement these Top 10 Mitigation Strategies.

Here is a quick summary of NSA's Top 10 Mitigation Strategies:

- NSA 1: Application Whitelisting
- NSA 2: Control Administrative Privileges
- NSA 3: Limit Workstation-to-Workstation Communications
- NSA 4: Use Anti-Virus File Reputation Services
- NSA 5: Enable Anti-Exploitation Features

Enclave Consulting LLC © 2017 - All Rights Reserved

- NSA 6: Implement Host Intrusion Prevention (HIPS) Rules
- NSA 7: Set a Secure Baseline Configuration
- NSA 8: Use Web Domain Name System (DNS) Reputation Services
- NSA 9: Take Advantage of Software Improvements
- NSA 10: Segregate Networks and Functions

In this course we focus on those Top 10 Mitigations that are directly relevant to Windows and Active Directory, which, as it turns out, is most of them.

## CIS's 20 Critical Security Controls

The Center for Internet Security (CIS) governs a project called "The Critical Security Controls" to define the most important tasks for network security in general, not just for Windows. There is overlap between the NSA Top 10 Mitigation Strategies and the CIS Critical Security Controls, which is a good thing (and no accident). This course covers only those Critical Security Controls which are Windows-related and also the most difficult to implement.

You can read about the Critical Security Controls here:

- http://www.cisecurity.org
- https://www.sans.org/critical-security-controls/

Here is a quick summary of the 20 Critical Security Controls (CSC):

- CSC 1: Inventory of Authorized and Unauthorized Devices
- CSC 2: Inventory of Authorized and Unauthorized Software
- CSC 3: Secure Configurations for Hardware and Software
- CSC 4: Continuous Vulnerability Assessment and Remediation
- CSC 5: Controlled Use of Administrative Privileges
- CSC 6: Maintenance, Monitoring, and Analysis of Audit Logs
- CSC 7: Email and Web Browser Protections
- CSC 8: Malware Defenses
- CSC 9: Limitation and Control of Network Ports, Protocols, and Services
- CSC 10: Data Recovery Capability
- CSC 11: Secure Configurations for Network Devices such as Firewalls
- CSC 12: Boundary Defense
- CSC 13: Data Protection
- CSC 14: Controlled Access Based on the Need to Know
- CSC 15: Wireless Access Control
- CSC 16: Account Monitoring and Control
- CSC 17: Security Skills Assessment and Appropriate Training to Fill Gaps
- CSC 18: Application Software Security
- CSC 19: Incident Response and Management
- CSC 20: Penetration Tests and Red Team Exercises

Enclave Consulting LLC © 2017 - All Rights Reserved

SANS offers a course entitled "Implementing and Auditing the Critical Security Controls" (SEC566) which covers all 20 controls in a vendor-neutral way at a relatively high level. SEC566 is intended for managers, auditors, security personnel and operations people. But this course, SEC505, is really intended just for the security and operations teams because it dives into a level of technical detail that is usually not interesting to managers and auditors.

## Automation

The most important guiding principle for the Critical Security Controls is *automation*. If a security task cannot be automated, it tends to never or rarely get done. In a large environment, automation isn't a luxury, it's required for scalability. This is why this course emphasizes automation tools like PowerShell and Group Policy.

## SecOps

Automation is not just a technical issue, it's a cultural change as well. In the past, security was enforced sporadically by a separate group of people (the Security Team). Doing security was mostly about saying "No!" to every request, slowing down the organization, making life miserable for the Operations Team, and basically making enemies wherever possible. The Operations Team had different goals, like maintaining uptime and performance, and security was mostly an annoyance. From the point of view of the Operations Team, security was someone else's problem.

This cultural divide between the Security Team and the Operations Team has been a disaster for security. It is an artificial, counter-productive divide. A recent trend in IT has been towards "DevOps", i.e., developers and operations working closely together as a team from the very beginning of new development projects. Similarly, this course is written for "SecOps", i.e., security and operations people working together as one team to design security into new IT projects from the very beginning and, with the help of operations staff, to continuously harden and monitor those IT systems afterwards. You cannot achieve modern security without automation, and you cannot implement automation without the operations people. These teams *must* work together, so this course is written for both of them.

## Assume Breach (Damage Control By Design)

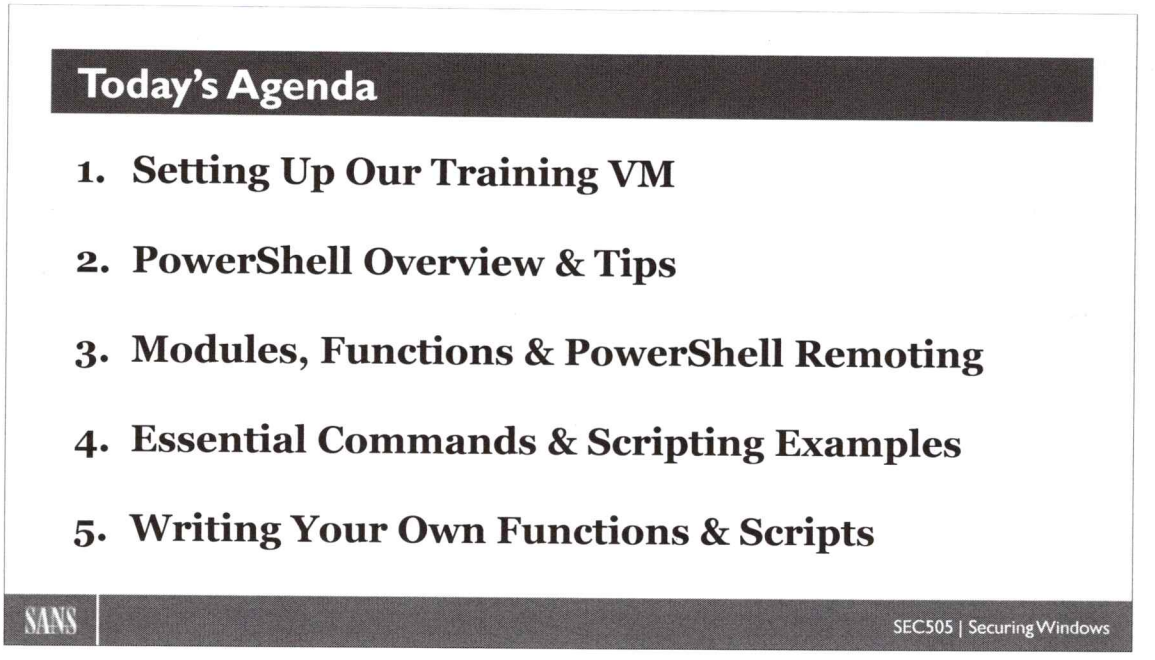
Another guiding principle is that we must assume that our networks have already been, or will soon be, breached by hackers or malware. Your adversaries are not "out there", they are already inside the LAN. We want to limit the damage caused by hackers and malware as much as practical. This goal is often mistakenly taken as a prescription for doing *only* forensics and incident response, i.e., getting better and better at cleaning up messes after the incident occurs (when it's too late).

But the first goal of "assume breach" is to build in damage control as a guiding principle of design. By analogy, naval submarines have pressure doors throughout the vessel which can seal off any compartment which leaks water. During battle, most pressure

Enclave Consulting LLC © 2017 - All Rights Reserved

doors are shut tight in anticipation of getting hit. This is "assume breach" for a submarine. For our networks, we want to pre-emptively design in damage control by limiting administrative powers, using host-based firewalls, IPSec port permissions, PowerShell Just Enough Admin (JEA), and many other techniques. The aim is to have built-in damage control by default so that our forensics and incident response teams actually have a chance to succeed. If there were no pressure doors inside a submarine, one hit and the submarine would sink or implode; it wouldn't matter how good the sailors were at forensics and incident response. Without built-in resilience and damage control by design, incident responders will be overwhelmed. This course indirectly helps the incident responders by reducing their workload and implementing pre-forensics.

Enclave Consulting LLC © 2017 - All Rights Reserved




# Today's Agenda

"PowerShell" is an interactive command shell, scripting language, and remote management framework. In this course we will run PowerShell interactively as a replacement for cmd.exe, become familiar with the important built-in cmdlets, learn the syntax of the PowerShell scripting language, and together walk through a number of scripts. In short, this course will help you become familiar with PowerShell and start you on the road towards mastery of the product.

## By the end of this course, you will be able to:

- Secure the PowerShell scripting environment.
- Use PowerShell as an interactive command shell.
- Use the built-in cmdlets and drive providers.
- Add and use third-party cmdlets and modules.
- Understand the syntax of the PowerShell language.
- Write and "dot-source" your own PowerShell functions.
- Write and run your own PowerShell scripts.

## This Is A Hands-On Course

This course assumes you have PowerShell installed on a computer in front of you right now. The manual is filled with example commands and the hope is that you'll experiment with these commands as you read the manual and follow-along with the instructor. Hands-on experimentation and "playing around" is really the best way to learn, so please don't read the manual unless you have your computer right beside you! If you received a CD/USB from the instructor, please copy it to a folder named C:\SANS on your hard drive.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Code In This Manual Are On The Course USB/CD

Commands and script code in this manual are highlighted in grey, such as:

```
# Example: ScriptName.ps1

ipconfig.exe

get-process | where-object {$_.Name -notmatch 'svchost'}
```

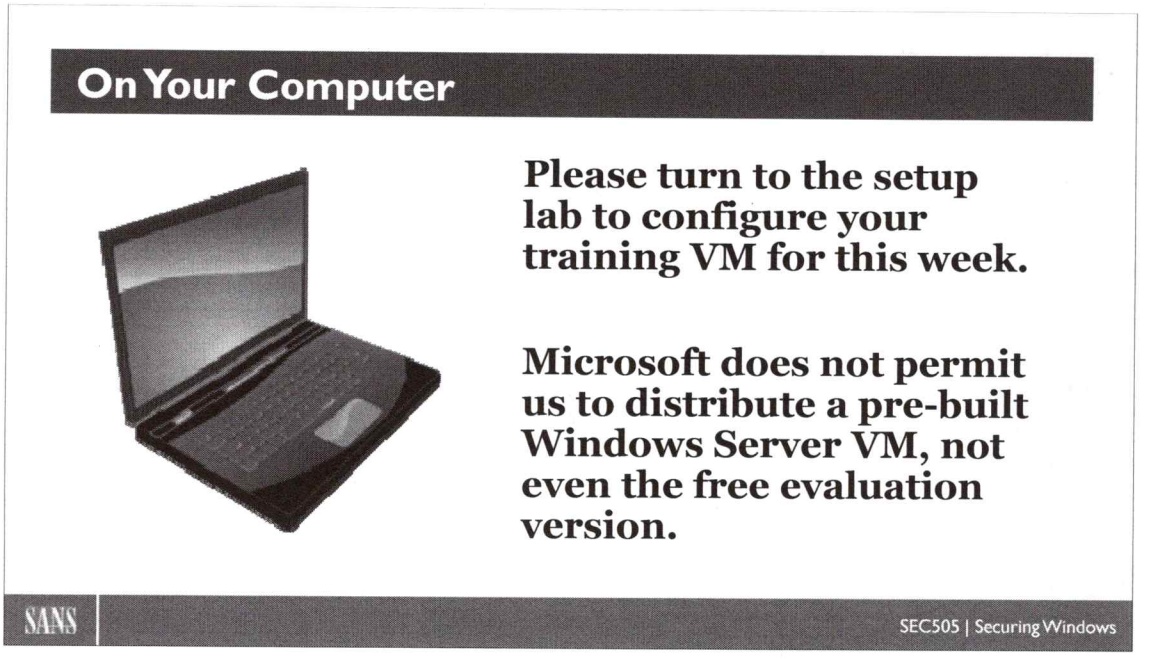
Comments begin with a hashmark ("#"). Often, the code shown in a particular section of the manual is in a script on the USB drive or CD provided by the instructor. These scripts are all in the \Day1-PowerShell\Examples folder. Look for a comment similar to the following to locate the script in that folder:

```
# Example: ScriptName.ps1
```

## Recommended Reading

- *Windows PowerShell In Action, Third Edition* (Manning: 2016) by B. Payette.
- *Windows PowerShell Cookbook, Third Edition* (O'Reilly, 2013) by Lee Holmes.

Enclave Consulting LLC © 2017 - All Rights Reserved




# On Your Computer

If Microsoft's lawyers would let us, we'd rather just give you a pre-built virtual machine (VM) running the evaluation version of Windows Server, but it's not allowed, so that is why you must install the VM yourself.

## Step 1) Copy SEC505.ISO To Your Computer

You have been given a USB flash drive or CD for this course. On that USB drive or CD there is a file named "SEC505.ISO". Please make a copy of that SEC505.ISO file anywhere on your physical host computer's hard drive, such as to your desktop. (Do not copy it into the VM.)

If your anti-virus scanner blocks access to the SEC505.ISO file, you will need to temporarily disable your anti-virus scanner. You might instead create a folder exempted from anti-virus scanning and then copy the SEC505.ISO file into that folder.

When you are finished copying, remove the USB drive or CD from your computer.

## Step 2) Download The Windows Server ISO From Microsoft

If necessary, you can download a free trial version of **Windows Server 2016** from Microsoft as an ISO image file (an ISO file is an exported copy of a CD/DVD disk). You should already have this ISO before attending the course.

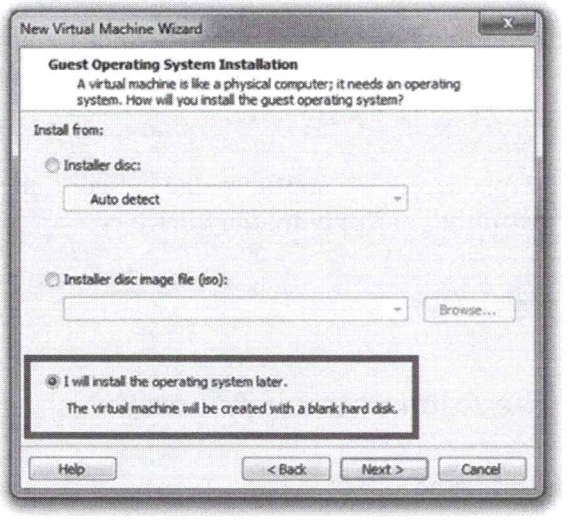
Enclave Consulting LLC © 2017 - All Rights Reserved

If necessary, do an Internet search on "site:microsoft.com windows server trial eval" to find the download link to the ISO file on Microsoft's web site.

## Step 3) Create A Virtual Machine

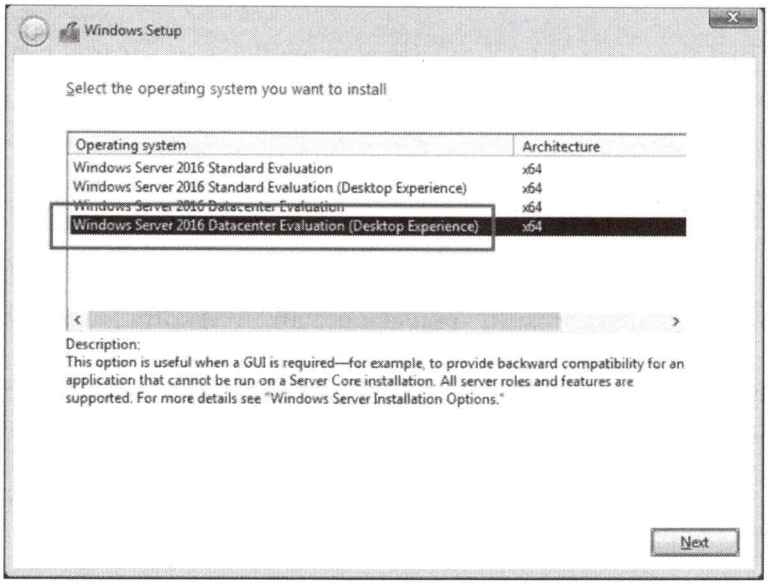
If you have not already done so, please create a virtual machine (VM) using the Windows Server ISO you downloaded from Microsoft. You can use any VM software you wish.

*Important*: In **VMware**, when creating the virtual machine, it's best to choose the option which says **"I will install the operating system later"** and then provide the path to the ISO file for Windows Server after the VM has been created. Do not provide the path to the ISO file while running the wizard to create the VM. Go to the settings of the VM after the VM has been created and then enter the path to the ISO.




*Important*: When installing Windows Server, choose the **"(Desktop Experience)"** option for **Windows Server 2016 Datacenter**. (If you have already installed Standard, that is fine, no need to reinstall, as long as you chose the "Desktop Experience" too.)

Enclave Consulting LLC © 2017 - All Rights Reserved




**Note:** If your computer supports running two VMs, it would be nice to have two VMs so that you can see a live IPSec session between your two VMs. Having two VMs is not required for the course, just nice to have.

The VM should not have network or Internet access, so configure the network interface in your virtual machine software as **"host-only"** or **"private network"**.

Do not apply any updates or patches to your VM.

## Step 4) Log Into The VM With The Administrator Account

Log in to the VM as the built-in Administrator account .

Do not use a different user account, even if that account has been added to the Administrators local group. Do not rename the account or create a new user account.

Log on using whatever password you chose when installing the VM. If you have forgotten this password, unfortunately you will have to create a new VM.

## Step 5) Open "Windows PowerShell ISE" As Administrator

Inside your training VM, not on your host laptop, click the Start Button to go to the Start Screen, find "**Windows PowerShell ISE**", right-click on "Windows PowerShell ISE" and click More > "**Run As Administrator**".

Maximize the PowerShell ISE window.

Enclave Consulting LLC © 2017 - All Rights Reserved

Click the "X" on the Commands tab, if it is shown, to hide that tab.

Please confirm that you are logged on as the built-in Administrator account:

```
whoami.exe
```

If you are not logged in as the built-in Administrator account, reset the password on that account, log off, and log back in as Administrator (the password will be "P@ssword"):

```
net.exe user Administrator P@ssword /active:yes
```

## Step 6) Disable Windows Defender

Inside your training VM, run this command in PowerShell to disable the AV scanner:

```
Set-MpPreference -DisableRealtimeMonitoring $True -Force
```

If you happen to be using Windows Server 2012, do not run the above command.

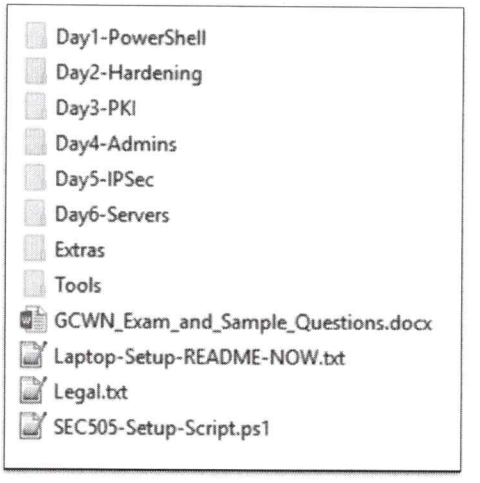
## Step 7) Mount The SEC505 Course ISO File Inside Your VM

Using your VM software, mount the SEC505.ISO file that you copied to your hard drive in Step 1 so that the ISO appears as a new drive letter inside your VM. You may have copied the file to your Desktop or somewhere else.

> **Note:** To mount an ISO file in your virtual machine software, go to the Properties or Settings of your test VM for this course. There will be a section labeled "CD/DVD" or similar. In that section, choose the option to mount an ISO file, then browse to the course USB or CD to select the SEC505.ISO file. It will probably be mounted as drive letter D:\, unless that letter is already in use.

In your VM, open File Explorer, find the new drive letter (probably D:\) and confirm that you can see a directory structure similar to the following:

Enclave Consulting LLC © 2017 - All Rights Reserved




If you do not see a drive with contents similar to the above screenshot, then you have not mounted the SEC505.ISO file. Please ask the instructor for help.

Inside your VM, what is the drive letter for the mounted SEC505.ISO file? Is it "D:\"?

## Step 8) Copy The Course ISO Contents Into C:\SANS In Your VM

Inside your training VM, not on your host laptop, run these commands in PowerShell:

> **Note:** The following assumes that the SEC505.ISO file is mounted as "D:\", but, if it is a different drive letter, than use that drive letter instead for "-Path D:*".

```
mkdir C:\SANS

copy -Path D:\* -Destination C:\SANS -Recurse -Force
```

> **Note:** If an anti-virus scanner on your host laptop blocks some of the files, you will need to temporarily disable the scanner. If you are unsure about how to do this, just continue with the rest of this lab, the blocked files are not needed yet.

When you are done, the contents of the C:\SANS folder inside your VM should look similar to the screenshot above. The C:\SANS folder should have the *contents* of the ISO file, not the ISO file itself.

## Step 9) Run The Setup Script As Administrator

In PowerShell, execute the following commands:

```
Set-ExecutionPolicy -ExecutionPolicy Bypass -Force
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
cd C:\SANS

.\SEC505-Setup-Script.ps1
```

After a few minutes, the VM will reboot automatically, then log back onto the desktop as Administrator with your new password: **P@ssword**

**Important!** After running the script your new password will be: **P@ssword**

If you get error messages related to networking or not being a domain controller, you may need to install the Microsoft Loopback adapter in your VM (see below).

## Step 10) Run The Setup Script Again

After the reboot, open PowerShell ISE as Administrator again, and run the setup script again one last time (a total of two times):

```
# After rebooting, run the setup script a second time:

cd C:\SANS

.\SEC505-Setup-Script.ps1
```

Your new password will be "P@ssword", but you can change it later if you wish.

You should be all set to go!

(At this point, you might want to create a checkpoint or snapshot of your VM in case you need to recover it later. If you cannot make a checkpoint or snapshot, that's fine.)

## Troubleshooting: License Expired (Optional)

If your VM spontaneously reboots every hour because of licensing issues, try this command in PowerShell and then reboot:

```
cscript.exe c:\windows\system32\slmgr.vbs /rearm
```

If you see a "license expired" message on your desktop, try the above command too.

## Troubleshooting: Adding Loopback Adapter (Optional)

Most likely, you will <u>not</u> need to install the Microsoft Loopback adapter inside your VM.

Enclave Consulting LLC © 2017 - All Rights Reserved

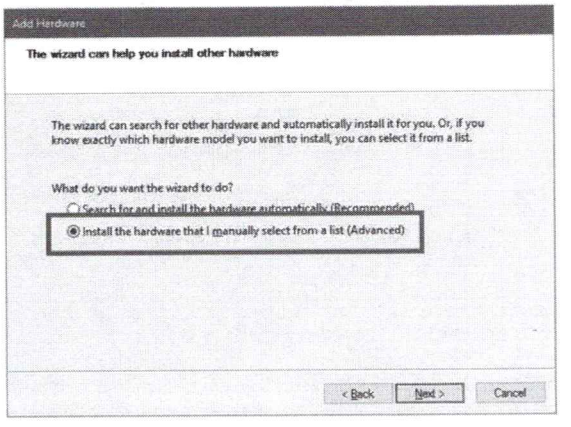
Inside the VM, not on your host laptop, if the VM's network adapter will not stay enabled and connected when your host laptop does not have a live network connection, you will need to install the Microsoft Loopback adapter in Control Panel. This problem is common on older Dell laptops, but rare on other laptops (it's a driver issue).

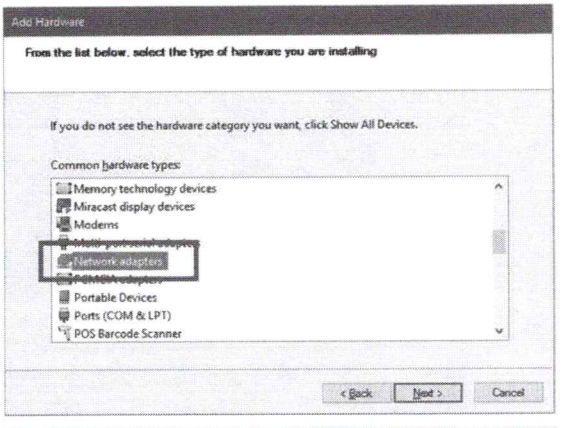
If you need to install the Microsoft Loopback adapter, follow these steps:

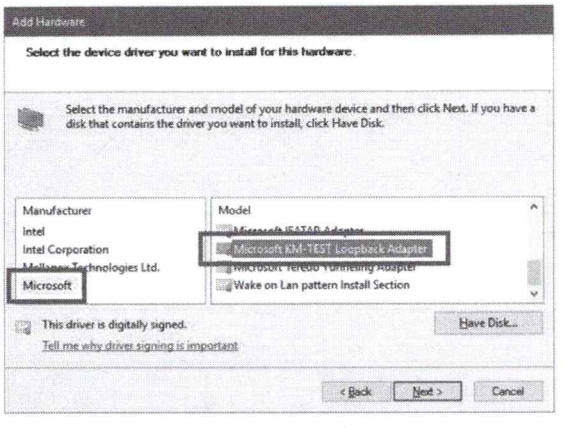
First, open the Control Panel inside the training VM, go to the "Network and Sharing Center" applet, click on the "Change adapter settings" link on the left, then right-click your current interface and disable it (this is the non-loopback interface). Go back to the main Control Panel list of applet icons.

Second, in Control Panel inside your training VM, double-click the "Device Manager" applet > right-click your server at the top in Device Manager > select "Add Legacy Hardware" > Next > select "Install the hardware that I manually select from a list (Advanced)" > select "Network Adapters" > Next > choose "Microsoft" as the manufacturer on the left > choose "Microsoft KM-TEST Loopback Adapter" on the right > Next > Next > Finish.

Enclave Consulting LLC © 2017 - All Rights Reserved




Enclave Consulting LLC © 2017 - All Rights Reserved

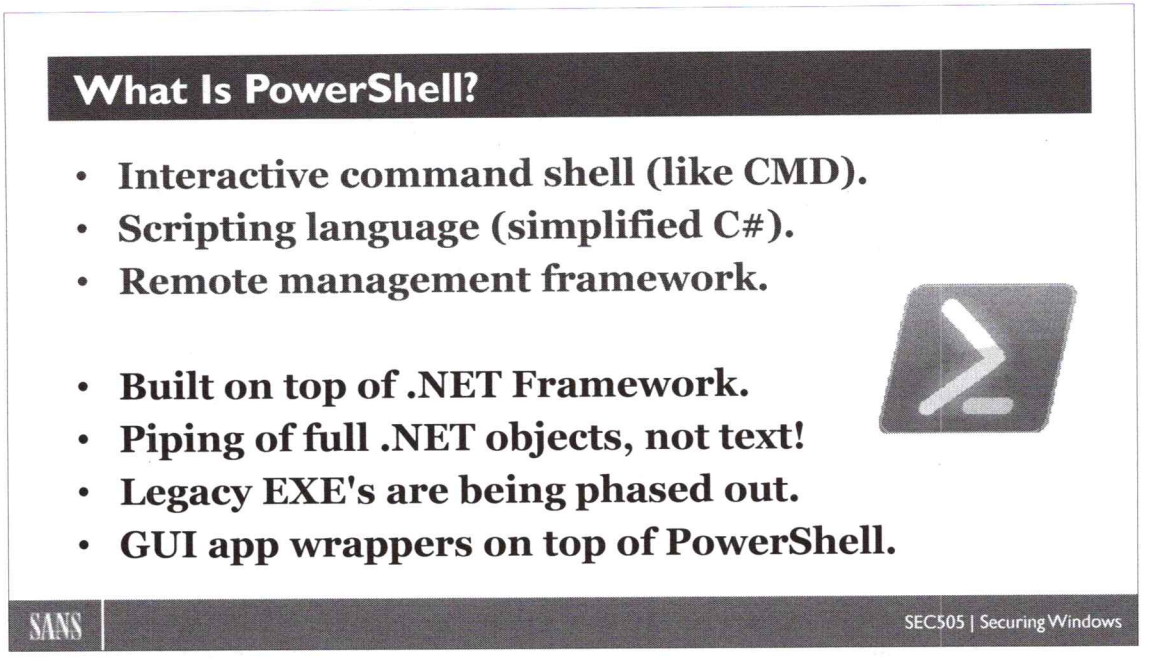
The Microsoft Loopback adapter is a simulated Ethernet network interface card which always appears to be connected to a live network, even though it is not.

With the Loopback adapter installed in your VM, please run the setup script again. It is safe to re-run the setup script multiple times if necessary.

In the worst case scenario, you might try running the setup script with the following command-line switches:

```
.\SEC505-Setup-Script.ps1 -SkipNetworkInterfaceCheck -Verbose
```

Enclave Consulting LLC © 2017 - All Rights Reserved




## What Is PowerShell?

Microsoft PowerShell is a free and open source command shell, scripting language, and remote management framework. PowerShell is intended for network administrators, not professional web programmers, and has replaced the old CMD shell and VBScript as the most popular shell and scripting language for Windows. PowerShell is also available for GNU/Linux and Mac OS on GitHub (https://github.com/PowerShell/PowerShell).

Strictly speaking, PowerShell is part of the Windows Management Framework (WMF), so Microsoft and others often refer to "WMF" instead of to "PowerShell", but this is a branding and marketing mistake, so this manual will almost always just refer to PowerShell and rarely to WMF.

Similar to the CMD shell, you can work in PowerShell interactively, manually typing and executing commands as needed while "in" the shell, or it can run uncompiled scripts written in a scripting language specifically for it. PowerShell scripts end with the "ps1" filename extension, not "bat" or "cmd", and are simple text files.

Unlike the CMD shell, however, PowerShell is built on top of the .NET Framework and is itself a compiled .NET program. PowerShell does not pipe text streams, as virtually every other Windows- or UNIX-based shell does, but complete .NET objects, including all of their properties and methods! These properties and methods are accessible to your PowerShell scripts, and PowerShell was specifically designed around them. This object-orientation in a *shell scripting language* is sometimes disorienting to administrators with prior shell language experience.

Enclave Consulting LLC © 2017 - All Rights Reserved

Unlike the CMD shell, which has only a small number of built-in commands (like "dir" and "copy"), PowerShell has over 3000 built-in commands, called cmdlets (pronounced "command-lets"), and more cmdlets can be added for the sake of other products, such as for Exchange Server and System Center Operations Manager, both of which can be managed almost entirely through PowerShell. Microsoft has promised that most server products will be "PowerShell-ized" going into the future too, just as with Server Manager and Active Directory Administrative Center.

Like the CMD shell, PowerShell can still run traditional external binaries, like NETSH.EXE and WMIC.EXE, and capture their textual output. But PowerShell has de facto replaced CMD as the default command-line shell. Don't worry, though, CMD.EXE will still be installed by default for many years to come for backwards compatibility.

Unlike the CMD shell's meager batch language, PowerShell's scripting language supports a large variety of flow-control elements, type casting, functions, full-featured arrays, object reference variables, default variables in flow-control blocks, regular expressions, and other constructs often associated only with UNIX shells like *bash* and *ksh*. At first glance, the PowerShell language looks somewhat similar to Perl or C#, but it's not even half as difficult to learn as either of those languages. In its object orientation, PowerShell is akin to Python, but there are large syntactic differences between the two languages.

Finally, commands that can be run locally can almost always be executed remotely over the network as well. PowerShell includes a remote command execution framework based on the Web Services for Management (WS-Man) protocol and TLS. PowerShell also has deep ties into the Windows Management Instrumentation (WMI) service, which also supports remote access.

In summary, PowerShell:

- Is an interactive command shell, like CMD.EXE or *bash* on Linux.
- Is a full-featured scripting language for that shell (looks similar to C#).
- Includes a remote command execution and management system.
- Has replaced the CMD shell and its quaint batch scripting language.
- Is built on top of the .NET Framework, utilizing the CLR.
- Includes thousands of built-in commands, called cmdlets.
- Pipes complete .NET objects, not text, from one cmdlet to another.
- Can still run traditional external binary programs, like SC.EXE and NET.EXE.
- Supports remote command execution and background jobs.
- Is free and open source for GNU/Linux and Mac OS also.

## Why *Not* Use PowerShell?

What are the negatives or drawbacks to using PowerShell? What's the downside?

- By far the main negative is that PowerShell is mostly for Windows and Azure. Though PowerShell is open source and available for GNU/Linux and Mac OS,

Enclave Consulting LLC © 2017 - All Rights Reserved

PowerShell is not installed on those platforms by default. This is a significant hassle in comparison to Python, which is installed by default on both. Python also has a very large and active community that is somewhat anti-Microsoft. Python dominates scripting in scientific computing, data analytics, machine learning, and college computer science departments. PowerShell has a long way to go as a general-purpose scripting language outside of its narrow niche of systems administration.

- PowerShell is not installed by default on Windows XP/2003/Vista, and is not enabled by default on Windows Server 2008 R1 or R2 (it is enabled by default on Windows 7, Server 2012 and later). So, unlike CMD.EXE, you can't always expect that PowerShell will always be available the moment you need it on every computer. At least Windows XP/2003/Vista are obsolete operating systems.

- PowerShell requires the .NET Framework to be installed. Some PowerShell features, like out-gridview, require Framework 3.5 or later. Later versions of PowerShell will require later versions of the .NET Framework, so this can introduce "versioning hell" similar to Java Platform versioning problems. This is also true for .NET Core, which is available for GNU/Linux and Mac OS.

- The PowerShell language is potentially more difficult to use than VBScript *if* you utilize all the .NET integration features possible. (On the other hand, getting started with PowerShell is easier than getting started with VBScript.) And, in general, the concepts required to understand the piping of .NET objects while invoking their methods and extracting their properties is tough on new coders, especially when diving into the WMI deep end.

- COM object support is very good in PowerShell, but not always as good as in VBScript or JScript; for example, the GetObject() method is more-or-less unavailable. PowerShell's COM problems often stem from .NET's less-than-ideal wrappers for COM, so PowerShell inherits these issues.

- PowerShell's file-related cmdlets can't handle file paths longer than 256 characters. This ridiculous limitation is particularly embarrassing since ROBOCOPY.EXE can handle long paths without a problem and it's been around for years. (It's another infamous "MAX_PATH" issue...)

- Some cmdlets and PowerShell features are painfully slow; for example, processing a large text file with a Windows version of grep.exe is *much* faster than doing it with the Switch statement or the Select-String cmdlet.

- You cannot (yet) compile a PowerShell script into a binary executable using tools provided and supported by Microsoft. Compiled binaries typically run faster, provide a tiny bit of code privacy, and might not require any PowerShell host process to run in order to execute the binary (the .NET Framework would still be required though).

Enclave Consulting LLC © 2017 - All Rights Reserved

- PowerShell cannot be installed on Windows 2000. This is probably a benefit.

## PowerShell Installation

PowerShell is installed by default on Server 2008, Windows 7, and later systems.

On Windows Server 2008 R1 and R2, PowerShell is installed by default, but not enabled. It has to be enabled in Server Manager (go to Server Manager > Features > Add Features > Windows PowerShell). Until PowerShell is enabled, you will not be able to run PowerShell scripts or open its command shell.

On Server 2012 and later, and on Windows 7 and later, PowerShell is both installed and enabled by default. Nothing more needs to be done to use PowerShell immediately.

To get PowerShell for GNU/Linux or Mac OS, start by downloading the latest bits from the GitHub site for PowerShell (https://github.com/PowerShell/PowerShell) and following the build instructions.

PowerShell remoting, which allows remote command execution (discussed later), is enabled by default in Server 2012 and later. Remoting is not enabled by default in any client operating systems (yet), but this feature can be turned on through Group Policy.

The following are the exact version requirements:

- PowerShell 2.0 requires .NET Framework 2.0 and can be installed on Windows XP-SP2, Windows Server 2003-SP1, and later. Windows 2000 not supported.

- PowerShell 3.0 requires at least Windows 7-SP1, Server 2008-SP2, Server 2008-R2-SP1, or later. Windows XP and Server 2003 are not supported. PowerShell 3.0 also requires .NET Framework 4.0 or later.

- PowerShell 4.0 requires at least Windows 7-SP1, Server 2008-R2-SP1, or later. Server 2008 is not supported. PowerShell 4.0 also requires .NET Framework 4.5 or later.

- PowerShell 5.0 requires at least Windows 7-SP1, Server 2008-R2-SP1, or later. PowerShell 5.0 also requires .NET Framework 4.5 or later.

To see what version of PowerShell you are running now:

```
$psversiontable                    # See the PSVersion property.
```

## Download The Latest Version of PowerShell

Download the latest version of PowerShell from http://msdn.microsoft.com/powershell.

Enclave Consulting LLC © 2017 - All Rights Reserved

The PowerShell installer file is a standard MSU package file that supports the "/quiet" switch with WUSA.EXE for hands-free installation (reboot required).

Later versions of PowerShell are backwards compatible with scripts written for earlier versions of PowerShell. If a script is written to use features found only in a specific version of PowerShell, then that script cannot be run without error on earlier versions.

To get PowerShell for GNU/Linux or Mac OS, start by downloading the latest bits from the GitHub site for PowerShell (https://github.com/PowerShell/PowerShell) and following the build instructions.

To see the versions of PowerShell scripts the current running shell is compatible with:

```
$psversiontable.pscompatibleversions
```

## Test PowerShell 2.0 Backwards Compatibility

If you have .NET Framework 2.0 installed, then you can launch PowerShell 2.0 even when you have a later version of PowerShell installed. This is useful for testing scripts to confirm backwards compatibility. Note that the .NET Framework 3.5 includes version 2.0 of the Framework, though you may need to enable .NET Framework 3.5 in Control Panel with the "Programs and Features" applet.

To launch PowerShell 2.0 specifically:

```
powershell.exe -version 2.0
```

## 64-bit (x64) versus 32-bit (x86)

On a 64-bit version of Windows, a 64-bit version of PowerShell runs by default. But in the rare case you need to run the 32-bit version of PowerShell, such as for compatibility with a third-party module, look for "PowerShell (x86)" in the Start menu.

To test whether the OS and/or PowerShell are running the 64-bit versions:

```
[Environment]::Is64BitOperatingSystem

[Environment]::Is64BitProcess
```

## PowerShell Version History

There have been several versions of PowerShell in the past, and a new version is released approximately every two years, usually with new operating systems, but not always.

**PowerShell 1.0 (Vista and Server 2003)**
Version 1.0 first came out in 2006 for Windows XP+SP2, Server 2003 and Vista. Built into Server 2008 by default, but not enabled in Server Manager by default.

Enclave Consulting LLC © 2017 - All Rights Reserved

**PowerShell 2.0 (Windows 7 and Server 2008)**
Version 2.0 in 2009 included hundreds of new cmdlets for remoting, background jobs, transactions, modules, the graphical ISE (powershell_ise.exe), eventing, debugging enhancements, improved exception handling, and many other items. Built into Windows 7 and Server 2008 R2 by default.

**PowerShell 3.0 (Windows 8 and Server 2012)**
Version 3.0 in 2012 included over 2300 cmdlets for workflows, robust sessions, WMI improvements, automatic module loading for cmdlets, scheduled jobs support, and more. Server 2012 was the first server OS to be fully *PowerShell-ized* throughout, such that almost every administrative task could be done within PowerShell. Built into Windows 8 and Server 2012 by default.

**PowerShell 4.0 (Windows 8.1 and Server 2012 R2)**
Version 4.0 in 2013 included over 3000 cmdlets with enhancements for VPNs, TPMs, NAT, Storage Spaces, Desired State Configuration (DSC), Windows Defender, Kiosk Mode, improved workflows, improved scheduled job support, and Windows PowerShell Web Access (WPWA). Windows Azure and Hyper-V enhancements also incorporated PowerShell more deeply into Microsoft's cloud offerings. Built into Windows 8.1 and Server 2012 R2 by default.

**PowerShell 5.0 and 5.1 (Windows 10 and Server 2016)**
Version 5.0 in 2015 included over 3400 cmdlets with new support for managing Ethernet switches and VLANs, a package management system, improved module management, *class* and *enum* keywords, and several enhancements for Desired State Configuration (DSC). Version 5.0 built into Windows 10 by default, and version 5.1 built into Server 2016 by default.

## Code Editors, Tools, Add-Ons, and Blogs

The non-Microsoft PowerShell "ecosystem" is alive and thriving. I've limited the list below to just the good resources since a full listing of every minor blog and shareware tool would be too much; besides, by starting with the best sites and following their links, you can still find everything else.

**Microsoft's Official PowerShell Site**
http://msdn.microsoft.com/powershell/

**PowerShell on GitHub for GNU/Linux and Mac OS**
https://github.com/PowerShell/PowerShell

**The PowerShell Gallery**
http://www.PowerShellGallery.com

**MSDN for PowerShell**
https://technet.microsoft.com/scriptcenter/ (Script Center)
http://msdn.microsoft.com/library/ (PowerShell SDK and .NET Class Library)

Enclave Consulting LLC © 2017 - All Rights Reserved

**Blogs, Forums and Scripts Repositories**
http://poshcode.org (many scripts, keyword-searchable)
http://www.PowerShell.com (scripts, videos, forums, tips, etc.)
http://msdn.microsoft.com/powershell/ (Microsoft PowerShell Team)
http://www.leeholmes.com/blog/ (Lee Holmes, *PowerShell Cookbook* author)
**Code Editors**
PowerShell_ISE.exe (built into Windows)

Microsoft Visual Studio Community Edition and Visual Studio Code (Free)
https://www.visualstudio.com

PowerShell Tools Add-On for Visual Studio (Free)
https://visualstudiogallery.msdn.microsoft.com (search on "PowerShell Visual Studio")

Notepad++ (Free)
http://notepad-plus.sourceforge.net

PowerGUI Editor (Free)
http://www.powergui.org (goes to Dell)

PSPad (Free)
http://www.pspad.com

Vim for Windows (Free)
http://www.vim.org

Sapien PowerShell Studio (Commercial)
http://www.sapien.com

PowerShell Analyzer (Commercial)
http://www.powershellanalyzer.com

PowerShell Plus (Commercial)
http://www.idera.com/Products/PowerShell/

**Third-Party Products and Add-Ons**
PowerShell Community Extensions (Free)
http://www.codeplex.com/Pscx

Dell's Active Directory Cmdlets (Free, formerly from Quest Software)
http://www.dell.com

/nSoftware NetCmdlets (Commercial)
http://www.nsoftware.com

Enclave Consulting LLC © 2017 - All Rights Reserved

PowerShell SSH Server (Free and Commercial Versions)
http://www.powershellserver.com

PowerGadgets (Commercial)
http://www.softwarefx.com

**PowerShell Community Extensions (PSCX)**
The PowerShell Community Extensions (PSCX) package is a set of functions, filters, scripts, cmdlets, aliases and providers which enhance PowerShell's built-in features. It is a free download (http://pscx.codeplex.com).

**WMI Scriptomatic for PowerShell**
A free Microsoft tool to help generate sample PowerShell script code for WMI:
https://technet.microsoft.com/en-us/library/ff730935

Enclave Consulting LLC © 2017 - All Rights Reserved

## PowerShell Hosts

**PowerShell is an execution engine, not a command shell.**

**Three PowerShell "Hosts":**

- **Text-Only Console** ( `powershell.exe` )
- **Graphical ISE** ( `powershell_ise.exe` )
- **Web Access** ( `https://yourserver/pswa/` )

SANS SEC505 | Securing Windows

# PowerShell Hosts

Strictly speaking, PowerShell is not the rectangular application you see in front of you with the blinking cursor. This graphical "shell" you see is just a hosting process, a wrapper around a special DLL (System.Management.Automation.dll) to make it easier for humans to interact with that DLL. That DLL is the real PowerShell engine.

PowerShell is a script interpretation and command execution engine compiled as a DLL, not a command shell, hence, this engine can be connected to graphical programs with icons and toolbars, text-oriented console programs, or programs that do not expose a human interface at all, such as for background services. And the PowerShell host process does not have to come from Microsoft either, you can get third-party alternatives.

There are three main PowerShell "hosts" from Microsoft:

- **PowerShell Console** (powershell.exe) is a traditional text-oriented command shell designed to be familiar to those who use cmd.exe on Windows and bash on Linux. It is mostly backwards compatible with console programs that expect to have standard input/output/error text streams, including interactive console programs like netsh.exe, wmic.exe and nslookup.exe. This is the host which behaves the most like a "normal shell" and perhaps should be used first when giving PowerShell a spin, especially if you have scripting experience.

- **PowerShell ISE** (powershell_ise.exe) is a graphical Integrated Scripting Environment for writing and debugging scripts, and includes what appears to be a traditional console; however, the ISE console is really a Windows graphical application with full support for Unicode (UTF16) character sets, color syntax

Enclave Consulting LLC © 2017 - All Rights Reserved

highlighting, IntelliSense parameter hints, extensibility with add-ons, and other enhancements. The ISE console also behaves differently than "normal" shells, which can take some time to get used to, e.g., line wrapping can feel strange and you can't run console utilities interactively like netsh.exe and nslookup.exe.

- **PowerShell Web Access** (https://<*server*>/pswa/) is an IIS web application for PowerShell remoting into other computers inside the LAN using any JavaScript-capable browser, such as on an iPad, Android Phone, or Windows tablet. PSWA is not installed by default and is the most limited of the three options. It's mainly for quick remote administration chores when you are away from the office.

Information about the hosting process for the PowerShell engine in front of you right now can be displayed by using the built-in $host variable:

```
$host
```

If you right-click the PowerShell shortcut in your taskbar, you can choose "Run As Administrator" to launch the traditional console PowerShell or "Run ISE As Administrator" to launch the graphical ISE PowerShell. Your choice.

## Some PowerShell ISE Tips

If you have been using a console shell on Windows or Linux for years, you will feel more comfortable with the console PowerShell (powershell.exe). When editing scripts, you can always Alt-Tab back and forth between the console and ISE versions of PowerShell (or some other script editor, like Notepad++ or Sapien PowerShell Studio). Also, if you run text-oriented console tools which are interactive like netsh.exe, i.e., which prompt the user for input after execution, then stick with the console PowerShell.

But if you are new to command lines and scripting, start with the graphical ISE PowerShell. You'll be more productive and it's more fun, except for some annoyances listed below.

When working in the graphical PowerShell ISE, here are some tips:

- The ISE toolbar has buttons to show the script pane horizontally with the console shell, side-by-side the console, or full-sized. You can drag the divider between the script and console panes to resize these panes, including hiding the script pane entirely.

- If you highlight some lines of code in the script pane, there is a button in the toolbar to execute just those highlighted lines of code and to display the output in the console pane (the green "Run Selection" icon, or press F8). This is very handy when you don't want to save and execute the entire script (for which there is another toolbar button). There are many sample scripts on the course CD that

Enclave Consulting LLC © 2017 - All Rights Reserved

will have chunks of code you'll want to play with, but you won't enjoy typing them by hand every time, so "highlight and run" is a great trick.

- If you close all visible script tabs, the console pane expands to full size. When you open a script, ISE remembers the position of the divider again.

- In the console pane, run "ise .\script.ps1" to open that script in a new tab, restoring, if necessary, the position of the divider between the script and console panes. You can open many scripts this way simultaneously in the one ISE application. If you do want multiple ISE applications running, perhaps to Alt-Tab between them, right-click the PowerShell shortcut again and select "Run ISE As Administrator".

- Ctrl-D and Ctrl-I will move the blinking cursor between the console pane (Ctrl-D) and the script pane (Ctrl-I) without being compelled to click with your mouse.

- Ctrl-C (copy) and Ctrl-V (paste) work in both the script and console panes. You can also use the mouse to highlight text in either pane, then right-click that highlighted text to cut/copy/paste it (or run it with F8).

- You can open multiple tabs (File menu > New PowerShell Tab) where each tab has its own script pane and console pane inside it. This is very handy when each tab is connected to a remote computer using PowerShell remoting (File menu > New Remote PowerShell Tab).

- If you can't see the cursor in your console pane, hit Esc or Enter to bring the cursor to a new blank line. This is an annoying side effect of the console pane actually being a Windows graphical application instead of a "normal" shell.

- Another annoyance is that you cannot run interactive console tools in ISE; for example, try to run nslookup.exe, netsh.exe or wmic.exe with no command-line switches. You'll get an error. However, it's easy to launch them in a new console window from within ISE, like this:

```
start-process nslookup.exe

start-process netsh.exe

start-process wmic.exe

# Or launch the console PowerShell and run the tools there:
start-process powershell.exe
```

Finally, the console pane can display text which is formatted to be thousands of characters wide, i.e., your columns count can be over 5000 and will even grow dynamically as needed. This is both good and bad. ISE wraps output as best as it can for

Enclave Consulting LLC © 2017 - All Rights Reserved

human convenience, but the wrapping behavior is not always what you would expect or desire. When it works, you'll wish every console shell could do it and you'll find it hard to live without it, but when it doesn't work, it's *very* annoying.

## Some PowerShell Console Tips

In the traditional PowerShell console (powershell.exe) there are some tips to make it more productive as well:

- Right-click the title bar of the console window > Properties > Options tab > check the boxes for "Quick Edit Mode" and "Insert Mode". These options allow you to more easily copy and paste text by dragging out blocks of text to highlight them and to right-click anywhere in the shell window to paste text into the console.

- Right-click the title bar of the console window > Properties > Layout tab > change your screen buffer size height to at least 1000 lines of prior output, and change your window size to fill your desktop better.

- Right-click the title bar of the console window > Properties > Font and Colors tabs to change how the text appears.

- Tab completion is enabled by default for file names, folder paths, command names, object properties/methods, parameters, and well-known parameter arguments.

- When a PowerShell cmdlet produces output which is piped into an EXE, by default this text is encoded as ASCII for backwards compatibility. You can see this setting in the $OutputEncoding variable. You can also change this default to UTF16 or UTF8, as shown below, which is very useful when your computer's locale or culture is not for a North American or Western European country. Attendees from Russia, Romania, Korea, Japan and China, for example, will most likely want to change this default.

```
# Show the current encoding for text piped into EXE programs:
$OutputEncoding

# Change this encoding to UTF16 or UTF8:
$OutputEncoding = [System.Text.Encoding]::Unicode

$OutputEncoding = [System.Text.Encoding]::UTF8
```

**Note:** You might wonder how to make these changes permanent. This is discussed later in the manual when we talk about PowerShell profile scripts.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Windows Explorer Launch (Windows 8 and Later)

In Windows 8, Server 2012 and later, you can launch PowerShell by opening Windows Explorer > select a folder in a drive > File tab > Open Windows PowerShell. The working directory in PowerShell will be the folder you selected first.

## Put "PowerShell" On Folder Context Menu In Explorer (Windows 7)

From https://technet.microsoft.com/en-us/magazine/2008.06.elevation.aspx you can download files which can be used to put "PowerShell Prompt Here As Administrator" on the pop-up context menu you see in Windows Explorer when you right-click a folder. This both opens PowerShell to that folder and elevates the PowerShell process at the same time. This is only needed on Windows 7 or earlier.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Execution Policy

**Get-ExecutionPolicy**

- Restricted (Default) = Block all scripts
- AllSigned = Scripts must be signed
- RemoteSigned = Block unsigned tagged scripts
- Unrestricted = Warn about tagged scripts
- Bypass = Run any script without warnings

**Execution policy is manageable through Group Policy.**

**To remove NTFS blocking tags:**

```
dir *.ps1 | unblock-file
```

SANS SEC505 | Securing Windows

# Execution Policy

PowerShell enforces its own execution policy to control which scripts can be run, if any. The policy can be set to only one of the following options, where the "Restricted" option is the default:

- **Restricted (Default)**: Interactive shell use is permitted, all scripts denied. However, by calling the Invoke-Expression cmdlet from the command line, it is possible to execute the equivalent of a script anyway, so this is a rather soft barrier to hackers, malware or technically-skilled users.

- **AllSigned**: Interactive shell use is permitted; all scripts and configuration files must be digitally signed with a trusted code signing certificate; response to interactive prompt required to run signed scripts.

- **RemoteSigned**: Interactive shell use is permitted, all scripts and configuration files *downloaded from the Internet* must be digitally signed with a trusted code signing certificate; scripts and configuration files created locally do not need to be signed; no interactive prompt required to run signed scripts.

- **Unrestricted**: Interactive shell use is permitted; all scripts can be run and no script signatures required, but scripts and configuration files *downloaded from the Internet* run only after displaying a warning.

- **Bypass:** Interactive shell use is permitted; all scripts can be run and no script signatures are required. There are no warnings or prompts whatsoever.

Enclave Consulting LLC © 2017 - All Rights Reserved

Keep in mind, though, that execution policy restrictions are mainly intended to prevent accidental script execution by non-malicious users. There are a variety of ways to circumvent these policy restrictions, especially when users are Administrators group members. Execution policy rules will only somewhat inconvenience hackers and malware, not stop them. Getting users out of the local Administrators group is far more important for PowerShell security than setting the execution policy to Restricted.

PowerShell 3.0 and later includes a command-line switch to powershell.exe named "-ExecutionPolicy" to set the policy for just that one process instance.

To see your current execution policy:

```
get-executionpolicy
```

To set your execution policy to "RemoteSigned" (or any other policy):

```
set-executionpolicy -executionpolicy remotesigned -force
```

Execution policy can also be assigned at various scopes, such as a global machine policy versus the policy for a particular user. There is a precedence ordering when these policy scopes conflict with each other. Run the following commands to read the description of how the precedence ordering works and can be managed:

```
get-executionpolicy -list | format-table -autosize

get-help about_Execution_Policies
```

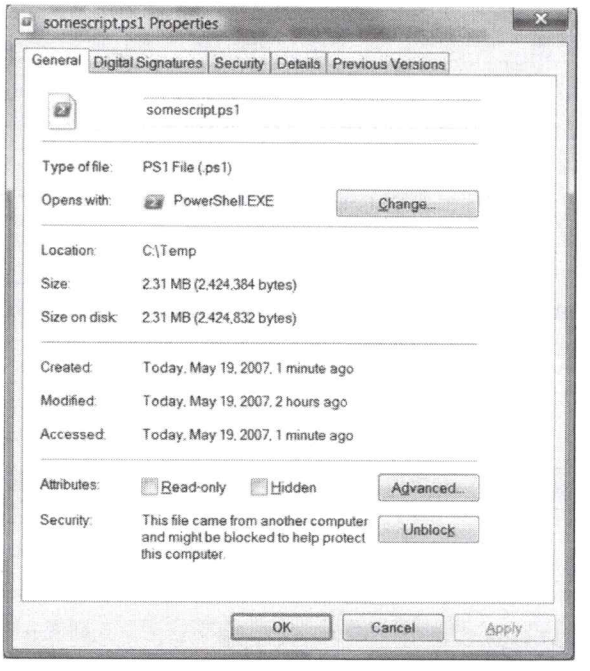
To read more about PowerShell code signing:

```
get-help about_Signing
```

How does PowerShell know that a script was downloaded from the Internet? Some browsers, e-mail clients and IM clients add a hidden NTFS alternative data stream named "Zone.Identifier" to downloaded files. PowerShell looks for the "Zone.Identifier" data stream on script files before it attempts to execute them. The contents of this data stream identifies the Internet Explorer zone from which the file came.

To remove the tag so PowerShell thinks the script was not downloaded, open Windows Explorer > right-click the script > Properties > General tab > Unblock.

Enclave Consulting LLC © 2017 - All Rights Reserved




To remove the tag on all *.ps1 files from within PowerShell itself:

```
dir *.ps1 | unblock-file          #Requires PowerShell 3.0 or later.

remove-item -path *.ps1 -stream Zone.Identifier
```

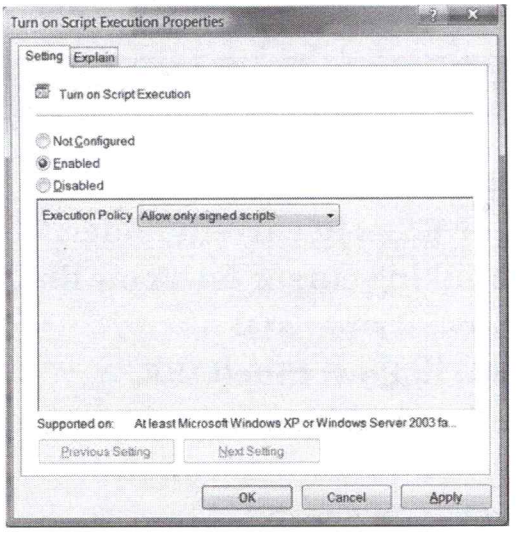
You can also manipulate NTFS alternative data streams directly with a number of tools, such as streams.exe from http://www.microsoft.com/sysinternals/.

## Manage Execution Policy Through Group Policy

PowerShell's execution policy is configured through a simple REG_SZ registry value under HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell named "ExecutionPolicy". Set this to the name of the policy you want via some non-PowerShell script or tool, such as REG.EXE, REGEDIT.EXE, or VBScript.

For enterprise-wide management of the execution policy, though, it's best to download Microsoft's ADM template for PowerShell and import this into a Group Policy object. The title of this download is "Administrative Templates for Windows PowerShell" (for Google) and it was last seen at http://www.microsoft.com/downloads/details.aspx?familyid=2917A564-DBBC-4DA7-82C8-FE08B3EF4E6D&mg_id=10050&displaylang=en

Enclave Consulting LLC © 2017 - All Rights Reserved




## Digitally Signing Your Scripts

PowerShell's about_signing help file describes how to generate a code signing certificate with makecert.exe, but this is quite pointless. For regular users, it's best to set the option to "AllSigned" through Group Policy and sign all of your scripts with a certificate you have obtained from your own Certification Authority (CA). Through Group Policy you can make your domain members trust your CA. If you want to use Windows Certificate Services as your CA, it's built into Windows Server for free, and SANS has a course on it too (it's part of the Windows Track: SEC505.3). For your own use, set your policy to "RemoteSigned" or "Unrestricted" and then exercise the same self-control you already exercise for *everything* you download: don't run it if you don't trust it.

To sign your scripts, it's most convenient to have it done for you automatically with a code editor that supports this feature, e.g., Sapien PowerShell Studio. It can also be done from within PowerShell using the Set-AuthenticodeSignature cmdlet.

There is a script on the CD for this course to sign multiple scripts at once (Sign-Script.ps1). It includes an option to recurse through subdirectories to find and sign all *.ps1 files below a given folder path. You'll need to obtain a code-signing certificate first, the script will not create one for you.

```
sign-script.ps1 -path c:\folder\file*.ps1 -recurse
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Tips For Executing Commands

- **Please use tab completion!**
- **Use ".\" in front of scripts or EXEs in current folder.**
- **Press Esc or Enter to put blinking cursor on a new line.**
- **Open script in a new tab (ise .\script.ps1).**
- **Run selection button (or F8) in PowerShell ISE.**
- **Show or hide the Commands tab in ISE.**
- **Paste code into command shell window to run it.**
- **Command history arrow keys (up arrow).**
- **Piping ( | ) command output into the next command.**

SANS SEC505 | Securing Windows

# Tips For Executing Commands

Here are some tips for executing commands in PowerShell.

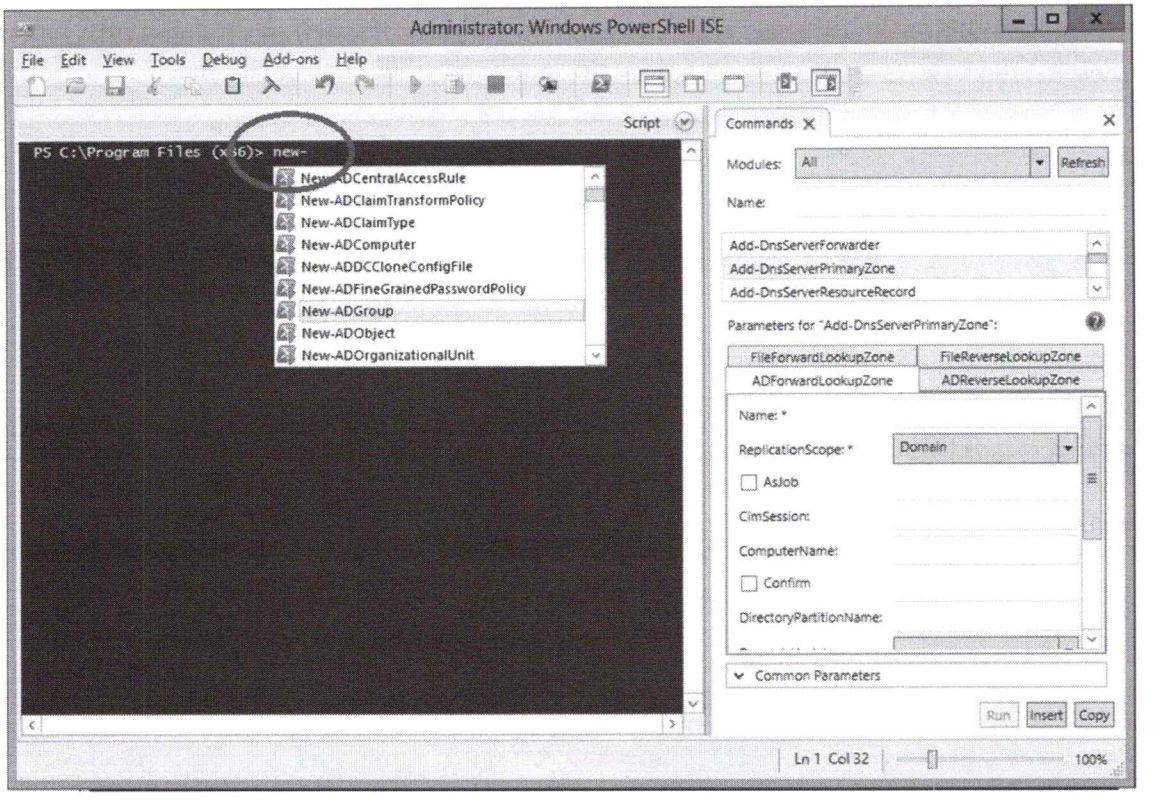
### Your New Best Friend: Tab Completion

The most important habit to learn in PowerShell is to use tab completion. When you begin to enter the first few letters of the name of a cmdlet, function, parameter, variable, file, folder, or almost anything else, PowerShell will guess what you are trying to type, and by pressing the Tab key you can see or select PowerShell's guesses. Many cmdlets and parameters have long names, so using tab completion is essential for speed, preventing typo mistakes, and saving your fingers from exhaustion.

> **Tip:** If you accidentally hit the Tab key too fast and go past what you wanted, hit Shift-Tab to go backwards to the prior options.

Tab completion is supported in the text-oriented powershell.exe console host, but only crudely: you have to the Tab key repeatedly to see matching options. In the graphical powershell_ise.exe host, on the other hand, you get a pop-up graphical window to see the list of options without hitting Tab a hundred times (this is similar to Visual Studio's IntelliSense).

Enclave Consulting LLC © 2017 - All Rights Reserved




## Put ".\" In Front Of Scripts or Executables In The Current Directory

The current directory is the folder you are in right now (also known as your "present directory"). The current folder path is displayed as part of your command prompt by default. The $pwd variable also shows you your current directory.

When you want to run a script or executable that is in the current directory, you must put ".\" in front of the script or executable name, such as with the following examples.

```
.\HelloWorld.ps1

.\setup.exe
```

When you use tab completion, PowerShell will put ".\" in front of your script or executable names for you, and show the file name extension too. This is another reason to use tab completion whenever possible.

Why is ".\" necessary? See the tips on command precedence below.

## Move The Blinking Cursor To A New Fresh Line (ISE)

In the graphical powershell_ise.exe host (ISE), your blinking cursor can be placed anywhere on-screen. This can be annoying when you click inside a command shell window and you want to type in a new command on a fresh blank line.

Enclave Consulting LLC © 2017 - All Rights Reserved

To get the blinking cursor back to a new blank line, just hit Esc or Enter. Thankfully, you don't have to use your mouse to place the cursor on a new line each time.

## Open Script In New Tab

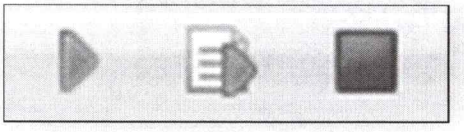
A fast way to open a text file in a new tab in PowerShell ISE is from the command line:

```
ise .\somescript.ps1

psedit .\somescript.ps1
```

## Run Selection In PowerShell ISE (F8)

In PowerShell ISE, if you open a script for editing, you can highlight one or more lines from that script and execute just those highlighted lines. Once the desired lines are highlighted, click the "Run Selection" button or press F8. The "Run Selection" button in the toolbar looks like a script with a green triangle on top of it.



Don't confuse this with the "Run Script" button just to the left, which looks like a big green triangle; the "Run Script" button (or F5) will run the entire script, not just the highlighted lines.

## Paste Code Into The Command Line

If you have one or multiple lines of PowerShell code copied into the clipboard that you'd like to execute, just paste those lines into a command shell window and hit Enter. To paste in ISE, right-click and select Paste, or just hit Ctrl-V.

If there are lines displayed in a command shell you would like to copy to the clipboard, drag out a rectangle around the lines with your mouse to highlight them, then right-click the highlighted area and select Copy (or hit Ctrl-C).

You can also pipe data into the clipboard like this:

```
dir | clip.exe

dir | set-clipboard
```

## The Commands Tab (ISE)

In the graphical PowerShell editor, pull down the View menu and make sure that "Show Command Add-On" is enabled. Then, on the right-hand side, you'll see a list of available

Enclave Consulting LLC © 2017 - All Rights Reserved

cmdlets. Double-click a cmdlet and then click the "Show Details" button to reveal a dialog box that guides you in entering arguments.

Notice too that in the Commands tab on the right side you can pull down the Modules menu, select a particular module (such as "PKI"), and it will show you all the cmdlets related to that module. Alternatively, by browsing the list of modules, you can see what categories of cmdlets are available. There are thousands of cmdlets available from Microsoft, so this browsing capability is very handy.

**Show-Command**
In PowerShell 3.0 and later, you can use the Show-Command cmdlet to display a graphical dialog box to help you enter the correct arguments. You can run your completed command (Run button) or just copy the command to the clipboard (Copy button). Any parameter name with an asterisk next to it ( * ) is a required value, and the Run button will not work until required arguments are entered (Copy still works though).

If you get an error while attempting to show a command which you know exists, then the module implementing that command has likely not been loaded. To see what modules are loaded, try running "show-command" by itself with no arguments.

To show help for entering the arguments to various cmdlets, try:

```
Show-Command Get-WmiObject

Show-Command Select-String

Show-Command Get-Service
```

## Command History (Up/Down Arrows)

By using the up or down arrow buttons, you can go backwards and forwards through your command history. You do not need to retype long complex commands from scratch each time you re-run a prior command.

There are also cmdlets just for your command history:

- Get-History
- Add-History
- Invoke-History

## Running Native Commands

A "native command" is just any command you used to do from within the CMD shell, including running batch scripts and binaries, launching graphical programs, executing VBScript files, etc. Native commands are just older, non-PowerShell-ish commands.

All of the following commands work in PowerShell just like they do in CMD:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
ipconfig.exe | findstr.exe /i "gateway"

netsh.exe firewall show config > outfile.txt

notepad.exe boot.ini

.\boot.ini

.\myvbscript.vbs >> outfile.txt

wscript.exe \\server\share\script.vbs -x -y -z

cmd.exe /c mybatchscript.bat

.\mybatchscript.bat
```

Just like in the CMD shell, you can also launch programs associated with filename extensions by simply typing the name of the file at the command line, e.g., typing ".\somepage.html" will open that file in your web browser.

However, when an executable creates its own interactive shell-like mode, like netsh.exe or nslookup.exe, that executable will still works in the text-oriented powershell.exe console host, but not in the graphical powershell_ise.exe (ISE) host.

To see what native commands must be run in powershell.exe, not ISE, run:

```
$psUnsupportedConsoleApplications
```

While you can launch a CMD batch script from within PowerShell, you actually cannot run that script *in* PowerShell. When you run a batch script, PowerShell launches CMD.EXE to do the actual interpretation, and then the output of that batch script, if any, is shown in PowerShell. If you want to run a single command in CMD and then immediately return to PowerShell, run "cmd.exe /c *command*".

## Command Precedence

When you execute a command without a full path to it, PowerShell searches for that command's name by trying to match it to one of these types of items, in this order: 1) aliases, 2) functions, 3) cmdlets, and then finally 4) native commands. Hence, if you execute "ping", PowerShell will first look for an alias named "ping", then a function named "ping", then a cmdlet, then finally a binary or script named "ping".

If two commands of the same type with the same name exist, such as two functions both named "ping", whichever was added most recently is what is executed. This is because the earlier one was replaced.

If there are multiple scripts or binaries with the same file name, but with different file name extensions (such as "ping.bat" and "ping.exe"), and they are in the same folder

Enclave Consulting LLC © 2017 - All Rights Reserved

(such as C:\Windows), and this folder is in the $env:PATH environment variable, then the $env:PATHEXT environment variable determines which file is executed. The $env:PATHEXT variable contains a list of file name extensions, ordered from most to least preferred. None of this matters, of course, if the full path to the script or binary is given when executing the command.

## What Version Of PowerShell Am I Running?

To see what version of PowerShell you are running now:

```
$psversiontable                    # See the PSVersion property.
```

## Running PowerShell Scripts From Shared Folders

You can run PowerShell scripts directly from shared folders like this:

```
powershell.exe \\server\share\script.ps1 -parameter "argument"
```

When a script is run from a shared folder, the script never touches the local hard drive, it is executed in memory only.

## Piping ( | ) and Redirection Operators ( >, >>)

When you want to feed the output of one command into a second command as input, use the pipe ("|") operator. In the following example, the output of Get-Process is fed as input to the Format-List cmdlet.

```
Get-Process | Format-List *
```

If you want to redirect (">" or ">>") the output of a command to a file, it'll usually work as expected, but it is better to use cmdlets like Out-File and Export-Csv instead. In general, avoid using the ">" or ">>" operators, which are just aliases anyway, and instead explicitly pipe your data into the desired cmdlet to save the output to a file.

```
Get-Service | Export-Csv -Path somefile.csv

Get-Process | Out-File -FilePath somefile.txt -Append
```

## Separate Multiple Commands On One Line With Semicolons

If you want to run multiple commands with a single line, just separate each command with a semicolon. A semicolon at the very end of a line or command is not required.

## Passing Complex Arguments To Native Commands

When passing arguments to native commands that include characters with special meaning to PowerShell (such as ', ", @, #, |, `, and $) then PowerShell can become confused and you'll get error. When this happens, try putting single-quotes ( ' ) around the arguments string. If the argument already contains a single-quote, then use two single-quotes (which is the not the same thing as one double-quote).

Enclave Consulting LLC © 2017 - All Rights Reserved

If that still doesn't work, then there is a special operator ( --% ) whose meaning can be described as "take every character after me literally and pass it as one uninterpreted string as the argument to the native command."

To pass a literal string as an argument for running a CMD shell native command:

```
find.exe --% /V "*prt.sys" %WinDir%\Inf\keyboard.inf
```

Also, see the help for the Start-Process cmdlet and its -ArgumentList parameter. If you have piping problems with native commands, see this cmdlet's other parameters too, such as -RedirectStandardInput, -RedirectStandardOutput, and -RedirectStandardError.

## Short 8.3 Names To Avoid Space Characters

It's best to always avoid using space characters in folder or file names. One way to do this is to use the "short 8.3 names" of folders and files that contain space characters. Short names are generated by Windows automatically for such folders and files.

To see the short 8.3 names in PowerShell:

```
cmd.exe /c "dir /x"
```

If you're running a batch script, shortcut or scheduled job to run a PowerShell script, and the path to the PowerShell script includes space characters, run it with the call ("&") operator, curly braces and single-quotes to help PowerShell to know that the path should be taken as a literal and should be executed as-is:

```
powershell -command "& {c:\'Program Files'\myscripts\Get-
FileHex.ps1 c:\autoexec.bat}"
```

Also, when creating a scheduled task to run a PowerShell script, use the full explicit path to the powershell.exe executable in the properties of the scheduled task.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Getting Help In PowerShell

```
get-help set-*
get-help *loc*      #Multiple wildcards are OK

get-help -full get-process
get-help -showwindow get-service

get-help about*     #List of tutorials
```

SANS SEC505 | Securing Windows

# Getting Help In PowerShell

Navigating through PowerShell can be daunting. Fortunately, there are many ways to get help from within PowerShell itself. The get-help cmdlet is your best friend. This is the tool to use to find cmdlets, explanations, examples, command-line syntax, aliases, and other help files.

To get help about the get-help cmdlet itself:

```
get-help
```

To see a list of all cmdlets, aliases, providers and help files:

```
get-help *
```

To see a listing of cmdlets that match a particular pattern:

```
get-help set-*

get-help *-item

get-help *loc*
```

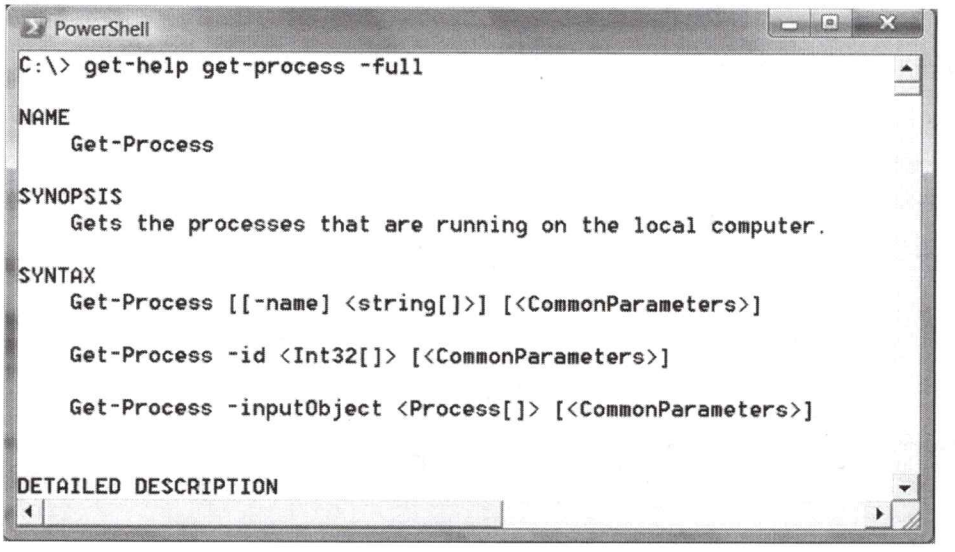
To see a summary of the help for a specific cmdlet, like the get-process cmdlet:

```
get-help get-process
```

Enclave Consulting LLC © 2017 - All Rights Reserved

To get more detailed help for a cmdlet, including parameter syntax and examples:

```
get-help get-process -full
```




To pop up a searchable graphical window to display the help text:

```
get-help get-process -showwindow
```

## Update Help

To update the local help files from Microsoft, if you have Internet access:

```
update-help -verbose
```

To update the local help files from Microsoft when the current machine does not have Internet access, see the details for the Update-Help and Save-Help cmdlets.

## HELP and MAN

If too much text scrolls by too fast when using get-help, you can increase your screen buffer height and scroll up with your mouse to see the text.

### *Try It Now!*

To increase your screen buffer size height in PowerShell.exe (not PowerShell_ISE.exe), right-click the title bar of your PowerShell window > Properties > Layout tab > set Screen Buffer Size Height (second from the top) to 1000 > OK. (If you are prompted to "Save properties for future windows with the same title", select that option > OK.) Notice that your PowerShell window has scrollbar on the right-hand side with which you can scroll up to prior output with your mouse. PowerShell_ISE includes this scrollbar already.

Enclave Consulting LLC © 2017 - All Rights Reserved

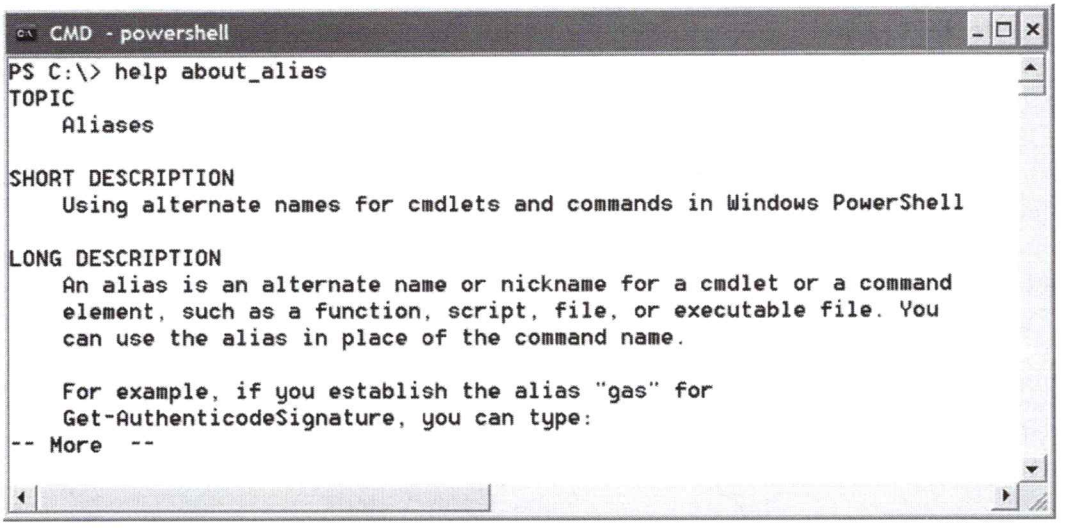
Or, instead of touching the mouse, if you want to see only one page of output at a time in the shell window, pipe the output through the `more` function:

```
get-help about_alias | more
```

In fact, there are built-in PowerShell functions (not cmdlets) named "help" and "man" which do exactly this; they pipe the output of get-help through `more` for you:

```
help about_alias

man about_alias
```




## Get-Help About_*HelpTopic*

PowerShell comes with over 50 plaintext help files on a variety of topics. They all have filenames like "about_*helptopic*.help.txt" and they are located under the shell's install directory for your language (e.g., $pshome\en-us\).

To see a listing of all help files:

```
get-help about*
```

To read a particular help file, such as for creating and using a function:

```
get-help about_function
```

Notice in the above that you do not need to add ".help.txt" to the end of the topic.

It's convenient to have one or multiple about files shown in separate windows so that you can alt-tab quickly among them. You don't have to open multiple instances of PowerShell or multiple tabs in ISE.

Enclave Consulting LLC © 2017 - All Rights Reserved

The following command will open a separate graphical window with the help text and it includes a search box too:

```
get-help about_wmi -showwindow
```

The following help topics are available and more are added with each new version of PowerShell Microsoft releases:

about_Aliases
about_Arrays
about_Automatic_Variables
about_Command_Precedence
about_Comment_Based_Help
about_Comparison_Operators
about_Core_Commands
about_Debuggers
about_Do
about_Escape_Characters
about_Execution_Policies
about_Foreach
about_Functions
about_Functions_Advanced_Methods
about_Functions_CmdletBindingAttribute
about_Group_Policy_Settings
about_History
about_Jobs
about_Join
about_Language_Modes
about_Locations
about_Methods
about_Objects
about_Operators
about_Parameters
about_Parsing
about_Pipelines
about_PowerShell_Ise.exe
about_Profiles
about_Properties
about_PSSessions
about_PSSnapins
about_Redirection
about_Regular_Expressions
about_Remote_Disconnected_Sessions
about_Remote_Jobs
about_Remote_Requirements
about_Remote_Variables
about_Arithmetic_Operators
about_Assignment_Operators
about_Break
about_Command_Syntax
about_CommonParameters
about_Continue
about_Data_Sections
about_DesiredStateConfiguration
about_Environment_Variables
about_Eventlogs
about_For
about_Format.ps1xml
about_Functions_Advanced
about_Functions_Advanced_Parameters
about_Functions_OutputTypeAttribute
about_Hash_Tables
about_If
about_Job_Details
about_Language_Keywords
about_Line_Editing
about_Logical_Operators
about_Modules
about_Object_Creation
about_Operator_Precedence
about_Parameters_Default_Values
about_Path_Syntax
about_PowerShell.exe
about_Preference_Variables
about_Prompts
about_Providers
about_PSSession_Details
about_Quoting_Rules
about_Ref
about_Remote
about_Remote_FAQ
about_Remote_Output
about_Remote_Troubleshooting
about_Requires

Enclave Consulting LLC © 2017 - All Rights Reserved

about_Reserved_Words
about_Run_With_PowerShell
about_Scripts
about_Script_Internationalization
about_Session_Configuration_Files
about_Special_Characters
about_Split
about_Throw
about_Trap
about_Types.ps1xml
about_Updatable_Help
about_While
about_Windows_PowerShell_4.0
about_Windows_RT
about_Wmi_Cmdlets
about_WS-Management_Cmdlets
about_Checkpoint-Workflow
about_InlineScript
about_Sequence
about_WorkflowCommonParameters
about_Scheduled_Jobs
about_Scheduled_Jobs_Basics
about_CIMSession
about_IntelNetcmdlets
about_Return
about_Scopes
about_Script_Blocks
about_Session_Configurations
about_Signing
about_Splatting
about_Switch
about_Transactions
about_Try_Catch_Finally
about_Type_Operators
about_Variables
about_Wildcards
about_Windows_PowerShell_ISE
about_WMI
about_WQL
about_ActivityCommonParameters
about_Foreach-Parallel
about_Parallel
about_Suspend-Workflow
about_Workflows
about_Scheduled_Jobs_Advanced
about_Scheduled_Jobs_Troubleshooting
about_BITS_Cmdlets

Enclave Consulting LLC © 2017 - All Rights Reserved

## Aliases

| Alias | Cmdlet |
|---|---|
| dir | get-childitem |
| ls | get-childitem |
| cls | clear-host |
| echo | write-output |
| type | get-content |
| del | remove-item |
| ps | get-process |
| cd | set-location |
| pwd | get-location |
| sort | sort-object |

```
get-alias  #All aliases
get-alias ps

new-alias nn notepad.exe

help about_alias
```

SANS SEC505 | Securing Windows

# Aliases

An "alias" is an alternative name for a frequently-used cmdlet, function, script, file, or executable. Aliases can be executed or invoked just like their targets. Aliases have shorter names than their targets, making them quicker to type when working interactively in the shell, and many aliases exist to ease the transition from other shells (CMD or UNIX) into PowerShell.

Some of the more common aliases you'll encounter are the following:

| ALIAS | TARGET CMDLET |
|---|---|
| echo | Write-Output |
| write | Write-Output |
| type | Get-Content |
| cat | Get-Content |
| cls | Clear-Host |
| clear | Clear-Host |
| copy | Copy-Item |
| cp | Copy-Item |
| del | Remove-Item |
| rm | Remove-Item |
| dir | Get-ChildItem |
| ls | Get-ChildItem |
| % | ForEach-Object |
| where | Where-Object |
| ? | Where-Object |

Enclave Consulting LLC © 2017 - All Rights Reserved

| | |
|---|---|
| kill | Stop-Process |
| ps | Get-Process |
| cd | Set-Location |
| pwd | Get-Location |
| gm | Get-Member |
| select | Select-Object |
| sort | Sort-Object |

To see all the aliases defined in the current shell:

```
get-alias *
```

To see how a particular alias, like "cd", is mapped:

```
get-alias cd
```

To create a new alias for `notepad.exe` named "nn":

```
new-alias nn notepad.exe
```

To change an existing alias to point towards some other target:

```
set-alias nn netsh.exe
```

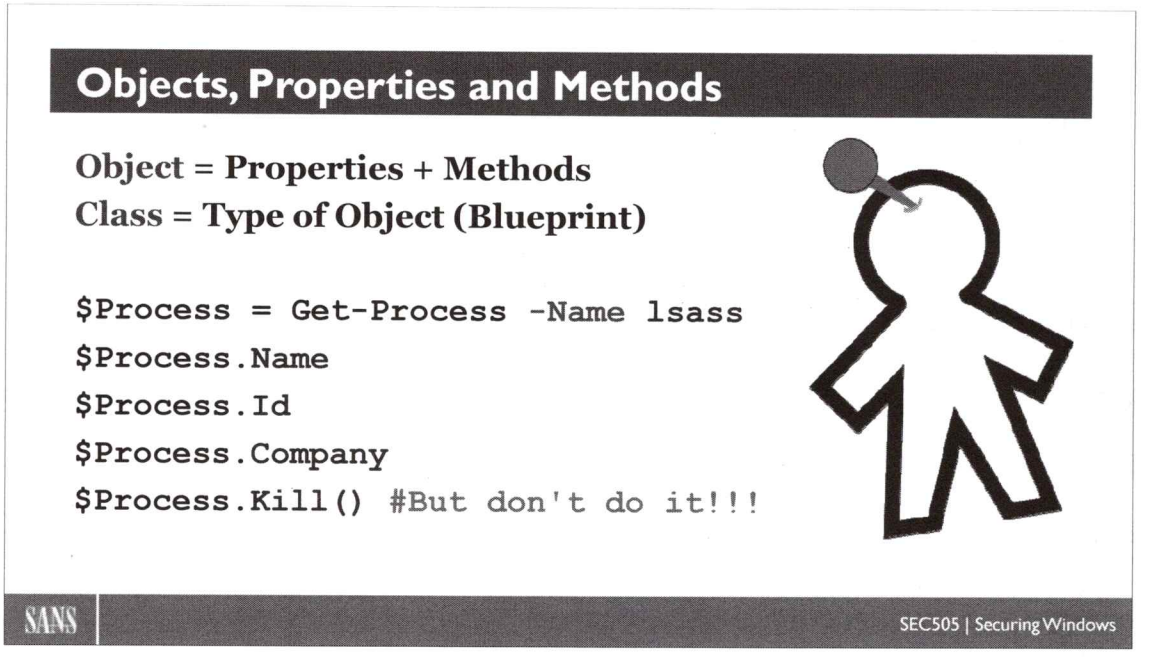
To read more about aliases and how to manage them:

```
help about_alias
```

## Alias Changes Are Not Permanent (Add Them To Your Profile)

Note, however, that any newly created aliases will not survive the current PowerShell session. Once you close PowerShell, the aliases are gone. To make new aliases permanent, you must add them to your profile script. PowerShell profiles will be discussed in a different section.

Enclave Consulting LLC © 2017 - All Rights Reserved




# Objects, Properties and Methods

In a CMD shell, the ipconfig.exe tool produces textual output and you see that text scroll by in the CMD shell window. When dealing with text, what you see is what there is, there's nothing else besides the text itself, there's no "hidden text" underneath the characters you see on the screen or print out as hardcopy. PowerShell can produce, pipe, consume and manipulate text, to be sure, but it's not what it's designed primarily to do. PowerShell is designed primarily to handle *objects*.

## What's An Object?

An "object" is a programming construct that is analogous to our everyday commonsense understanding of physical objects, such as dogs, microwave ovens, galaxies, and so on. All these physical objects (like microwave ovens) have properties (such as color and weight) and they do things (like make beeping sounds or boil hot chocolate). In our grammar and cognitive psychology, this is how we conceive of the universe and everything in it: Every property is a property *of* some object, and every action is an action *performed by* some object. But just *what* is an object? An object is nothing but all of its properties and potentials for action bound together into a single unit persisting through space and time.

In programming, then, an object is a named area of memory containing data: some of this data is executable code, some of this data is passive/non-executable information. All of the properties or attributes of the object are the passive/non-executable chunks of data, while all the actions the object can do are the different pieces of executable code. "Attribute" is a synonym for "property", and "an action the object can do" is called a "method" of that object (think "method actor"). All the properties and methods of an

Enclave Consulting LLC © 2017 - All Rights Reserved

object are collectively called the "members" of that object (think "dismember an object", i.e., to cut off its properties and methods).

## How Do I Use Objects?

So imagine you've somehow got an area of memory containing a collection of properties and some code-chunks (methods). To work with that object, it's assigned a name, and using that name in your script allows you to access its properties and to invoke/call/execute its methods.

Let's get an object that represents the LSASS.EXE process:

```
$Process = Get-Process -Name lsass
```

In the above command, the Get-Process cmdlet can create objects representing processes. In this case, we gave the cmdlet the name of a particular process (lsass) and the output object was placed inside the $Process variable. $Process is like a holder or pointer towards an object in memory; we can't see that object in memory, but we can see the variable ($Process) in our command shell and in a script. If we could see the object in memory, it would just be a huge collections of ones and zeros, so having a variable named after the after ($Process) is much more human-friendly.

When you have plain text, what you see is what you get, because what you see is all there is! With an object, on the other hand, you cannot see the object itself, and that object has all these mysterious properties and methods. To get at a particular property, you have to know its name. To invoke a method, you have to know the name of that method, and maybe even what argument(s) should be given to it.

For example, an object representing a process has lots of properties:

```
$Process.Name
$Process.Id
$Process.Company
$Process.Description
$Process.StartTime
$Process.VirtualMemorySize
$Process.Modules   #This property has lots of objects inside it!
```

**Warning!** Don't run the following command!

And a process object also has an interesting and dangerous method:

```
$Process.Kill()       #But don't do it!!!
```

Notice the use of the period symbol (".") in the above examples? Typically, to the left of the period is the name of an object, and to the right of the period a property name or method name. We can extract or update properties, and methods can be executed or run

Enclave Consulting LLC © 2017 - All Rights Reserved

by "calling" or "invoking" them by name. A property name is usually a noun or adjective, while a method name is usually a verb followed by a pair of parentheses.

In general, methods can take zero, one or many arguments separated by commas. It's up to the original designer of the object to decide how many.

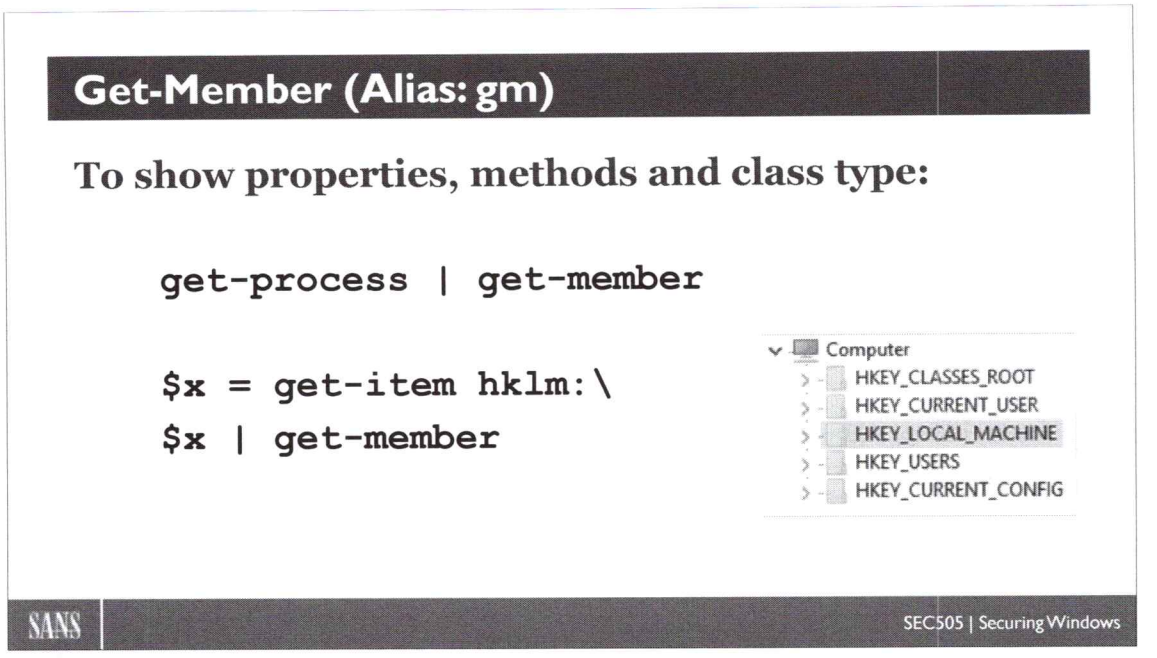
## How Does An Object Get Its Properties And Methods?

What determines which properties and methods comprise an object? For the most part, the "type" or "class" of an object determines what properties and methods it has. By analogy, an architect creates a blueprint and this blueprint could be used to build thousands of identical houses. The blueprint is the "type definition" or "class definition", and each physical house is a specific instance of that type/class of house. Why does a particular house have just the properties and methods (like *Lean*, *Creak*, *OverHeat*, *Collapse*, etc.) that it does? Because that house has a type, or is "of a class", which includes those properties and methods. Almost all the objects you'll use in PowerShell are instances of type definitions from the .NET Framework's class library, Windows' COM classes, or the many classes in WMI.

## What Objects Exist? What Are Their Properties And Methods?

How can you figure out what types of objects exist? The easiest way is to look at other peoples' scripts, see what objects they're using, then read more about these objects on the Internet and in the Software Developer Kits (SDKs) you've downloaded from Microsoft. How can you figure out what properties and methods a given type of object possesses? Again, the easiest way is to look at other peoples' scripts, see what properties and methods are used, then read more about it on the Internet and in the SDKs you've downloaded from Microsoft. On the other hand, there are cmdlets built into PowerShell itself which are designed to help you do just this!

Enclave Consulting LLC © 2017 - All Rights Reserved




# Get-Member (Alias: gm)

A very useful cmdlet when you need help finding the properties and methods of an object is get-member. Every type or *class* of object in .NET has enumerable properties and/or methods which can be accessed by PowerShell. The properties and methods of an object are called the "members" of that object. For example, the get-process cmdlet will emit a stream of objects, each representing a running process on the computer. Each of these process objects is of type *System.Diagnostics.Process*, as defined in the .NET class library.

The get-member cmdlet will show you 1) the type or class name of the first object piped into it and 2) the names of all the properties and methods of that object. The type name is shown at the very top of the output of get-member.

To show the properties and methods of *System.Diagnostics.Process* objects:

```
get-process | get-member
```

If you have captured the output of a command to a variable, but you don't know what kind of data it is or what its members are, pipe that variable into get-member:

```
$x = get-item hklm:\

$x | get-member
```

Another way to use get-member is to specify the object to be queried with the -inputobject parameter instead of piping into the cmdlet. When a collection or array is

Enclave Consulting LLC © 2017 - All Rights Reserved

piped into get-member, the members of the first item inside the collection or array is shown. But if a collection or array is specified using the -inputobject parameter, the members of the collection or array itself is shown, not the members of an item inside it.

Also, when piping into get-member, if different types of objects are piped in, the members of each distinct type of object are shown, but not the members of every single instance; for example, get-childitem might return fifty files and twenty folders, but, if the output is piped into get-member, only two things are shown: the members of file-type objects and the members of folder-type objects (you don't get the members of seventy things).

To see the members of the first item in an array and then see the members of the array itself (an array of objects is of type "`System.Object[]`"):

```
$output = dir c:\

$output | get-member                #Members of first item in array.

get-member -inputobject $output  #Members of the array itself.
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Drives

**"Drives" are more than just disk volumes:**

- `get-psdrive`

**On Your Computer:**

- `dir hkcu:\`
- `dir variable:\`
- `cd cert:\localmachine\ca`

SANS SEC505 | Securing Windows

# Drives

We all know that a hard drive "really" only has sectors, clusters and partitions, but the operating system allows us to access the drive's contents by assigning a drive letter to each volume and a name to each folder and file. Hence, the operating system organizes file system data in a certain hierarchical way which is easy to understand and manipulate. Windows and UNIX then both come with tools that are designed around the easy manipulation of file systems, which is better than forcing users to directly manipulate the sectors and clusters that "really" exist.

The important thing about a file system is that it is a hierarchical set of containers in which each container (directory) can contain discrete objects (files) or more subcontainers (subdirectories). Because the structure is hierarchical, we can identify the path to the location where we currently are, change locations, drill down recursively from one container through all of its subcontainers, and otherwise "move around" in that nested set of containers. And in each container we can store useful items, like documents and programs. But notice that the registry is also a set of hierarchical containers (keys) which hold items (values) and subcontainers (subkeys). And note that Active Directory is also a set of containers (organizational units) which holds items (users, computers, etc.) and other subcontainers (sub-OUs). Can you recursively search the registry? Yes. Can you move a user from one OU to another in Active Directory? Easily.

So here's the Big Idea: Just as the operating system makes file systems available to `CMD.EXE` as drives, folders and files, why can't PowerShell make the registry, certificate stores, Active Directory, variable collections, SQL Server, and other hierarchical sets of containers with items inside them, including the regular file system itself, available as abstract "drives" with "containers" and "items"? Well, it can!

Enclave Consulting LLC © 2017 - All Rights Reserved

Try these commands (and remember that "dir" is an alias for the get-childitem cmdlet):

```
# Example: Drives.ps1

dir c:\
dir hklm:\
dir hkcu:\
dir variable:\
dir alias:\
dir cert:\
dir env:\
dir function:\
```

A "drive" is just the name of a top-level container that can hold named subcontainers, such as folders, keys and organizational units. An "item" is just a non-container object of some type, such as a file, registry value, or user account. Your "location" in a set of nested containers determines how you access other containers and items, i.e., are those other things "above" you, "below" you, in the "current location", etc.? Very often, PowerShell makes the things you want to manage available to you as items in containers, with the top-level container named as a "drive".

A "provider" is a .NET assembly compiled as a DLL and loaded into PowerShell for the purpose of making the things you want to manage available as a drive. A PowerShell drive can be made available through resources other than providers too.

To see your currently installed providers and their corresponding drive names:

```
get-psprovider
```

To see your currently available drives:

```
get-psdrive
```

The standard PowerShell providers and drive names are:

| Provider Name | Drive(s) | Description |
|---|---|---|
| Alias | alias:\ | PowerShell aliases. |
| Environment | env:\ | Environmental variables. |
| FileSystem | C:\, D:\, E:\, … | Standard file systems. |
| Function | function:\ | PowerShell functions. |
| Registry | hklm:\, hkcu:\ | Windows registry. |
| Variable | variable:\ | PowerShell variables. |
| Certificate | cert:\ | Digital certificate stores. |

An important design goal of PowerShell is to have a single set of cmdlets which could be used with all or most providers. If both the file system and the registry can be viewed as

Enclave Consulting LLC © 2017 - All Rights Reserved

a collection of items stored in various nested containers, why have different toolsets for the file system and the registry? PowerShell tries to avoid duplicating toolsets whenever possible or practical (but it's not always practical). And why have obscure or difficult-to-learn names for these tools? In PowerShell, most cmdlets follow a rational *verb-noun* naming standard, like "move-item" or "sort-object", with optional shorthand aliases for these as well.

> **Note:** When listing your providers with get-psprovider, the "ShouldProcess" capability means that the -confirm switch can be used with associated cmdlets, while the "Filter" capability means the provider supports wildcard searches.

Cmdlets for moving around in drives of any type, from any provider, with full support for relative path addressing (using "." and "..") and tab completion:

- get-location (aliases: gl, $pwd)
- set-location (aliases: sl, cd, chdir)

```
get-location

$pwd

set-location hkcu:\software\microsoft\windows\

cd env:\

C:
```

> **Note:** You might wonder why you can just enter "C:" instead of "set-location C:" to switch to the C: drive. It's because "C:" is actually the name of a built-in function that executes set-location for you (run "dir function:\" to see it).

Cmdlets for manipulating items in drive containers:

- new-item (alias: ni, but see also: mkdir)
- remove-item (aliases: ri, del, rm, rd, rmdir)
- move-item (aliases: mi, move, mv)
- copy-item (aliases: cpi, copy, cp)
- rename-item (aliases: rni, rn, ren)

```
# Example: Working_With_Items.ps1

new-item $home\somefile.txt -type file

new-item $home\somefolder -type directory

new-item hkcu:\software\somekey -type key
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
new-item hkcu:\software\somekey\somevalue -type value

del hkcu:\software\somekey\somevalue

ren hkcu:\software\somekey otherkey
```

Cmdlets for accessing the properties of an item or multiple items:

- get-item (alias: gi)
- get-childitem (aliases: gci, dir, ls)
- set-item (alias: si)
- clear-item (alias: ci)

```
get-item env:\systemroot

get-item function:\more

get-item c:\windows | get-member

get-childitem c:\windows\*.exe

get-childitem \\localhost\c$

dir cert:\currentuser\root

set-item variable:\fishtype -value "Trout!"

clear-item variable:\fishtype
```

Cmdlets for accessing an individual property of an item:

- get-itemproperty (alias: gp)
- set-itemproperty (alias: sp)
- clear-itemproperty (alias: clp)
- rename-itemproperty (alias: rnp)
- move-itemproperty (alias: mp)
- remove-itemproperty (alias: rp)
- copy-itemproperty (alias: cpp)

```
get-itemproperty $home | format-list *

get-itemproperty $home -name creationtime

set-itemproperty $home -name creationtime "4/18/2017"
```

Cmdlets specifically for working with strings and the contents of files:

Enclave Consulting LLC © 2017 - All Rights Reserved

- get-content (aliases: gc, type, cat)
- set-content (alias: sc)
- add-content (alias: ac)
- clear-content (alias: clc)

```
get-content $env:windir\inf\volsnap.inf

get-content variable:\home

get-content .\somefile.txt -wait        #Similar to 'TAIL.EXE -F'

get-service | set-content c:\services.txt
```

## Creating New PowerShell Drives

You can use a provider to create a new "drive" rooted at a location of your choice. Using the new-psdrive cmdlet, you specify which built-in provider you want to use, the name of your new drive, and the path to the desired container or object which will be the top or "root" of that drive.

To map the "share:" drive to the "\\localhost\c$" UNC network path:

```
new-psdrive -name share -psprovider filesystem
   -root \\localhost\c$

cd share:
```

Enclave Consulting LLC © 2017 - All Rights Reserved

**Environment Variables**

```
dir env:\

$env:PATH
$env:SYSTEMROOT

new-item env:\VERYABLE -value "Flash Gordon"
set-item env:\VERYABLE -value "Wonder Woman"
remove-item env:\VERYABLE
```

SANS SEC505 | Securing Windows

# Environment Variables (ENV:)

In a CMD shell, you can see all your environment variables by running the "set" command, and you can use a variable's contents, like %SYSTEMROOT%, by simply enclosing the variable's name in percentage signs. It's different in PowerShell.

In PowerShell, to see all your environment variables:

```
get-childitem env:\

dir env:\
```

To see a single variable whose name you know, like PATH or SYSTEMROOT:

```
$env:PATH

$env:SYSTEMROOT
```

> **Note:** In both CMD and PowerShell, environment variables can be in uppercase or lowercase. Also, the backslash isn't necessary when manipulating environment variables, but it's been added for consistency with the other providers.

To create a new environment variable:

```
new-item env:\VERYABLE -value "Flash Gordon"
```

To change an environment variable that already exists:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
set-item env:\VERYABLE -value "Wonder Woman"
```

To delete an environment variable:

```
remove-item env:\VERYABLE
```

To use an environment variable in a string you are constructing:

```
$x = "My user name is $env:USERNAME, and I'm free!"
```

If you try to use an environment variable in your scripts surrounded by just percentage sign characters, you'll get an error message. The variable name must be preceded with "$env:" and should not include percentage signs or a backslash.

## Environmental Variable Changes Are Not Permanent

Similar to aliases, changes to environment variables do not persist from one PowerShell invocation to the next. If you exit PowerShell, all changes to these variables are lost. If you want to make permanent changes to environment variables, modify your PowerShell profile. PowerShell profiles will be discussed later.

## Built-In PowerShell Variables

While not exactly environment variables, there are other variables built into PowerShell that you see very frequently. This seems as good a place as any to list them.

To see a list and description of the variables automatically created in PowerShell:

```
help about_automatic_variables
```

Some of the more commonly-used automatic variables are the following:

- **$?** -- Contains $true if last operation succeeded, $false otherwise.
- **$_** -- Contains the current object in a pipeline.
- **$args** -- Contains an array of the parameters passed to a function or script.
- **$false** -- Contains the boolean value of False ([Int] 0, or $null).
- **$home** -- Contains the user's home directory path.
- **$input** -- Contains objects piped into a function, filter, or script.
- **$lastexitcode** -- Contains the error code of the last Win32 executable execution.

Enclave Consulting LLC © 2017 - All Rights Reserved

- **$matches** -- Contains matches to a regular expression search.
- **$null** -- Contains nothing and cannot have contents.
- **$pshome** -- Contains the directory path where PowerShell is installed.
- **$profile** -- Full path to your profile script.
- **$OFS** -- Output Field Separator, used as the default delimiter when converting arrays to strings, and is a single space character by default.
- **$true** -- Contains the boolean value of True ([Int] 1, or -not $null).

Enclave Consulting LLC © 2017 - All Rights Reserved

## Your Profile Script(s)

**Your own functions, aliases and variables in memory are temporary, they don't survive the closing of the PowerShell host process.**

**$Profile scripts run automatically at launch.**

**On Your Computer:**

- `ise $Profile.CurrentUserAllHosts`

SANS SEC505 | Securing Windows

# Your Profile Script(s)

Your own functions, aliases, variables in memory will exist only in the current session, hence, if you close PowerShell and open it again, these items are lost. By adding them to your profile, however, they can be reloaded every time you open PowerShell.

A PowerShell "profile" is a set of scripts which run automatically every time PowerShell is opened. None of these scripts exist by default, but they can be created. The paths to these scripts are contained in the properties of a special variable named "$profile":

```
$profile | format-list * -force
```

The $profile variable has the following properties, each of which contains a script path, and, if all of these scripts exist, this is also the order in which they are executed:

```
$profile.AllUsersAllHosts                #Run first.

$profile.AllUsersCurrentHost             #Run second.

$profile.CurrentUserAllHosts             #Run third.

$profile.CurrentUserCurrentHost          #Run last.
```

Note that the CurrentUserCurrentHost property is the default for the $profile variable, so these two commands will produce the same output:

```
$profile.CurrentUserCurrentHost
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
$profile
```

The "AllUsers" part refers to scripts which would be run by any user who logs onto the computer and runs PowerShell. The "CurrentUser" part refers to scripts which apply only to the user currently logged on and will have a different path for each user. But what about "CurrentHost" and "AllHosts"? What is a PowerShell "host" anyway?

Strictly speaking, PowerShell is a script interpretation and execution engine, not a command shell, hence, this engine can be connected to graphical programs, text-oriented console programs, or programs that do not expose an interface user at all. When you run `powershell.exe`, you are running a text-only console host process for the PowerShell engine. When you run `powershell_ise.exe` (perhaps using the "ise" alias), you are running a graphical Windows application, but this application can have multiple PowerShell engines hosted inside it.

Information about the hosting process for the PowerShell engine in front of you right now can be displayed by using the built-in $host variable:

```
$host

$host.UI.RawUI
```

Because the above profile scripts are run in a specific order, later scripts can override the changes made by the prior scripts. This permits administrators to define global defaults and allow certain users to override these defaults if necessary. As always, keep in mind that Group Policy can be used to create, delete or overwrite any of these scripts.

Incidentally, it's easy to miss something when running the console PowerShell versus the graphical ISE: their profile scripts have slightly different names (look for "ISE_"):

```
# When running the console POWERSHELL.EXE:
$profile.CurrentUserCurrentHost
...\WindowsPowerShell\Microsoft.PowerShell_profile.ps1

# When running the graphical POWERSHELL_ISE.EXE:
$profile.CurrentUserCurrentHost
...\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1
```

So, which profile script should you edit for your own personal use? Unless you want different settings for the console and the graphical ISE PowerShell hosts, it's easiest to just have one profile script for both hosts. In fact, you could have just one script, detect which host is executing it, then define different settings per host anyway. In other words, start by editing the $profile.CurrentUserAllHosts script, knowing that you can always change your mind later, perhaps after installing some new PowerShell host application. This is the script normally located here:

Enclave Consulting LLC © 2017 - All Rights Reserved

C:\Users\<*YourName*>\Documents\WindowsPowerShell\profile.ps1

Hence, a quick way to create and edit your all-hosts profile script would be to run:

```
# Example: Edit-Profile.ps1

if (-not $(test-path $profile.CurrentUserAllHosts))
{
new-item -path $profile.CurrentUserAllHosts -itemtype file -force
}

PowerShell_ISE.exe $profile.CurrentUserAllHosts
```

The -force switch for the new-item cmdlet above will create the file, even if the folder path does not yet exist; it will create the necessary folders first in order to create the file.

Once your profile script exists, you can place any commands you wish into it. It is executed like any other PowerShell script when you open PowerShell, except that it is run automatically.

Enclave Consulting LLC © 2017 - All Rights Reserved

## On Your Computer

How do you find cmdlets related to services from the command line?

How do you find cmdlets related to services in the graphical PowerShell ISE editor? (Hint: see the View menu.)

How do you get Windows service objects?

How do you view the properties and methods of service objects? (Hint: use piping.)

What command will show what "gm" is an alias for?

Enclave Consulting LLC © 2017 - All Rights Reserved

How do you open your personal profile script for all PowerShell host processes in a new tab in the PowerShell ISE editor? (Hint: there is an editor alias and a built-in variable.)

How do you view a list of the current "drives" available in PowerShell?

How do you view the PATH environment variable?

Enclave Consulting LLC © 2017 - All Rights Reserved

## Cheat Sheet

Find cmdlets related to services from the command line:

```
get-help service
```

Find cmdlets related to services in the graphical PowerShell ISE editor by pulling down the View menu, select Show Command Add-On, then, on the Commands tab to the right, enter the word "service" in the Name text box.

Get objects representing Windows background services:

```
get-service
```

View the properties and methods of service objects:

```
get-service | get-member
```

What is "gm" an alias for?

```
get-alias gm
```

Open your main PowerShell profile script in a new PowerShell ISE tab:

```
ise $profile.CurrentUserAllHosts
```

View a list of the current "drives" available in PowerShell:

```
get-psdrive
```

View the PATH environment variable:

```
$env:path
```

## Finished Already?

The -Join operator can join together multiple strings into one long string using your choice of delimeter, such as a colon (":"), like this:

```
"Date","User","Computer" -Join ":"
```

There is also a -Split operator which can break apart a string using your choice of delimeter. How do you split the semicolon-delimited PATH environment variable?

Enclave Consulting LLC © 2017 - All Rights Reserved

## Today's Agenda

1. Setting Up Our Training VM
2. PowerShell Overview & Tips
3. Modules, Functions & PowerShell Remoting
4. Essential Commands & Scripting Examples
5. Writing Your Own Functions & Scripts

SANS SEC505 | Securing Windows

# Today's Agenda

Now that you're comfortable working in the shell, it's time to start digging into the details of running cmdlets, functions, scripts and native commands.

Enclave Consulting LLC © 2017 - All Rights Reserved

**Running Functions And Dot-Sourcing Scripts**

**Function = Just a block of code with a name.**

**To see your available functions:**

```
dir function:\
get-content function:\mkdir
```

**To "dot-source" functions from a script:**

```
. C:\folder\library.ps1
. \\server\share\library.ps1
. .\libraryscript.ps1
```

SANS SEC505 | Securing Windows

# Running Functions and Dot-Sourcing Scripts

A function is a set of PowerShell script statements which have been assigned a name and which can accept zero or more parameters as input. You can use a function just like a cmdlet, including placing functions in the middle of a pipeline (if the function is designed for this). Some functions are builtin, whereas others will be loaded by you.

To see a list of the functions available in the current shell:

```
get-childitem function:\

dir function:\
```

As you can see from the list of functions, help, more and even C: are all functions. What does the "C:" function do? It simply executes "Set-Location C:".

To see the contents of a function, such as the mkdir function:

```
get-content function:\mkdir
```

How to write your own functions will be discussed later.

## Dot-Sourcing Function Libraries

If a function is in a script, then of course your script can use that function, but how can you make a function available for interactive use inside PowerShell other than running a script which contains that function? One way to load functions into PowerShell's

Enclave Consulting LLC © 2017 - All Rights Reserved

function:\ drive, but not to run them at the same time, is to "dot-source" the script containing the functions. Dot-sourcing a script executes the lines in it in the current scope just as though you had entered each line by hand. If the script only contains function definitions and no commands, the functions are simply copied to the function:\ drive without executing any of them.

To dot-source a script containing functions you want to load into the function:\ drive:

```
. c:\folder\library.ps1

. .\libraryscript.ps1

. \\server\share\script.ps1
```

As you can see from the above example, to dot-source a script is to execute a single period followed by one space character and then the path to the script. That's it! It's also customary, but not required, to put the word "library" somewhere in the name of a script that has only functions and no commands in it, e.g., "sqlserver.library.v.2.3.ps1".

Once dot-sourced, you'll see the functions loaded ("dir function:\") but only for the current session. If you close PowerShell and open a new one, the function script will have to be dot-sourced again. If you want the dot-sourced functions to be available in your shell every time, you can add them to your PowerShell profile ("notepad.exe $profile").

## Overlapping Names

What happens when two scripts are dot-sourced and they both contain functions with the same names? The last script to be dot-sourced wins. The function drive cannot have multiple functions with the same names, so the last change made to a function is the change that sticks.

On the other hand, what if an alias, a function, a cmdlet and a native command all have the same name, such as "Ping"? Which will be executed when "Ping" is entered? PowerShell follows a search procedure when the name of command is entered where the first match to the name wins. When a command is entered, PowerShell searches for the command name in the following locations: 1) the Alias:\ drive, 2) the Function:\ drive, 3) the list of cmdlets currently loaded, and finally 4) the folders listed in the $env:PATH environment variable. Hence, an alias named "Ping" would take precedence over a function named "Ping", which would take precedence over a cmdlet named "Ping", and the ping.exe binary itself would be selected last.

Enclave Consulting LLC © 2017 - All Rights Reserved

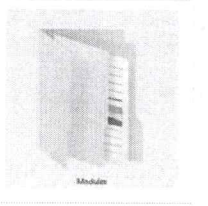
## Running Cmdlets And Loading Modules

### What is a Module?

- A folder with scripts, DLLs, XML, and other files.
- Provides cmdlets, functions, drives, and variables.
- Where does PowerShell look for these folders?

```
$env:PsModulePath -Split ";"

Get-Module -ListAvailable
Import-Module -Name AppLocker
Get-Command -Module AppLocker
```



SANS SEC505 | Securing Windows

# Running Cmdlets and Loading Modules

Another type of command which can be executed in PowerShell is a cmdlet (pronounced "command-let"). Get-Process is an example of a cmdlet. Most commands unique to PowerShell are either cmdlets or scripts/functions which run cmdlets. A cmdlet is almost always compiled into a .NET binary DLL. Think of a cmdlet as a compiled function, but it's not in the Function:\ drive, it's usually a DLL that is part of a module.

## Modules

A "module" is one or more files which implement a set of cmdlets, functions, variables, aliases, provider drives, and other related resources. Modules are typically organized into folders, where the name of each folder is identical to the name of the module. In general, modules are a convenient way to organize code into manageable units.

The primary file in a module can be a PowerShell script (.psm1), a compiled binary (.dll), or a manifest file (.psd1). A manifest specifies all of that module's component files, such as .psm1 scripts, .dll binaries, XML type definitions (.ps1xml), graphics, videos, audio files, database files, etc.

At a minimum, a module that actually does something must be composed of at least one PowerShell script (.psm1) or at least one compiled binary (.dll).

A module folder can simply be copied to another computer in order to use that module. A module does not have to be "installed" like a standard Windows program, just copied into the appropriate parent folder. What folder is that?

Enclave Consulting LLC © 2017 - All Rights Reserved

To see the default folders where PowerShell searches for modules:

```
$env:PsModulePath

$env:PsModulePath -Split ";"        #To see the order more easily.
```

When looking for a module, PowerShell first looks in the user's own local profile directory, then afterwards in system-wide locations. This means that users do not have to be members of the local Administrators group in order to use or create modules of their own. Modifying a system-wide module does require administrative privileges though.

Modules can have unique GUIDs and version numbers in their .psd1 manifest files for the sake of version control and patch management, but this is not required. Module scripts and binaries can be digitally signed too.

## Using Modules

When a module is imported or "loaded", the cmdlets, functions, aliases, provider drives and variables exported from that module become available for use. One module can load another module in a parent-child relationship; the first module loaded in a chain of modules is called the "root module", while the other modules loaded by the root or other child modules are called "nested modules".

To see which parent or "root" modules are currently loaded:

```
# Example: Modules.ps1

get-module
```

To see all currently loaded modules, including the nested modules:

```
get-module -all
```

To see what root modules are available for use, loaded or otherwise:

```
get-module -listavailable
```

To see all available root and nested modules, including their folder paths:

```
get-module -listavailable -all
```

To load or import a module to make its cmdlets and functions available for use:

```
import-module -name pki
import-module -name netadapter
import-module -name applocker
import-module -name bitlocker
```

Enclave Consulting LLC © 2017 - All Rights Reserved

In PowerShell 3.0 and later, modules will normally be loaded as needed on-the-fly so that you do not have to import them explicitly first before using one of their cmdlets. Some modules may need to be explicitly loaded, though, if you try to access other resources related to those modules, e.g., a provider drive or global variable. It doesn't hurt to attempt to import modules which are already loaded; PowerShell won't reload the module again unless the -force switch is used.

A module can also be loaded by providing the full path to it:

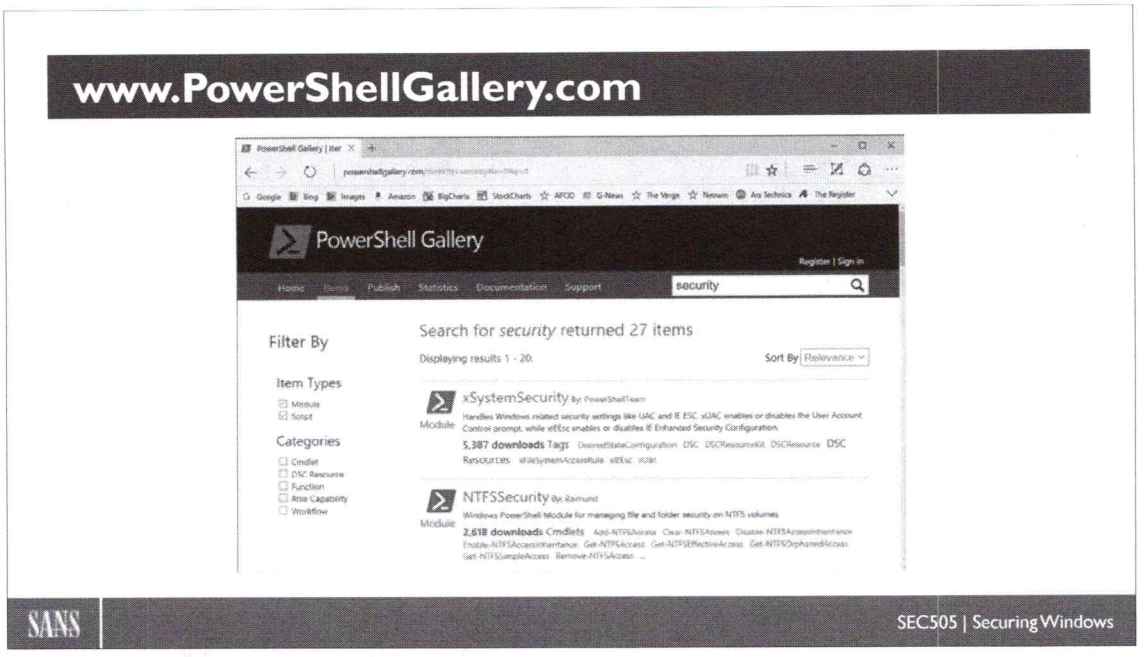
```
import-module -name .\scriptmodule.psm1
import-module -name .\binarymodule.dll
import-module -name .\manifest.psd1
```

To see the commands made available by a particular module:

```
get-command -module pki
```

How to create your own compiled or script-based modules is beyond the scope of this course, but many modules are available for free from PowerShell web sites and blogs.

Enclave Consulting LLC © 2017 - All Rights Reserved




# The PowerShell Gallery

The main public repository of PowerShell modules, scripts and DSC resources is the PowerShell Gallery (www.PowerShellGallery.com). The PowerShell Gallery includes code from authors at Microsoft, third-party companies, and community volunteers. This site is sponsored and funded by Microsoft.

Many items in the PowerShell Gallery began life on GitHub and continue as open source projects there (https://github.com/PowerShell). GitHub is where you can find some of the larger PowerShell projects. You can track these projects' version updates on GitHub and see discussions about known issues/bugs. If you want to live on the cutting edge or see what's coming over the horizon, then browse the GitHub PowerShell projects, but normally you'll just need to use the PowerShell Gallery.

The easiest way to find modules and scripts available in the PowerShell Gallery is to use the web site (www.PowerShellGallery.com) and filter using search keywords.

You can also search the PowerShell Gallery from within PowerShell itself, and this might be faster anyway once you get the hang of it (especially if you are already familiar with running commands like "apt-cache search *keyword*" on Linux).

To list the available modules and scripts from the Gallery (requires Internet access):

```
Find-Module

Find-Script
```

Enclave Consulting LLC © 2017 - All Rights Reserved

To download and save a module or script for testing (including DSC modules):

```
Save-Module -Name SomeModule -Path C:\SomeLocalFolder

Save-Script -Name SomeScript -Path C:\SomeLocalFolder
```

The difference between "saving" a script or module versus "installing" a script or module is determined by the destination folder. A script or module in one of the default paths that PowerShell uses to find scripts or modules (see next few commands) is "installed", but a script or module in any other folder is merely "saved" there.

Also, if you install a script or module for machine-wide use, not just personal use, then you must be a member of the local Administrators group or otherwise have the necessary NTFS permissions to write to those special folders. Which folders? Here are some examples.

To download and save a **module** for your own **personal** use into $env:USERPROFILE\Documents\WindowsPowerShell\Modules, which does not require Administrators membership:

```
Install-Module -Scope CurrentUser -Name SomeModule
```

To download and save a **script** for your own **personal** use into $env:USERPROFILE\Documents\WindowsPowerShell\scripts, which does not require Administrators membership:

```
Install-Script -Scope CurrentUser -Name SomeScript
```

To download and save a **module** for **machine-wide** use into $env:ProgramFiles\WindowsPowerShell\Modules, which does require Administrators membership:

```
Install-Module -Scope AllUsers -Name SomeModule

Install-Module -Name SomeModule
```

To download and save a **script** for **machine-wide** use into $env:ProgramFiles\WindowsPowerShell\Scripts, which does require Administrators membership:

```
Install-Script -Scope AllUsers -Name SomeScript

Install-Script -Name SomeScript
```

To list all modules and scripts installed from the PowerShell Gallery using Install-Module or Install-Script:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
Get-InstalledModule

Get-InstalledScript
```

To list just DSC modules and their DSC resources (as opposed to all modules):

```
Get-DscResource
```

To update all modules and scripts installed from the PowerShell Gallery using Install-Module or Install-Script in the past:

```
Update-Module

Update-Script
```

To update a specific module or script:

```
Update-Module -Name SomeModule

Update-Script -Name SomeScript
```

To uninstall a module or script that was installed from the PSGallery:

```
Uninstall-Module -Name SomeModule

Uninstall-Script -Name SomeScript
```

## Overlapping Names (Again)

What if two modules are loaded and they both include a function with the exact same name, such as "Ping"? The modules are both changing the Function:\ drive, so the last module loaded wins. It's the same Function:\ drive as before, no matter how we are updating it.

If two modules are named "Module1" and "Module2", and they both export a function named "Ping", then the desired instance of the Ping function can be executed like this:

```
Module1\Ping

Module2\Ping
```

In PowerShell, namespaces are not explicitly used and there is no *Using* keyword like in other languages. The module name is the namespace and nested modules follow scoping rules as they search up through their parent modules up to the top-level (global) namespace. For more information about implicit namespaces and how overlapping names are handled, run the following:

```
get-help about_Command_Precedence
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## What Are Packages, NuGet, and Package Providers?

To use modules from the PowerShell Gallery you do not have to know about the plumbing under the hood which makes finding, installing, updating and uninstalling modules so easy. Hence, feel free to ignore this subsection of the manual.

The module named "PowerShellGet" is what provides the above commands for using modules and scripts from the PowerShell Gallery. Here is the list of those commands:

```
Get-Command -Module PowerShellGet
```

But these commands from the PowerShellGet module are just functions which wrap lower-level commands from another module named "PackageManagement":

```
Get-Command -Module PackageManagement
```

A "package" is a piece of software along with metadata which describes that software. Usually, a package is a single compressed archive that contains both the software and the metadata. A package might contain a script, module, DLL, or an entire application composed of many binaries. The metadata inside a package will normally include information like author, version number, and redistribution license. There are different types of packages from different sources. They don't all have the exact same internal file format. On RedHat Linux, for example, you have RPM packages, and on Windows, MSI packages. RPM and MSI packages have the same purpose --wrapping up pieces of software and metadata into manageable units-- but different internal formats.

PowerShell can install and manage packages not just from the PowerShell Gallery, but potentially from any network-accessible repository using any package format type whatsoever!

Because different package file types have different internal formats, PowerShell requires a different "provider" for each type. A package provider allows PowerShell to open, parse, understand, and use a particular package type. A package provider is normally a DLL, and PowerShell will load this DLL on-the-fly as necessary behind the scenes.

To see a list of current-installed package providers:

```
Get-PackageProvider | Format-Table Name,ProviderPath
```

Package files could be downloaded manually by clicking a hyperlink in a browser, but that is not much fun. We want a nice, short, memorable name for each online package repository that we'll regularly use, and then associate that alias name with an HTTPS path and one of our installed package providers. This way, we can use a command to search a particular repository by its alias name, PowerShell will know what HTTPS web service to

Enclave Consulting LLC © 2017 - All Rights Reserved

talk to, and PowerShell will know what provider DLL to use to handle any packages downloaded from that repository.

For example, we could associate the alias name "PSGallery" with the URL of the web interface to that repository, like "https://www.powershellgallery.com/api/v2/", and also associate that alias name with the provider that can understand the packages downloaded from that repository. In fact, "PowerShellGet" is the name of the provider associated with the "PSGallery" alias and its web service Internet URL.

To see repository alias names and their associated URLs and package providers:

```
Get-PackageSource | Format-Table Name,Location,ProviderName
```

The combination of 1) the alias name for an online package repository, 2) a web service URL to that repository, and 3) a provider for the packages downloaded from that repository is called a "package source" in PowerShell. Two package sources you'll see a lot are named "PSGallery" and "nuget.org."

So, when people talk about "NuGet", that is shorthand for many things. "NuGet" is shorthand to refer to the package source named "nuget.org", the NuGet package provider (which is a DLL), the NuGet package file format, and the NuGet web site (https://www.nuget.org). NuGet is very popular with developers who use Microsoft Visual Studio. You can search and install NuGet packages from within PowerShell too.

To search the NuGet repository for all packages whose name includes "*numerics*":

```
Find-Package -Source nuget.org -Name "*Numerics*"
```

To install the MathNet.Numerics package from NuGet into C:\SomeFolder:

```
Install-Package -Source nuget.org -Name "MathNet.Numerics"
   -Destination C:\SomeFolder -Force
```

The PackageManagement module includes other commands to otherwise manage package sources and providers; you are not stuck with just the sources and providers built in by default. When you use commands like Find-Package and Install-Package, you should specify the name of the package source, like PSGallery or nuget.org.

Finally, we now also see why the following two commands produce the same output: the first command is mostly just a convenient wrapper for running the second command:

```
Find-Module

Find-Package -Source PSGallery -Type Module
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## What Are Snap-Ins? (Deprecated)

PowerShell 1.0 did not support modules, it only supported snap-ins. A "snap-in" is a compiled DLL which had to be installed like a Windows program, i.e., it required special registry modifications. A snap-in DLL implements providers and/or cmdlets. Snap-ins are still supported for backwards compatibility, but any new development should use modules instead.

To see the snap-ins loaded in the current shell:

```
# Example: SnapIns.ps1

get-pssnapin | format-list *
```

To see which snap-in makes a given cmdlet available, if the cmdlet is not from a module:

```
get-command get-process | format-list Name,DLL
```

It's possible to load more snap-ins as needed; for example, Dell has a great set of free cmdlets for Active Directory, formerly from Quest Software, that come in the form of a set of snap-in DLLs (http://www.dell.com). After running the setup program to install the snap-in DLL on your computer, its cmdlets can be made available for use.

To load the snap-in with the Dell/Quest cmdlets into your current shell:

```
add-pssnapin Quest.ActiveRoles.ADManagement
```

To see a list of all Dell/Quest cmdlets (they all have "QAD" in their names):

```
get-command Quest.ActiveRoles.ADManagement\*
```

If you want a snap-in or module loaded every time you open PowerShell, add the necessary commands to your profile script.

Enclave Consulting LLC © 2017 - All Rights Reserved

## PowerShell Remoting

**Remote command execution and interactive shell!**
**Long-term replacement for RPC, including for GUI apps.**

**Requirements:**

- PowerShell 2.0 or later.
- Remoting enabled in the WinRM service.
  - Server 2012 and later: enabled by default.
  - Client operating systems: enabled manually or by Group Policy.
- TCP ports 5985 and 5986 accessible.
- Be a member of the local Administrators group.

SANS SEC505 | Securing Windows

# PowerShell Remoting

PowerShell remoting will change your IT life. Welcome to an essential part of your future as a Windows systems administrator or SecOps engineer. PowerShell remoting is just as necessary for security and administration as RDP, SMB or RPC.

A "remote shell" looks and works just like an interactive PowerShell session on the local computer, but the commands are actually running on a remote machine over the network instead. A remote shell redirects live input/output across the network, similar in appearance to *telnet* or *ssh*, though the underlying protocols are vastly different.

And an interactive shell is not required: you can also do remote command execution, such as with a scheduled script, without a human user present. The PSEXEC tool is popular, but should never be used again.

In fact, PowerShell remoting does not require a visible command shell or script at all. A graphical tool can use PowerShell remoting under the hood, just like older management tools used RPC. Indeed, what RPC used to do for Windows in the past is being taken over by PowerShell remoting.

And PowerShell remoting does not even require PowerShell. The underlying protocols are open and vendor-neutral, so alternative clients besides PowerShell may use these remoting technologies as well.

## Requirements and Default Settings

Remoting requires at least PowerShell 2.0 on both client and target.

Enclave Consulting LLC © 2017 - All Rights Reserved

PowerShell remoting uses TCP ports 5985 and 5986 by default, so perimeter and/or host-based firewalls must allow access to these ports.

By default, only members of the local Administrators group on a target computer can connect inbound to that computer using PowerShell remoting, and, by default, only Kerberos and NTLM may be used to authenticate.

On Server 2012 and later operating systems intended for servers, remoting is enabled by default. You only have to confirm that your firewall rules allow inbound access to it.

Currently, no client operating systems turn on remoting by default.

To enable PowerShell remoting manually, run the following command:

```
enable-psremoting -force
```

This will configure the Windows Remote Management (WinRM) service and inbound firewall rules for you. If remoting is already enabled, running the above command will not hurt anything.

Remoting may also be enabled through Group Policy. In a GPO, navigate to Computer Configuration > Policies > Administrative Templates > Windows Components > Windows Remote Management (WinRM) > WinRM Service, then configure the option named "Allow Remote Server management through WinRM" to allow "*" IPv4 and IPv6 source IP addresses.

Enclave Consulting LLC © 2017 - All Rights Reserved

Allow remote server management through WinRM

Allow remote server management through WinRM

Previous Setting | Next Setting

○ Not Configured
◉ Enabled
○ Disabled

Comment:

Supported on: At least Windows Vista

Options:

IPv4 filter: *

IPv6 filter: *

Syntax:

Type "*" to allow messages from any IP address, or leave the field empty to listen on no IP address. You can specify one or more ranges of IP addresses.

Example IPv4 filters:

2.0.0.1-2.0.0.20, 24.0.0.1-24.0.0.22

*

Example IPv6 filters:

3FFE:FFFF:7654:FEDA:1245:BA98:0000:0000-3FFE:FFFF:7654:FEDA:1245:BA98:3210:4562

*

Help:

This policy setting allows you to manage whether the Windows Remote Management (WinRM) service automatically listens on the network for requests on the HTTP transport over the default HTTP port.

If you enable this policy setting, the WinRM service automatically listens on the network for requests on the HTTP transport over the default HTTP port.

To allow WinRM service to receive requests over the network, configure the Windows Firewall policy setting with exceptions for Port 5985 (default port for HTTP).

If you disable or do not configure this policy setting, the WinRM service will not respond to requests from a remote computer, regardless of whether or not any WinRM listeners are configured.

The service listens on the addresses specified by the IPv4 and IPv6 filters. The IPv4 filter specifies one or more ranges of IPv4 addresses, and the IPv6 filter specifies one or more ranges of IPv6addresses. If specified, the service enumerates the available IP addresses on the computer and uses only addresses that fall within one of the filter ranges.

You should use an asterisk (*) to indicate that the service listens on all available IP addresses on the computer. When * is used, other ranges in the filter are ignored. If the filter is left blank, the service does not listen on any addresses.

For example, if you want the service to listen only on IPv4 addresses, leave the IPv6 filter empty.

Ranges are specified using the syntax IP1-IP2. Multiple ranges are separated using "," (comma) as the delimiter.

Example IPv4 filters:\n2.0.0.1-2.0.0.20, 24.0.0.1-24.0.0.22
Example IPv6 filters:\n3FFE:FFFF:7654:FEDA:1245:BA98:0000:0000-3FFE:FFFF:7654:FEDA:1245:BA98:3210:4562

OK | Cancel | Apply

Enclave Consulting LLC © 2017 - All Rights Reserved

## Remoting Implementation Details

**WSMan Protocol = Web Services for Management.**
**WSMan is an open, DMTF protocol based on SOAP.**
**WSMan implemented by the WinRM service on Windows.**

```
#Global WinRM service settings:
dir WSMan:\localhost\Service

#Current WinRM authentication options:
dir WSMan:\localhost\Service\Auth
```

SANS SEC505 | Securing Windows

# Remoting Implementation Details

PowerShell remoting uses the Web Services for Management protocol (WS-Man or WSMan). WSMan is based on Simple Object Access Protocol (SOAP). SOAP uses HTTP/HTTPS to transmit XML-wrapped data. Hence, the WinRM service uses the WSMan protocol to implement PowerShell remoting.

WSMan is a vendor-neutral, open, ISO protocol defined by the Distributed Management Task Force (DMTF). The specifications are at http://www.dmtf.org/standards/wsman. Strictly speaking, you don't need PowerShell to remote into the WinRM service!

The Windows Remote Management service (WinRM) uses the HTTP.SYS driver to accept inbound TCP connections on ports 5985 (HTTP) and 5986 (HTTPS) by default to implement the WSMan protocol. When remoting is enabled, you'll have one or more "listeners" for WinRM in order to accept inbound WSMan connections. These listeners are actually sets of HTTP.SYS configuration settings; they are conceptually similar to protocol stack bindings.

> **Note:** It is possible to use WinRM on TCP ports 80 and 443, such as for backwards compatibility, but this is not recommended.

WinRM configuration settings can be seen and edited in the WSMAN:\ drive.

For example, to show your current WinRM listeners:

```
dir WSMan:\localhost\listener | format-list *
```

Enclave Consulting LLC © 2017 - All Rights Reserved

To list the current, global WinRM service settings:

```
dir WSMan:\localhost\Service
```

For example, to list the current WinRM authentication methods supported:

```
dir WSMan:\localhost\Service\Auth
```

By default, only Kerberos and Negotiate methods are enabled. "Negotiate" refers to an authentication package which includes only Kerberos and NTLM.

To see which group(s) currently have inbound remoting permission, we need to convert the Security Descriptor Definition Language (SDDL) string defining that permission to something more humanly readable:

```
$SddlString  = dir WSMan:\localhost\Service\RootSDDL |
   Select -ExpandProperty Value

ConvertFrom-SddlString -SDDL $SddlString
```

To see which ports the WinRM service is listening on (TCP 5985 and 5986):

```
dir WSMan:\localhost\Service\DefaultPorts
```

The WinRM service also listens on TCP port 47001, but WS-Management requests to this port will be rejected if they do not come from the local machine; nonetheless, it's best to confirm that your firewall does not allow network access to this port anyway.

The WS-Management listener settings are found in the registry under HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WSMAN\Listener\. Because the HTTP.SYS driver is used to handle the connections, it may be useful to examine its configuration too: HKLM\SYSTEM\CurrentControlSet\services\HTTP\Parameters\.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Remoting Encryption and Authentication

**By default, only Kerberos or NTLM are supported:**

- AES encryption key derived from Kerberos or NTLM handshake.
- Eliminate NTLMv1, harden Kerberos, and use a passphrase.

**Certificate authentication and SSL/TLS is optional:**

- Install certificates through PKI and Group Policy.
- Configure-RemotingForSSL.ps1

**Combine with IPSec port permissions and encryption.**

**Limit source IP with perimeter and host-based firewalls.**

SANS SEC505 | Securing Windows

# Remoting Encryption and Authentication

WSMan on Windows supports only Kerberos and NTLM authentication by default, though other authentication protocols are supported, such as SSL/TLS certificate authentication and Basic. Basic should never be enabled.

When sniffing WSMan packets, the IP, TCP and HTTP headers are all plaintext, but the SOAP payloads are encrypted using an AES key derived from the Kerberos or NTLM mutual authentication. WSMan uses Kerberos or NTLM to negotiate session keys similarly to RPC or SMB when those protocols negotiate encrypted connections on Windows. By eliminating NTLMv1 and hardening Kerberos, you improve the encryption strength of WSMan and PowerShell remoting too.

**IPSec**

For added security, use IPSec with PowerShell remoting. IPSec can restrict access to the WSMan remoting ports based on group memberships in Active Directory, and IPSec can also wrap the entire remoting session in a strong second layer of encryption.

Because there are sometimes issues getting through other peoples' firewalls when using IPSec, PowerShell remoting can optionally use SSL/TLS encryption. SSL/TLS can be used in place of IPSec or in addition to IPSec.

**SSL/TLS**

To use SSL/TLS with WSMan remoting, an SSL/TLS-compatible digital certificate must be installed on the remoting target. The subject's common name or DnsNameList name in the certificate must match the name used to connect to the target; for example, if you wish to use server47.sans.org as the fully-qualified domain name (FQDN) to resolve the

Enclave Consulting LLC © 2017 - All Rights Reserved

IP address of the remoting target, then server47.sans.org must be the subject name or one of the DnsNameList names in the certificate.

With an Enterprise Certification Authority (CA) in one's Public Key Infrastructure (PKI), Group Policy can be used to enroll computers for SSL/TLS-compatible certificates automatically in the domain.

However, configuring the WinRM service to use a certificate is a bit of a pain. You will need to script the configuration changes yourself; there is currently no automatic or Group Policy method of assigning a certificate for use with WSMan remoting.

To assist with the certificate configuration, run this script:

```
C:\SANS\Day1-PowerShell\Remoting\Configure-RemotingForSSL.ps1
```

The script will show a list of the SSL/TLS-compatible certificates on the computer, prompt you to select the desired certificate, and then configure the WinRM service to use that certificate for WSMan remoting. By modifying the source code, you can customize it for your environment to do automatic configuration.

Enclave Consulting LLC © 2017 - All Rights Reserved

**Remote Interactive Shell (Enter-PSSession)**

**Manually enable remoting on a client OS:**

```
Enable-PSRemoting -Force
```

**Connect interactively like Telnet:**

```
Enter-PSSession -ComputerName Server47

Enter-Pssession -UseSSL -Computer box.sans.org

Exit-PSSession    #When you're done.
```

SANS SEC505 | Securing Windows

# Remote Interactive Shell (Enter-PSSession)

Once the WinRM service on a local or remote system is configured to accept connections, we can establish an interactive (telnet-like) session within PowerShell. The command prompt will change, showing you the name of the machine to which you are connected.

To open an interactive remote session with a local or remote system:

```
# Example: Remoting.ps1

enter-pssession -computername Server7

enter-pssession Server7

enter-pssession localhost

enter-pssession -usessl -computername box.sans.org
```

Within the remote session, you can run commands and scripts just as though you were sitting at the machine's console, similar to a Telnet or *ssh* session; for example, run "ps" or "get-service" after connecting.

To use SSL/TLS, you must have a compatible certificate installed, the WinRM service configured to use that certificate, and you must use the -UseSSL switch when you connect.

Enclave Consulting LLC © 2017 - All Rights Reserved

To exit an interactive remote session and return to the local computer:

```
exit-pssession                # Or just "exit" by itself.
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Remote Command Execution (Invoke-Command)

**Non-interactive commands or scripts:**

```
Invoke-Command -Scriptblock {...} -Computer $Box

Invoke-Command -FilePath .\script.ps1 -Computer $Box

$Output = Invoke-Command -FilePath .\script.ps1
```

SANS SEC505 | Securing Windows

# Remote Command Execution (Invoke-Command)

Instead of an interactive session, you can also execute a single command or script on a remote system and then return to the local shell immediately. If you execute a script, the script can be on the drive of the remote computer or on the local hard drive.

To execute the command(s) in the scriptblock on a remote system:

```
invoke-command -computername Server7 -scriptblock { ps }
```

> **Note**: While a remote command is running, if you hit Ctrl-C, this keystroke sequence will be sent to the remote system to cancel the command there, the Ctrl-C is not processed locally to return control and abandon the remote process.

The -ScriptBlock parameter can contain one or many semicolon-separated commands, and these commands can include pipe symbols, flow control, aliases, etc. If the command is to run a binary or script, that file must be in the PATH of the remote system or you must specify the file system path to the file on the remote computer.

## Remote Execution Of A Local Script (Invoke-Command -FilePath)

You can also run a local script on the remote machine with the -FilePath parameter; you don't have to copy the script to the remote machine first, and the output of the script will be returned and displayed on the local system. In this case, the -FilePath parameter takes the file system path to a local script, not a script on the drive of the remote machine.

To execute a local script on a remote system, but see the output in the local shell:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
invoke-command -computername Server7 -filepath .\somescript.ps1
```

If you need to pass in arguments to the -FilePath script, this is done with the -ArgumentList parameter, but be aware of how the arguments are being passed in: you pass in an array of objects which go into the $Args array at the target computer. The objects must be passed in the correct order so that the parameter binding mechanism at the target will pass them into the correct named parameters required by the script.

Enclave Consulting LLC © 2017 - All Rights Reserved

**Fan-Out Remoting As Background Job**

```
$TargetsArray = @("Srvr47","Srvr48","Srvr49")

Invoke-Command -FilePath .\script.ps1 -AsJob
   -Computer $TargetsArray

Get-Job  #Query job status and get job ID number.

#The output will be collated by computer name:
$Output = Receive-Job -Keep -ID 6
```

SANS SEC505 | Securing Windows

# Fan-Out Remoting As Background Job

If you need to execute commands on a large number of remote computers, you don't have to write a loop in a script, the invoke-command cmdlet can take an array of computer names as an argument. This works with both the -ScriptBlock and -FilePath parameters.

To run a command or script on multiple remote systems:

```
$servers = @("Server7", "Server8", "Server9")

invoke-command -computername $servers -scriptblock { ps }

invoke-command -computername $servers -filepath .\somescript.ps1
```

As you might imagine, PowerShell remoting may have some security implications, but hardening of the feature cannot be discussed here due to time constraints. The WS-MAN session used for remoting is authenticated and encrypted, but if you're worried, you can augment this with IPSec port permissions and encryption.

**Remote Execution As A Background Job**

Long-running scripts or scriptblocks executed across hundreds of machines can tie up the local shell for a long time. You can always just open another shell, but an alternative is to invoke the command as a background job, which immediately returns the local user to their interactive shell, but continues execution and output capturing in the background. Periodically, the status of the job can be queried with Get-Job until the State property shows that it has "Completed." After all the commands on all the remote machines have been executed, the output of these commands can be captured to a variable. Very

Enclave Consulting LLC © 2017 - All Rights Reserved

importantly, every object returned will have a PSComputerName property which indicates the name of the remote computer on which the object was originally created. This is important for the sake of sorting and organizing the output, especially when handling errors.

To execute a set of commands on multiple remote machines as a background job, query the status of the job, then capture the output of the job (the commands on the machines) to a variable which includes the name of the target computer as a property on each object:

```
$Servers = @("Server7", "Server8", "Server9")

Invoke-Command -ComputerName $Servers -ScriptBlock { ps } -AsJob

Get-Job    # Query status and get the ID number.

$Output = Receive-Job -Keep -ID 6

$Output | Format-Table PSComputerName,ProcessName -AutoSize
```

Just imagine querying all the computer accounts in an OU in Active Directory and piping these computer objects into Invoke-Command for mass remote execution...very handy.

## Remoting Cmdlets and Other Tools

There is a distressingly large number of commands related to the configuration of WinRM and PowerShell remoting. Don't worry, you'll rarely need to touch them.

### CmdLets

To get a listing of all the cmdlets and help files related to WinRM and remoting:

```
get-help *wsman*
get-help *remot*
get-help *session*
get-help *cim*        # *cim* requires PowerShell 3.0 or later.
```

The main cmdlet for configuring WinRM is Enable-PSRemoting, which will run other cmdlets for you to get things quickly up and running.

### WSMan Drive

There is also a provider and drive for the WS-Management protocol and WinRM service named "WSMan:\". Many of the remoting-related cmdlets modify objects in this drive, but you can examine and modify these objects yourself directly too.

To enter the WSMan:\ drive and, for example, list the WinRM authentication settings:

```
cd wsman:\

dir .\localhost\service\auth
```

Enclave Consulting LLC © 2017 - All Rights Reserved

### WINRM.VBS

There is also the legacy winrm.vbs script, which still works and can be useful for troubleshooting. Hackers and malware also like to use it. The script is located in the %SystemRoot%\System32 folder. (There you'll find a winrm.cmd batch script too, but that's just a wrapper to run the winrm.vbs script using cscript.exe.)

To see the help for the legacy winrm.vbs script and show the current WinRM version:

```
winrm /?

winrm identify
```

### Group Policy Management

Of course, all of this is intended for enterprise deployment, so the WinRM options can be managed through Group Policy too. To find the WinRM settings in a GPO, navigate to Computer Configuration > Policies > Administrative Templates > Windows Components > Windows Remote Management (WinRM).

### WINRS.EXE

In a GPO you'll also find a container named "Windows Remote Shell". This is for remote command execution with the winrs.exe tool, which is not the same thing as PowerShell remoting despite the confusing name of the GPO container. The GPO path to this container is Computer Configuration > Policies > Administrative Templates > Windows Components > Windows Remote Shell.

The winrs.exe tool predates PowerShell 2.0 and, once the WinRM service has been configured, this tool can also be used for remote command execution. Its usage is fairly straightforward, just run "winrs.exe /?" for help and examples.

PowerShell is preferred, so winrs.exe is not discussed any further here.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Remoting File Copy and Hyper-V Direct

```
# Create a WSMan session to a remote host with PowerShell 5.0+:
$Session = New-PSSession -ComputerName dc.testing.local

# Upload a file to the remote host (-ToSession):
copy -Path C:\LocalFolder\file.txt -Destination C:\RemoteFolder\file.txt
   -ToSession $Session

# Download a file from the remote host (-FromSession):
copy -Path C:\RemoteFolder\file.txt -Destination C:\LocalFolder\file.txt
   -FromSession $Session
```

SANS SEC505 | Securing Windows

# Remoting File Copy and Hyper-V Direct

PowerShell 5.0 and later supports the copying of files through the remoting channel itself, which means no additional ports must be opened between source and destination machines other than the already-open remoting port(s).

```
# Create a session to a remote host with PoSh 5.0+:

$session = New-PSSession -ComputerName dc.testing.local


# Upload a file to the remote host (-ToSession):

copy -Path C:\LocalFolder\file.txt -Destination
   C:\RemoteFolder\file.txt -ToSession $session


# Download a file from the remote host (-FromSession):

copy -Path C:\RemoteFolder\file.txt -Destination
   C:\LocalFolder\file.txt -FromSession $session
```

The performance is far slower than SMB, though, so the main advantage of this technique is penetrating firewalls that would otherwise block the file copy traffic.

Enclave Consulting LLC © 2017 - All Rights Reserved

Note that you cannot remote into two hosts simultaneously (A and B) and copy files directly between them (A <--> B) using your two sessions.

## Hyper-V PowerShell Direct

If you have Windows 10, Server 2016 or later, with PowerShell 5.0 or later, on 1) a system running Hyper-V as a host and also 2) in a VM guest running on that host, then all of the above remoting commands can be used from the host Hyper-V server to the guest VM using the Hyper-V VMBus instead of a real network connection.

This means that the VM guest does not have to be accessible over the network and does not require any TCP/UDP ports to be opened. The VMBus is a communications shared memory buffer implemented by the Hyper-V hypervisor, it does not use the protocol stack on either the host server or the guest VM.

This also means that you can remote into a VM on the Hyper-V server with Enter-PSSession and Invoke-Command. PowerShell Direct is not just for file copying. The guest VM does not have to be accessible over the network or even have a virtual network interface as a VM.

Again, both host and guest must be Windows 10, Server 2016 or later, both must have PowerShell 5.0 or later, PowerShell must be running elevated on the Hyper-V server itself, and you must be a member of the Administrators group in the guest VM.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Today's Agenda

1. Setting Up Our Training VM
2. PowerShell Overview & Tips
3. Modules, Functions & PowerShell Remoting
4. Essential Commands & Scripting Examples
5. Writing Your Own Functions & Scripts

SANS SEC505 | Securing Windows

# Today's Agenda

In the next section we will get more experience running essential commands. These are cmdlets and functions you'll use in almost every script you write. We will also get more practice running scripts.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Parameters, Arguments, Switches and Piping

**Parameters, Arguments, Switches**

```
get-member -inputobject $x -membertype property -static
get-member -input $x -m property -static
gm -i $x -m property -s
```

**A piped object becomes a parameter argument:**

```
get-date "11/29/17" | gm -m property -static
```

**The `-WhatIf` switch for job security...**

SANS SEC505 | Securing Windows

# Parameters, Arguments, Switches and Piping

Cmdlets are named with verb-noun pairs, like "get-process" and "stop-process", to make the cmdlets easier to identify and use. Most cmdlets take one or more parameters as input when the cmdlet is run.

To see the parameters a cmdlet takes, such as with the new-item cmdlet:

```
get-help new-item -detailed
```

The overall syntax for a command might look like the following:

*command -parameter1 arg1 -parameter2 argument2 -switch*

Parameters begin with a single dash ("-") and can take zero or more arguments. A parameter that doesn't take any arguments is called a "switch" because it's either there or it's not when the command is run (on/off, true/false). Multiple arguments to a parameter are often separated by a comma. Loosely speaking, an "argument" is *any* input value, but, strictly speaking, an argument is an input to a parameter only; that is to say, arguments are what follow parameters, arguments are never direct inputs to cmdlets, functions or scripts (that's what parameters are).

Ideally, every parameter used should be fully spelled out when used, but a PowerShell component called the "parameter binder" will try its best to figure out what you're trying to do when passing in parameters and arguments, hence, you can often pass in abbreviated names for parameters and the command will still work as expected. For example, in the following statements "$x" is an argument to the "-inputobject" parameter,

Enclave Consulting LLC © 2017 - All Rights Reserved

"property" is an argument to the "-membertype" parameter, "-static" is a switch without arguments, and "gm" is an alias for "get-member", hence, these three commands all do exactly the same thing:

```
# Example: Parameters.ps1

get-member -inputobject $x -membertype property -static

get-member -input $x -m property -static

gm -i $x -m property -s
```

And you can often pass in parameters and arguments without naming them at all, as long as it is clear enough to PowerShell's binder what you are intending. The following statements all do the same thing because the PowerShell binder figures out what you are intending:

```
get-process -name "powershell"

get-process -n "powershell"

get-process "powershell"
```

PowerShell's parameter binder will go to great lengths to try to match piped objects to the correct parameter(s) in the next cmdlet of the pipeline so that these objects can be taken as arguments to these parameters without error. If a piped object's .NET class type does not exactly match the class type required by the next cmdlet's available parameters, then the binder will try to *convert* the object to the correct expected type; if that doesn't work, the binder will try to match the name of the receiving parameter against the name of at least one property of the piped object and use that for the mapping; and if that doesn't work because of object type mismatch, then the binder will again try to convert the piped object to the correct type by coercion; and if *that* doesn't work, well, then the binder finally gives up and an error is produced.

## Piping and Redirection

Cmdlets and functions can take parameters as input. Each parameter has a name and any arguments to the parameter follow the parameter name. But PowerShell allows you to use a trick when passing arguments to parameters: you can "pipe" an argument into a cmdlet or function and PowerShell will try to bind that argument to the correct parameter automatically. Usually, the data being piped is the output of a prior command, and a chain of two or more commands that are linked such that the output of the prior command is taken as the argument to the parameter of a subsequent command is called a "pipeline". How do you link commands together like this? With the pipe (" | ") symbol of course!

```
get-childitem c:\ | format-table

get-process "powershell" | get-member
```

Enclave Consulting LLC © 2017 - All Rights Reserved

But which parameter in the receiving cmdlet gets the piped object as its argument? If you look at the help for a cmdlet you'll often find a parameter named "-InputObject", and that parameter has a note saying "Accept pipeline input?" which will be True. Only one parameter to a cmdlet can be marked as accepting piped input and that's the parameter which gets the piped data.

Redirection is not the same as piping. The redirection symbols, ">" and ">>", also pump data somewhere else, but they typically only pump data into files on the hard drive. ">" will create or overwrite the data in the target file, while ">>" will create or append to the data already in the target file. A special case of redirection is when the output of a command is silently destroyed by redirecting it to $null.

## What If More Than One Object Gets Piped?

A command will often emit more than one object as its output. If the command's output is being piped into another cmdlet, what happens to all the objects piped? A receiving cmdlet processes input objects as soon as those objects are piped, one at a time, instead of waiting for all of the input objects to arrive. That is to say, the second and subsequent cmdlets in a pipeline don't wait for the first cmdlet at the front of the pipeline to emit its last object before they begin processing the first object received. That is to say, the first cmdlet in a pipeline emits its output objects as soon as possible, one at a time, instead of "batching up" all of its output objects and then disgorging them all at once on the head of the next cmdlet in the pipeline. If each cmdlet in a pipeline had to process 100 million objects, it's far better to work as a continuously streaming *assembly line* instead of each cmdlet standing idle until all 100 million objects are piled on it when it's that cmdlet's "turn" -- and *that's* how PowerShell pipelines indeed work, as continuous assembly lines, with the parameter binder working overtime to keep everything straight. (Will 100 million processes be launched, one after another, to do all the work? No, there will be just one `powershell.exe` process utilizing just one of its threads to do the work.)

## What If I Want "Sniff" The Pipeline? (Tee-Object)

When you are piping objects from one command to another, it would be nice sometimes to capture copies of the objects at some location in the pipeline without interrupting the flow of objects through the pipeline, i.e., it would be nice to "sniff" the pipeline just as you might use a protocol analyzer to sniff the network. This is what the tee-object cmdlet is for.

The tee-object cmdlet accepts objects coming through the pipeline, copies these objects into a file or variable, then passes the unchanged objects down the pipeline to the next command. This is a useful for debugging.

```
dir | tee-object -variable sniff | where {$_.length -gt 20}
```

Importantly, note that "sniff" is the variable to be filled with objects coming down the pipeline, but it is not preceded with a "$" as normal; just specify the name of the variable

Enclave Consulting LLC © 2017 - All Rights Reserved

only. You also don't have to pre-create the variable, it'll be created for you. If the variable already exists, its contents will be overwritten.

## Measure Twice, Cut Once (The "-WhatIf" Switch)

Notice in the help for many cmdlets the existence of a "-WhatIf" switch. If this is used, the cmdlet will not make any changes to the system; instead, the cmdlet will show a list of the changes it *would* make *if* the -WhatIf switch were not used.

```
remove-item * -recurse -WhatIf
```

This is very handy when you're about to delete hundreds of files or registry values and you want to make sure your code will do what you expect!

## Grouping

A "grouping" is a pair of plain parentheses (like these) which contain either a mathematical calculation, a single command, or a pipeline of commands.

When grouping mathematical calculations in an expression, it's for explicitly controlling the order in which the mathematical operators in the expression are executed:

```
4 - 9 * 2              # Yields -14, * has a higher precedence.

(4 - 9) * 2            # Yields -10

4 - (9 * 2)            # Yields -14, you know what to expect now.
```

A pair of parentheses can also contain a command so that the properties and methods of the object(s) outputted by the command can be directly accessed without assigning that output to another variable first. For example, the following emits the string "Running" as the output:

```
$x = get-service spooler
$x.status
```

But the following statement prints the same thing without using a temporary variable:

```
(get-service spooler).status
```

This works because the parentheses act to capture the output of the command in a temporary unnamed variable represented by the contents of the parentheses themselves, hence, the properties and methods of the object(s) in the variable can be directly invoked.

To show the status of the spooler service, then stop and restart it:

```
# Example: Grouping.ps1
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
(get-service spooler).status
(get-service spooler).stop()
(get-service spooler).start()
```

To get today's date and time in a *System.DateTime* object, add three days to the current date, then show the result:

```
$ThreeDaysFromNow = (get-date).AddDays(3)
```

But a parenthetical grouping can also contain an entire pipeline instead of a single command. This is really no different than executing a single command, the output is all the same as far as PowerShell is concerned, i.e., one or more objects.

To get the count of processes using over 50MB of workingset memory:

```
(get-process | where {$_.workingset -gt 50MB}).count
```

> **Note:** "50MB" is actually a PowerShell shorthand trick to represent "50 * 1024 * 1024", hence, we're still doing the comparison against the number of bytes in the workingset, not the number of megabytes (try executing "50MB" in the shell).

Grouping can also be used to ensure that one part of a command completes before the rest of the command begins execution. For example, in the following command, the contents of a file are read, those contents modified in RAM, then the changes written back to the file. To ensure that the file is read in its entirety and the handle to the file released before proceeding through the pipeline, the get-content command is placed in parentheses.

```
(get-content file.txt) -replace "out","in" | set-content file.txt
```

## Subexpressions

A subexpression is similar to a grouping, but a subexpression can contain multiple commands or pipelines, separated by semicolons, as well as flow-control statements like *if-elseif-else* or *foreach*. Another difference is that the parentheses of a subexpression are preceded by a dollar sign ($) or the at-symbol (@) to demarcate it as a subexpression instead of a simple grouping.

To see how many megabytes total are in the workingsets of all running processes:

```
# Example: Subexpressions.ps1

"The sum of all workingset sizes is $($x = 0 ; get-process |
foreach {$x += $_.workingset} ; $x / 1024 / 1024) MB."
```

If a subexpression is preceded by "$" and the statements in the parentheses return a single object, then the subexpression becomes a scalar (singular) value, but if multiple objects are returned, then the subexpression becomes an array. On the other hand, if the

Enclave Consulting LLC © 2017 - All Rights Reserved

subexpression is preceded by "@", then the subexpression will always become an array, even if only a single object is returned by the statements in the parentheses. If you know for sure that the subexpression will *always* return a single object, then use "$", but if you don't know if one or multiple objects will be returned, then use "@" to force the subexpression to become an array so you can deal with it consistently.

To get the count of PowerShell processes running by getting the count property of the resulting array, even if the array only has a single item:

```
@(foreach ($x in get-process) {if ($x.name -like "powershell")
{$x}}).count
```

Behind the scenes, the "@" syntax is casting the result of the statement(s) inside the parentheses to a *System.Object* array, hence, if the result is already an array, casting it as an array again does nothing; but if the result is a single value, it is cast as an array whose only item is the original single item. (Casting an array as an array does *not* create a new array whose first and only item is the original array, that is something different.)

Enclave Consulting LLC © 2017 - All Rights Reserved

## "Hello World!"

**These commands all do the same thing:**

```
"Hello World!"
"Hello World!" | out-default
"Hello World!" | out-default | out-host
```

**If you don't choose where to pipe your output, your objects will secretly get piped into `Out-Default` and it will decide what to do with them!**

SANS SEC505 | Securing Windows

# "Hello World!"

Here's the traditional "Hello World!" script written in the PowerShell language:

```
"Hello World!"
```

What, no "echo"? No "print"? No "write"? How or *why* does this result in the text being displayed on-screen?

A string like "Hello World!" is really an object in the .NET class library of type *System.String*. You can cast that string as a *System.String* object with no change in output because you didn't change its type, it was already that type of object:

```
[System.String] "Hello World!"
```

But why is it printed on-screen at all when there's no command to do so?

When an object is not piped anywhere else, PowerShell pipes it into the out-default cmdlet automatically, hence, the following two commands do the same thing:

```
"Hello World!"
"Hello World!" | out-default
```

Out-default examines the objects fed into it to determine what to do with them. If the objects are just simple strings, the strings go straight to out-host. If the objects have a registered "view" in one of PowerShell's *.format.ps1xml files, this information determines which formatter cmdlet to use and PowerShell pipes to that formatter (these

Enclave Consulting LLC © 2017 - All Rights Reserved

XML files are in the $pshome folder). If no view is specified and the first object in the pipeline has five or more properties, then format-list is used as the formatter, with four or less properties, then format-table is used. The output of a formatter cmdlet is just string data, which goes straight to out-host. Because our "Hello World!" is a single string, the formatter is skipped and the output goes straight to out-host.

Formatters are the format-* cmdlets that format object data as strings, while outputters are the out-* cmdlets that actually display/print/redirect that data somewhere. When using PowerShell as a shell, the default outputter is out-host, hence, the following three lines do the exact same thing:

```
"Hello World!"
"Hello World!" | out-default
"Hello World!" | out-default | out-host
```

## Alternative Formatters and Outputters

There are other formatters available which can be used:

```
"Hello World!" | format-list
"Hello World!" | format-table
"Hello World!" | format-wide
"Hello World!" | format-custom
```

And there are other outputters too:

```
"Hello World!" | format-list | out-null
"Hello World!" | format-list | out-string
"Hello World!" | format-list | out-file c:\temp\file.txt
"Hello World!" | format-list | out-printer
```

The outputters will be discussed shortly, but their names are fairly self-explanatory.

## Why Have Out-Host? What's The Point?

But what is the point of having an out-host cmdlet? PowerShell is really a programmable engine that can be utilized by different "hosts". The command shell we're working in is just one type of host that can use PowerShell. Other hosts might be DLL snap-ins for the graphical MMC console or touch-oriented "Metro" apps on Windows 8 and later.

Enclave Consulting LLC © 2017 - All Rights Reserved

## The Formatters (Aliases: fl, ft)

**Format-List**

- Shows each property on a separate line.

```
get-process | format-list *
```

**Format-Table**

- Shows properties in columns, like a spreadsheet.

```
ps | format-table id,name,path -autosize
```

SANS SEC505 | Securing Windows

# The Formatters (Aliases: fl, ft, fw, fc)

The format-* cmdlets do not display information, they merely format it. The out-* cmdlets are what display or otherwise output information in some way. PowerShell pipes to out-default by default, which then decides which formatter and outputter to use.

Objects generally don't know how to format themselves, so PowerShell includes XML files in the $pshome directory with the ".ps1xml" extension that tell PowerShell how to format certain types of objects. If you don't wish to use these defaults, the format-* cmdlets support a -view parameter to use a different or custom XML file instead.

## Format-List (Alias: fl)

When formatting data, format-list is often the best formatter to use because it shows all items in a vertical list that usually isn't affected by the width of the command shell window. Other formatters might wrap or truncate data.

```
# Formatters.ps1

get-item hklm:\software | format-list *
get-item hklm:\software | fl *

get-service | format-table Name,DisplayName,Status -autosize
get-service | format-list *
```

Often you'll want to capture just a single property or a few properties of an object or collection of objects. In this case, you can tell all the format-* cmdlets which properties you're interested in and the cmdlets will ignore the rest.

Enclave Consulting LLC © 2017 - All Rights Reserved

To format just the name of the registry key and its count of subkeys:

```
get-item hklm:\software\microsoft | fl name,subkeycount
```

## Format-Table (Alias: ft)

To format information into a table and have its column widths automatically resized to fill the entire command window's width, use the format-table cmdlet (alias: ft).

To try to show the *.exe files in %WinDir% using the format-table cmdlet:

```
dir $env:windir\*.exe | format-table *
dir $env:windir\*.exe | ft *
```

As you can see from the output of the above commands, the data in each column is truncated so severely that the display is almost useless. Either use the format-list cmdlet instead or only show a small subset of the available properties.

When the resizing doesn't work correctly, another option is to use the -autosize parameter to make the cmdlet wait for all of the information to be completely piped into the cmdlet before it makes its resizing decisions.

To show only the names of the *.exe files in the %WinDir% folder, along with their last-accessed date and time, using the -autosize switch:

```
dir $env:windir\*.exe| ft name,lastaccesstime -autosize
```

## Format-Wide (Alias: fw)

The format-wide cmdlet is similar to format-table, but it only shows one property, either the one you specify or just the first in the object. Using the -column parameter, you can choose the number of columns into which the information will be formatted.

To show just the names of the files and folders in %WinDir%, but format that data into three columns to show more data per page:

```
dir $env:windir | format-wide name -column 3
dir $env:windir | fw name -col 3
```

## Format-Custom (Alias: fc)

The format-custom cmdlet is mainly for use with custom XML format files (with ".ps1xml" extensions) in the $pshome folder. You can add your own format files with the update-formatdata cmdlet.

It will be rare that you'll need it, but you can also format information about objects and properties in a verbose way as your recurse through the various objects-as-properties. To get a taste, compare the outputs of the following two commands, but be careful of setting

Enclave Consulting LLC © 2017 - All Rights Reserved

the -depth parameter higher than 4 (and don't forget about Ctrl-C when it runs on too long):

```
dir $env:windir | format-custom * -depth 1
dir $env:windir | format-custom * -depth 2
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## The Outputters

**Out-Host**
- Displays text in the shell.

**Out-File**
- -Append
- -Encoding

**Out-Gridview**
- Pops up a GUI app.
- -PassThru

**Out-Printer**
- Local or network-attached printer.

**Out-Null**
- Silently destroys data (and laughs).

**Out-String**
- Converts objects to "real" strings.
- The only outputter with pipeable output, such as to `grep.exe`.

SANS SEC505 | Securing Windows

# The Outputters (Aliases: oh, lp, ogv)

The format-* cmdlets do not display information, they merely format it. The out-* cmdlets are what display or otherwise output information in some way. PowerShell pipes to out-default and then to out-host if no other outputter is specified. If you wish to output data directly into a comma-separated values (CSV) file, extensible markup language (XML) file, JavaScript Object Notation (JSON) file, or HTML file, then use export-csv, export-clixml, convertto-json, or convertto-html cmdlets instead.

## Out-Default

All output that is not explicitly piped into another outputter is piped into out-default. This cmdlet can also be replaced in custom applications. Out-default examines the objects piped into it and decides which format-* cmdlet, if any, and which out-* cmdlet should be used to format and output the data. If the objects are just strings, the strings go straight to out-host. If the objects have been tagged with a preferred "view", this information determines which formatter to use and PowerShell pipes to that formatter. If no view is specified and the first object has five or more properties, then format-list is used; four or less, then format-table. Since the output of a format-* cmdlet is one or more strings, this data goes straight to out-host by default.

## Out-Host (Alias: oh)

The interactive command interpreter window we call "PowerShell" is not the only application that can use the underlying PowerShell engine. A graphical program, such as the Exchange Server 2007 MMC console snap-in, can also use PowerShell directly, but not by opening up a command prompt window. The "host" of the PowerShell engine is what sends it commands and which receives PowerShell's output. *Whatever* the current

Enclave Consulting LLC © 2017 - All Rights Reserved

host process is for PowerShell, the out-host cmdlet sends output to it. For this course, the only host we care about is the DOS-prompt-looking window with the blinking cursor in front of you now, i.e., the interpretive shell.

Unless some other outputter is chosen, all output eventually is passed to out-host. The out-default cmdlet, for example, will always pipe to out-host by default. You don't need to specify "|out-host" as the last part of every command, it's taken care of for you. But if you wish, you can explicitly pipe to out-host so that you can invoke its -paging parameter to only show one page of output at a time in the shell, just like with the "more" command. In fact, "more" is the name of a built-in PowerShell function that simply calls out-host with the -paging parameter. However, the "-paging" switch only works in the console PowerShell.exe host, not the graphical PowerShell_ISE.exe host.

To show lengthy output one page at a time in PowerShell.exe (not ISE):

```
get-childitem $env:windir | out-host -paging

get-childitem $env:windir | more
```

If an unformatted object is passed into out-host (or any of the out-* cmdlets for that matter), out-host will first run it back through out-default to let it decide which formatter to use, and then out-default will pipe back out through out-host again. Hence, even if you don't specify it, one way or another a formatter is invoked before any non-string data is displayed in the shell (simple strings are just sent to out-host).

If you try to pipe or redirect the output of out-host to something else, nothing goes through, it doesn't work. Strictly speaking, out-host produces no output in the PowerShell sense (it emits no objects or raw text), out-host only manipulates the shell window you see before you. This isn't just a hair-splitting distinction, try running the following two commands (the first doesn't work as expected, but the second one does):

```
get-process | out-host | findstr.exe "pow"

get-process | out-string | findstr.exe "pow"
```

## Out-String

In traditional shells, like bash and CMD, you only pipe text. In PowerShell, you can pipe text or objects. While object-orientation is generally more powerful, sometimes text manipulation is easier or the only available option, especially when piping to older binaries like FINDSTR.EXE.

The out-string cmdlet converts objects into strings. By default, out-string emits one long string containing all the output data, but with the -stream parameter it can output one line at a time instead. And this output is pipeable! Out-string is the only out-* cmdlet which produces pipeable output. However, the output is merely text, not objects, hence, if the next command in the pipeline is expecting objects, the command will fail.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Out-File

Where else can output be sent? To a file! Out-file takes a path to the output file as a mandatory parameter, but there are other parameters to control other aspects of the operation, such as character encoding (ASCII, Unicode, etc.), to overwrite or append, to force the creation of folders as necessary, to ignore the read-only bit, to confirm actions, and so on.

To save your list of aliases to a file (c:\aliases.txt) and overwrite that file if it exists, even if the file has the read-only bit set (-force):

```
get-alias | out-file c:\aliases.txt -force
```

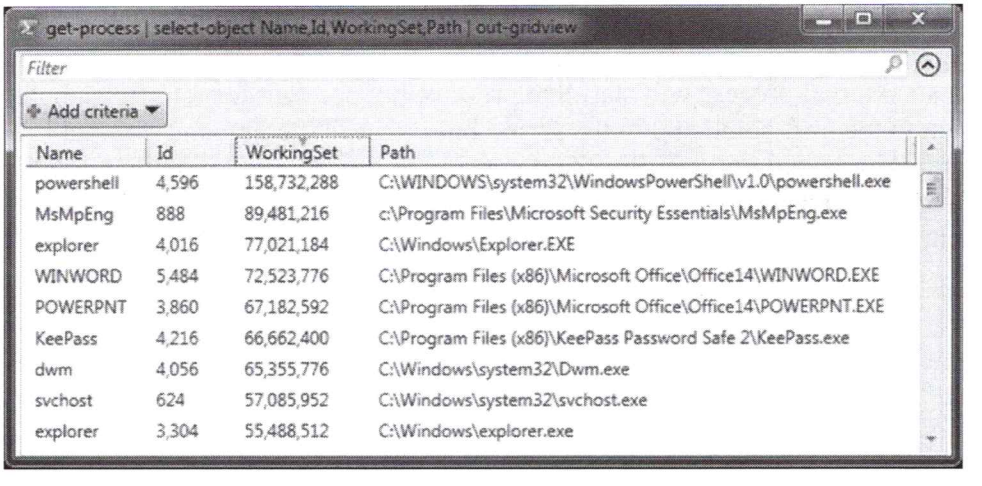
You could also use the redirection symbol (">") instead of piping through Out-File, but this wouldn't work if the file had the read-only bit set since you can't use the -force parameter with that redirection operator.

## Out-GridView (Alias: ogv)

The out-gridview cmdlet requires PowerShell 2.0 or later. This cmdlet will pop up a graphical table containing the data piped into the cmdlet. You can sort on the columns in the table and filter rows with custom criteria. With the -passthru switch, objects piped into out-gridview can be filtered and highlighted, then only the highlighted objects will be passed through the pipeline to the next command when the OK button is clicked.

To see your running processes in a graphical table:

```
ps | select-object Name,Id,WorkingSet,Path | out-gridview
```




| Name | Id | WorkingSet | Path |
|---|---|---|---|
| powershell | 4,596 | 158,732,288 | C:\WINDOWS\system32\WindowsPowerShell\v1.0\powershell.exe |
| MsMpEng | 888 | 89,481,216 | c:\Program Files\Microsoft Security Essentials\MsMpEng.exe |
| explorer | 4,016 | 77,021,184 | C:\Windows\Explorer.EXE |
| WINWORD | 5,484 | 72,523,776 | C:\Program Files (x86)\Microsoft Office\Office14\WINWORD.EXE |
| POWERPNT | 3,860 | 67,182,592 | C:\Program Files (x86)\Microsoft Office\Office14\POWERPNT.EXE |
| KeePass | 4,216 | 66,662,400 | C:\Program Files (x86)\KeePass Password Safe 2\KeePass.exe |
| dwm | 4,056 | 65,355,776 | C:\Windows\system32\Dwm.exe |
| svchost | 624 | 57,085,952 | C:\Windows\system32\svchost.exe |
| explorer | 3,304 | 55,488,512 | C:\Windows\explorer.exe |

To choose which highlighted objects continue through the pipeline (click the OK button):

```
get-service | out-gridview -passthru | stop-service
```

Enclave Consulting LLC © 2017 - All Rights Reserved

**Tip:** After highlighting one or more objects shown in out-gridview, press Ctrl-C to copy the data to the clipboard. You can then use Ctrl-V to paste that text into an Excel spreadsheet, OneNote page, or other document.

## Out-Printer (Alias: lp)

The out-printer cmdlet is self-explanatory. Without any parameters, the output is sent to the default printer on the computer where the command is run. The optional -name parameter allows you to specify the name of a local or remote shared printer instead.

If you have Adobe Acrobat or Microsoft OneNote installed, you can also "print" to a PDF file or to a OneNote page.

To print a hardcopy of all your aliases:

```
get-alias | out-printer
get-alias | out-printer \\server\sharedprinter
get-alias | out-printer "Local Printer Name"
```

## Out-Null

Anything piped to out-null is silently destroyed. It outputs nothing. You can also redirect unwanted output to a built-in variable named "$null" to accomplish the same thing. $null always contains nothing.

To discard unwanted output:

```
get-alias | out-null
get-alias > $null
```

Another use for out-null is to make PowerShell wait for the current line to finish executing before moving on to the next; for example, a command in a script which launches an external process will start that process, but not wait for it to finish before moving on to the next command in the script, but if that first command's output is piped to out-null, PowerShell will wait for the command to finish before moving on to the next line. If you want PowerShell to wait for a command to finish but you don't want to lose the output, capture the output to a variable.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Exporting, Importing and Converting Object Data

**Export-CSV and Import-CSV**

- Properties become column headers, objects are rows:

```
get-service | export-csv -path services.csv
$data = import-csv -path services.csv
# What is inside the $data array now? Text?
```

**Export-CLIXML and Import-CLIXML**

**ConvertTo-JSON and ConvertFrom-JSON**

SANS SEC505 | Securing Windows

# Exporting, Importing And Converting Object Data

There are several cmdlets to export, import and convert data to/from a variety of different data formats, such as XML, JSON and HTML.

```
get-command -verb import

get-command -verb export

get-command -verb convert*
```

## Export-CLIXML And Import-CLIXML

The export-clixml cmdlet will create an XML file containing the type name and all the properties of the objects piped into the cmdlet. The objects can later be recreated using import-clixml.

```
get-service | export-clixml -path services.xml

$objects = import-clixml -path services.xml
```

The export-clixml cmdlet supports an optional parameter named "-depth" to control how many object layers deep you wish to export, since a property of one object might itself be an entire object with its own properties, including another object, and so on. The -depth parameter can give us confidence that we are saving all the object data desired, even when there are complex nested objects. And when you have large amounts of XML data, such as a 100GB XML log file, there is a special query language named "XPath" to efficiently search that data and extract only what you need. You can also import data

Enclave Consulting LLC © 2017 - All Rights Reserved

from an XML file and pipe those objects in where-object to perform searches. When in doubt, export data to XML.

## Export-CSV And Import-CSV

Another cmdlet for exporting objects and their properties to a file is export-csv (alias: epcsv), which will create a comma-separated values (CSV) file. The objects can later be recreated using import-csv, but there is no -depth parameter (as with export-clixml), hence, objects might not be exported with fidelity if they are too complex. On the other hand, CSV files can be directly opened in Excel without modification, are easy to import into databases, easy to edit by hand, and easy to modify by script or external tools.

Another advantage of CSV files is that when you import a CSV file, each row from the CSV file is imported as an object and the column headers from the file are used by PowerShell as property names. This makes using CSV files very convenient since you don't have to expend energy parsing the file, extracting fields, creating objects, assigning fields to variables, etc.

```
# Example: CSV.ps1

get-service | export-csv -path services.csv

$objects = import-csv -path services.csv

$objects | where-object {$_.status -eq 'Running'} |
   ft DisplayName
```

So, while XML has better object fidelity, CSV files are easier to work with and smaller; often, the properties you need to view and manipulate are simple strings and numbers, hence, you don't need the capabilities (or complexities) of XML.

When you have simple (non-nested) objects, when you need to keep the export file as small as possible, when you want to manipulate the data in a spreadsheet or using tools like sed/grep/awk, then export your data to CSV.

## ConvertTo-Json And ConvertFrom-Json

A popular data format used with web applications is JavaScript Object Notation (JSON). JSON is half-way between XML and CSV in terms of complexity and features, but closer to XML than to CSV.

Similar to the import/export cmdlets just discussed, there are two cmdlets related to JSON: ConvertTo-Json and ConvertFrom-Json. Since these are "converter" cmdlets, you must explicitly save to and read from JSON files using other cmdlets, such as Out-File and Get-Content.

JSON files are just text files, so they might be saved with either the ".txt" or ".json" file name extensions, or both (like "*filename*.json.txt") to give your users a hint.

Enclave Consulting LLC © 2017 - All Rights Reserved

```
Get-Service | ConvertTo-Json | Out-File -FilePath services.json

$object = Get-Content -Raw services.json | ConvertFrom-Json
```

Notice the use of the -Raw switch with Get-Content in the example above: the -Raw switch tells Get-Content to not try to parse the file into separate lines, but instead read the file byte-for-byte without parsing or interpretation. Always use the -Raw switch when reading a JSON text file. Unlike CSV files, a single object in a JSON file typically spans multiple lines, and the ConvertFrom-Json cmdlet expects one big string anyway.

The ConvertTo-Json cmdlet supports a -Depth switch just like Export-CliXml. This means that JSON files may contain complex, nested object data too. If your output seems to be missing information that you know should be there, make sure to capture enough layers of depth in your output.

The ConvertTo-Json cmdlet also supports a -Compress switch to eliminate all so-called whitespace character bytes, such as newlines and spaces, which are not a part of the data being converted. This reduces the size of large JSON files by roughly 50%. Compressed JSON data is saved to one long line and is not easy for humans to read in text editors, just like complex XML data is difficult for humans to read, but it does require less disk space.

```
Get-Service | ConvertTo-Json -Compress | Out-File oneliner.json

dir *.xml,*.csv,*.json | Sort length     #Notice size differences
```

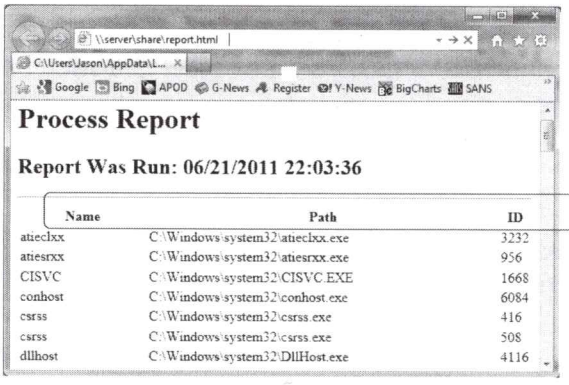
If you've run the commands in this section, please peruse the text files created so far to become acquainted with how the same data can be represented in different ways:

```
ise services.xml
ise services.csv
ise services.json
ise oneliner.json
```

If you know that your data will be submitted to a web application that expects JSON data, such as with the Invoke-WebRequest or Invoke-RestMethod cmdlets, then use the JSON cmdlets instead of the CSV or XML cmdlets. If you must save complex, nested object data, then don't use Export-CSV, use either Export-CliXml or ConvertTo-Json with the appropriate -Depth argument. If you must consume as little disk space as possible, prefer the ConvertTo-Json cmdlet with the -Compress switch. When in doubt, still prefer XML because it will morely closely match your expectations when manipulating data, and there is a greater variety of technologies associated with XML, such as the XPath query language and XSLT transformation of XML into other forms.

Enclave Consulting LLC © 2017 - All Rights Reserved

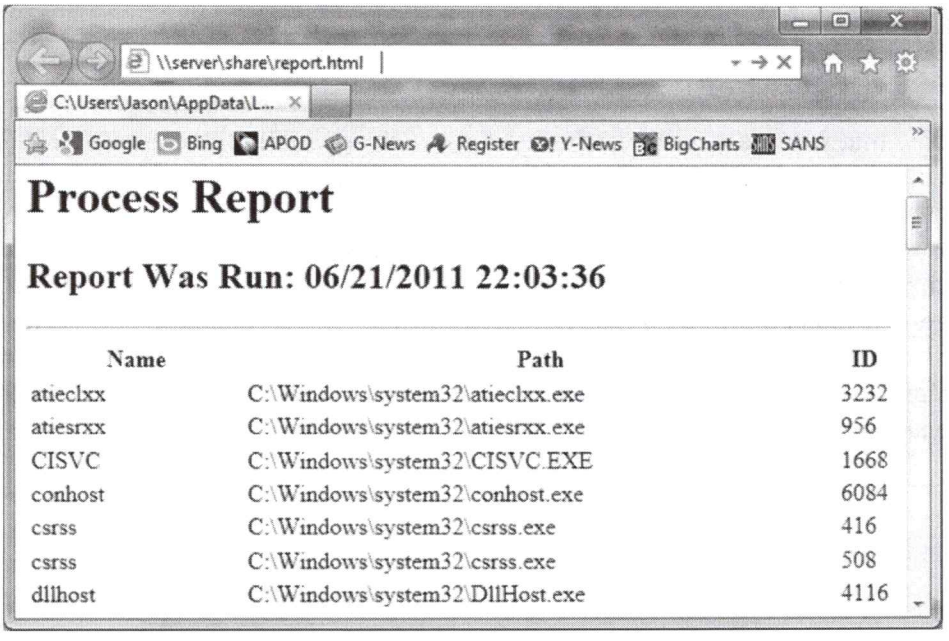
```
get-process | convertto-html -property Name,Path,ID
-title "Process Report" -head "<h1>Process Report</h1>"
-body "<h2>Report Was Run:(get-date)</h2><hr>" |
out-file -filepath \\server\share\report.html
```




SANS SEC505 | Securing Windows

# ConvertTo-HTML

Instead of saving data to a CSV or XML file, you can save it to a more human-readable HTML file and then show that file in a browser or copy it to a web server. The convertto-html cmdlet will create an HTML table where each column is a property of the objects piped into it and each row represents a separate object so piped. The output of the cmdlet should be redirected (">") to a file with the ".htm" extension (or use out-file). The cmdlet also has parameters for customizing the HTML with a title, header and additional body text (just remember to add the HTML markup yourself first).




Enclave Consulting LLC © 2017 - All Rights Reserved

To generate an HTML report of current processes and open that report in a browser using the invoke-item cmdlet on the HTML file produced (remember that the back tick character (" ` ") indicates that a command wraps to the next line):

```
# Example: HTML_Report.ps1

get-process |
convertto-html -property Name,Path,ID `
      -title "Process Report" `
      -head "<h1>Process Report</h1>" `
      -body "<h2>Report Was Run: $(get-date) </h2><hr>" |
out-file -filepath $env:temp\report.html

invoke-item $env:temp\report.html
```

And if you'd like to display your data in charts, gauges and other graphical formats, you might be interested in the cmdlets from http://www.softwarefx.com/powergadgets/.

Enclave Consulting LLC © 2017 - All Rights Reserved

## On Your Computer

Remote into your own computer:

```
enter-pssession -computername $env:computername
```

Find cmdlets related to shared folders:

```
get-help share
```

Show share information in a table with only the Name and Path properties:

```
get-smbshare | format-table -property Name,Path -autosize
```

How do you see all the properties of shared folders in a list?

```

```

How do you export shared folder objects to C:\SANS\shares.csv?

```

```

If you try to open shares.csv in a graphical application like Notepad, it does not work as expected. Why?

Enclave Consulting LLC © 2017 - All Rights Reserved

View the raw contents of the exported CSV file inside the shell:

```
get-content -path C:\SANS\shares.csv
```

Disconnect from the remoted session to get back to your local computer:

```
exit
```

Recreate the original objects from the exported CSV file with all of their properties:

```
import-csv -path C:\SANS\shares.csv
```

Run the same command (up-arrow on your keyboard) but capture the objects to an array:

```
$array = import-csv -path C:\SANS\shares.csv
```

Execute the name of the array by itself to see the objects inside of it:

```
$array
```

Save that array of objects to a JSON-formatted text file:

```
$array | convertto-json | out-file -filepath shares.json
```

How do you remotely execute a command to get shared folder objects and then export those objects to a local XML file named "shares.xml" on your own drive? (Hint: you will "remotely" execute the command at $env:COMPUTERNAME and then pipe the output into another cmdlet to save the XML file locally.)

Open all three text files to see how the same data may be saved in different formats:

```
ise shares.csv

ise shares.xml

ise shares.json
```

Close the ISE tabs with the CSV, XML and JSON files when you are done.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Cheat Sheet

See all the properties of shared folders in a list:

```
get-smbshare  | format-list *
```

Export shared folder information to a comma-delimited file: C:\SANS\shares.csv

```
get-smbshare | export-csv -path C:\SANS\shares.csv
```

If you try to open shares.csv in a graphical application like Notepad, it does not work as expected because you have a remoted connection.

Remotely execute the Get-SmbShare cmdlet and save the output as a local XML file:

```
invoke-command -computername $env:COMPUTERNAME -scriptblock
  { get-smbshare } | export-clixml -path shares.xml
```

## Finished Already?

Remember, you can *import* from XML and JSON files too, but only CSV files can be directly opened in Microsoft Excel by simply double-clicking the CSV file.

How do you import objects from an XML file and immediately export those objects to a new CSV file? Please convert shares.xml to exported.csv in one line of code.

Enclave Consulting LLC © 2017 - All Rights Reserved

**Select-Object (Alias: Select)**

**Similar to "SELECT" in SQL queries:**

```
get-process "powershell*" |
select-object ProcessName,ID,Path |
out-gridview
```

**Extract first, last and unique items only:**

```
get-service | select-object -first 5
get-hotfix  | select -last 3
get-content users.txt | select -unique
```

SANS SEC505 | Securing Windows

# Select-Object (Alias: Select)

When piping objects, you'll often want to work with only a subset of those objects' properties. Or, when one of those properties is an array of items, you'll want to get at just the array. Or, similarly, when you're piping an entire array of objects, you'll often want to work with just a subset of those objects. Fortunately, the select-object cmdlet offers easy-to-use methods of getting at just the properties and objects you want from a pipeline. Under the covers, the way the cmdlet works is to create new objects or new object arrays with just the properties/objects you select (so, even though the name of the cmdlet is select-*object*, but you're only interested in certain *properties*, you're still dealing with whole objects).

To show just one property (Path) of a piped object:

```
# Example: Select_Object.ps1

get-process "powershell*" | select-object -property Path
```

## When A Property Is An Array Of Objects

When a selected property is itself an array and you print that array out in the command shell, the array's contents are often truncated by your formatter to make it fit in the shell's window. But what if you want to show or otherwise manipulate the entire array?

The following command shows the Modules property of the PowerShell process object. This property is an array (notice the curly braces) of all the DLLs loaded into that processes' address space. But note that it doesn't show every item in that Modules array:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
get-process "powershell" | select-object modules
```

What we might want to do is to get just the Modules property and extract all the objects in it as separate objects for processing in a pipeline.

To show just one property (Modules) of a piped object, but, when that property is an array of objects, have the array's contents expanded to emit all the objects in it:

```
get-process "powershell" | select-object -expandproperty modules
```

Remember, if the contents of a property are shown in curly braces ("{ }") then that property is itself an array, and that property might even itself be an array of objects which have their own properties, some of which might be arrays of objects, etc. You can use multiple select-object cmdlets in a pipeline to finally extract what you need.

## First, Last and Unique Items

To only show *unique* event ID numbers from the Application event log:

```
get-eventlog Application | select-object EventID -unique
```

To select the last 10 events from the System event log:

```
get-eventlog -logname system | select-object -last 10
```

To select the first 5 services:

```
get-service | select-object -first 5
```

Of course, what counts as *first* or *last* is relative to how the data is sorted. If you don't like the default sorting, use the sort-object cmdlet.

## Select-String: PowerShell Version of GREP

PowerShell emphasizes the use and piping of entire objects, but you can still pipe and manipulate plain text just as well.

The select-string cmdlet was designed to be similar to UNIX `grep` and Windows `findstr.exe` to search text with regular expressions. The text to be searched can be read from a file or piped into the cmdlet.

To search all the *.inf files in %WinDir%\Inf for the pattern 'K5.*[efg]lex':

```
# Example: Select_String.ps1

select-string 'K5.*[efg]lex' $env:windir\inf\*.inf |
format-table path,line -autosize
```

Enclave Consulting LLC © 2017 - All Rights Reserved

To search the output of `ipconfig.exe` for all version 4 and 6 IP addresses, but change the pattern to be case sensitive (by default, case of letters is ignored):

```
ipconfig.exe | select-string 'IP.*Address' -casesensitive |
format-table line
```

You can also search the textual output of cmdlets, but you'll usually need to pipe the cmdlet's output through the out-string cmdlet first to convert the complex objects emitted into a single string object. Keep in mind, though, that out-string will produce one long string by default, which usually is not what you want when piping into select-string; instead, you'll usually want each line of output to be piped into select-string one at a time. To make out-string output text one line at a time, use the -stream switch.

To convert the output of get-process to text and search that text one line at a time:

```
get-process | out-string -stream | select-string "svchost|rundll"
```

**Does Select-String Output *Strings*? No!**
The output of select-string are *objects* representing the matches to the regular expression pattern. Give it a shot, pipe the output of a select-string command into get-member!

These objects have properties like the name of the file containing the match (.filename), the line number in that file with the match (.linenumber), the full path to that file (.path), and the matching line itself (.line). If the data is piped in instead of read from a file, the .path and .filename properties simply say "InputStream".

If you don't care what text actually matched the pattern, if you only care to know if at least one match was found, you can use the -quiet switch to make the cmdlet emit just either $true or nothing at all (interpreted as $false).

To show just the full paths to the files which match a pattern:

```
select-string 'p[at]t*rn' *.log | format-table path
```

To do a boolean test of some text, yielding either $true or nothing (used as $false):

```
netstat.exe -ano -p tcp | select-string ':139' -quiet
```

**But What If I Just Want The Strings Anyway?**
If you want just the strings which match a pattern (or just a submatch in the regex pattern), then pipe your strings-to-be-searched into this filter:

```
filter extract-text ($RegularExpression) {
 select-string -inputobject $_ -pattern $regularexpression -all |
 select-object -expandproperty matches |
 foreach {
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
    if ($_.groups.count -le 1) { if ($_.value){ $_.value } }
    else
    {
      $submatches = select-object -input $_ -expandproperty groups
      $submatches[1..($submatches.count - 1)] |
           foreach { if ($_.value){ $_.value } }
    }
  }
}
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Sort-Object (Alias: Sort)

**Sort by one or more properties:**

```
dir *.exe,*.dll,*.sys |
sort-object length,extension,name -desc |
select-object -first 20 |
format-table length,fullname -autosize
```

SANS SEC505 | Securing Windows

# Sort-Object (Alias: Sort)

The select-object and sort-object cmdlets are often used together: you select the properties or objects of interest to you, sort them in some way, and then perhaps select the first or last item(s) again. The sort-object cmdlet sorts in ascending order in a case-insensitive way by default, but the default can be changed. You can also sort objects on the basis of multiple properties just as a table or spreadsheet can be sorted on multiple columns, i.e., the piped objects are grouped by the first property specified, then, if there are any duplicates within a group, each group is next sorted by the second property into subgroups, and so on for each additional property specified for the sorting. If you don't specify any properties to sort on, the sort-object cmdlet will do its best to sort the objects depending on the type of objects they are, usually alphabetically on the name of the objects.

To get the smallest file in the root of the C: drive:

```
# Example: Sort_Object.ps1

dir c:\ | sort-object length | select -first 1
```

To get the largest file in the root of the C: drive:

```
dir c:\ | sort-object length -descending | select -first 1
```

To list the EXE's in the C:\Windows folder, sorted on the Last Access Time property:

```
dir c:\windows\*.exe |
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
sort-object lastaccesstime |
select-object name,lastaccesstime
```

To first sort all EXE and DLL files in C:\Windows\System32 by size, extension and name, all in descending order, and then to only show the full path and size of the first 20 such files:

```
dir c:\windows\system32\*.exe,c:\windows\system32\*.dll |
sort-object length,extension,name -descending |
select-object -first 20 |
format-table length,fullname -autosize
```

## Named Calculated Properties

Similar to SQL's capability to create a named field which is calculated from one or more other fields, both sort-object and select-object support the use of scriptblocks to more-or-less implement calculated fields as properties for the sake of sorting and selecting objects.

The calculated property is enclosed inside of "@{ … }" and within that the custom property is calculated within a another scriptblock that looks like "expression={ … }".

In the case of select-object, the scriptblock can also include a "name" element to name the property to be captured and displayed (the formatter will use that name as though it were a built-in property).

To show all the subkeys under HKEY_LOCAL_MACHINE\System\CurrentControlSet, sorted by a custom property which is the sum of the count of all values and subkeys in each key, then that custom property, named "Item Count", is selected along with the name of the registry key for display, but only the last 10 are shown, hence, only the subkeys with the most items inside them (values and subkeys) are shown:

```
get-childitem hklm:\system\currentcontrolset\control |
sort-object @{expression={$_.subkeycount + $_.valuecount}}|
select-object name,@{expression={$_.subkeycount + $_.valuecount};
name="Item Count"} -last 10
```

## Group-Object

A related cmdlet is group-object, which groups objects together based on one or more properties that you specify. For each property used to group objects together, you can get the count of items in that group.

To get the count of files under the C:\Windows\System32 directory for each unique filename extension found in that directory, sorting by count number, as long as the count is at least 10:

```
dir c:\windows\system32 -recurse |
group-object -property extension |
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
where-object {$_.count -gt 10} |
sort-object count -desc
```

But what is that "where-object" cmdlet? Sorting, grouping, calculated properties, and now *where clauses* -- this is looking almost like SQL!

Enclave Consulting LLC © 2017 - All Rights Reserved

## Where-Object (Aliases: Where, ?)

**$_ is a temp variable for each object piped in.**
**{ . . } is a test that must evaluate to true or false.**

```
get-service | where-object {$_.status -ne "running"}

dir hkcu:\  | where-object {$_.subkeycount -gt 2}

get-command | where {$_.name -like "*item*"}

get-hotfix  | where {$_.description -match "^Secur."}

ps | ? {($_.id -ge 100) -and ($_.name -like "svc*")}
```

SANS SEC505 | Securing Windows

# Where-Object (Aliases: Where, ?)

When piping objects from one cmdlet, function or script to another, you'll very frequently wish to limit or restrict exactly which objects are passed down the pipeline to the next command. The select-object cmdlet is easy to use, but very limited; for example, you might want to copy all the DLLs in a directory which are greater than a certain size and which were last modified during a certain 72-hour period, but you'll be frustrated to try to do this with just select-object.

To filter the objects moving through a pipeline with maximum flexibility, use the where-object cmdlet instead of select-object. The where-object cmdlet uses a scriptblock and the default variable to implement its filtering, so it's much more powerful than select-object.

As each piped object is given to the where-object cmdlet, that object is temporarily assigned to a variable named "$_" (and, no, that's not a misprint, the variable is a dollar sign followed by an underscore). This is called the "default variable" and you'll see it very frequently in PowerShell. The meaning or reference of the default variable changes with every object piped into where-object.

As each piped object is given to the where-object cmdlet, the cmdlet executes the code inside the curly braces. This chuck of code in the braces is called a "scriptblock". If the scriptblock evaluates to $true for a given piped object, then that object is passed along to the next command in the pipeline. If the scriptblock evaluates to $false for a given object, then that object is not passed along the pipeline, i.e., it's filtered out.

```
# This pipeline emits nothing, everything is filtered out:
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
get-service | where-object {$false}

# This pipeline filters nothing out, all services are shown:
get-service | where-object {$true}
```

**Note:** "Where" and "?" are aliases for the where-object cmdlet, and "-ne" is a comparison operator that means "not equal".

To only list the services which are not running:

```
# Example: Where_Object.ps1

get-service | where-object {$_.status -ne "running"}
get-service | where {$_.status -ne "running"}
get-service | ? {$_.status -ne "running"}
```

To show the full paths to every file under C:\Windows\System32 over 10MB in size:

```
get-childitem c:\windows\system32 -recurse |
where-object {$_.length -gt 10000000} |
sort-object length -desc |
format-table fullname,length
```

To get a listing of all commands for manipulating items of any type:

```
get-command | where-object {$_.name -like "*item*"}
```

To list the keys under HKEY_CURRENT_USER which have more than two subkeys:

```
get-childitem hkcu:\ | where-object {$_.subkeycount -gt 2}
```

To show in a list the last ten messages in the System event log which have the word "computer" in them and which were generated after August 18, 2013:

```
get-eventlog -logname system -newest 10 |
where-object {($_.message -match "computer") -and
($_.timegenerated -gt "8/18/2013")} | format-list
```

But, because the "$_" variable is kind of strange and ugly, in PowerShell 3.0 and later you can instead use "$psitem" wherever $_ is also permitted:

```
get-service | where { $psitem.status -ne "running" }        #v3.0+
```

Enclave Consulting LLC © 2017 - All Rights Reserved

**Arrays Are Like In-Memory Database Tables**

**Creating Arrays**

```
$empty = @()

$one = @("SANS")

$x = @(1,"hi",4.2,"bye")

$x += "AppendMoreStuff"

$big = @(0..50000)
```

**Using Arrays**

```
$big.count

$x          #Everything
$big[0]     #First
$big[1]     #Second
$big[-1]    #Last

$d,$f = $nums[3,7]
$another = $nums[3..209]
$last5 = $big[-1..-5]
```

SANS SEC505 | Securing Windows

# Arrays: Like In-Memory Database Tables

A "scalar" is a single object or it is a variable which contains or refers to a single object only. An "array" is a list of objects or it is a variable which contains or refers to a list of objects. An array can be empty, have one item, or it can have many items in it. Hence, while a scalar can only be a single object, an array can be zero, one or many objects. An item in an array is also called an "element" of that array. Arrays are sometimes called "collections" or "lists".

Arrays are used *very* frequently in PowerShell. Almost every script you'll ever write will use arrays. As a language, PowerShell supports a variety of advanced features for array creation and manipulation, such as arrays of arrays, multi-dimensional arrays, array slicing, range operators in indexes, etc. Understanding arrays is very important for understanding how to use PowerShell.

## Creating Arrays

To create an empty array with no elements inside it:

```
# Example: Arrays.ps1

$array = @()
```

To create an array with some elements in it, separate the elements with commas:

```
$array = @(1,"hello",433.2,"world")
```

Enclave Consulting LLC © 2017 - All Rights Reserved

The @-symbol forces data into an array, even if there is only one datum. Strictly speaking, the @-symbol is not necessary when it is obvious to PowerShell that there are multiple items being pushed into the variable, but it doesn't hurt to use it either. When you don't know whether there will be more than one item going into a variable, use the @-symbol to force the creation of an array so you can deal with the variable in a consistent way.

To create an array without explicitly casting it as an array, just use the comma operator, and PowerShell will know it's an array despite the lack of an @-symbol:

```
$array = 1,"hello",433.2,"world"
```

To fill an array with a series of numbers in a range (50,000 items maximum):

```
$x = 0..20000

$y = @(-2..150)     # @-symbol not necessary, just more explicit.

$z = -203..-4

$w = 58..50058      # This isn't too large, see where it starts.
```

To create an array with only one element, precede that item with a comma or place it within "@(...)":

```
$array1 = ,"Hello"

$array2 = ,67

$array3 = @("World")

$array4 = @(68)
```

To add or append more data to the end of an existing array, use the "+=" operator:

```
$array += "morestuff"
```

## Getting Data Out of Arrays

To show the number of items or elements in an array:

```
$array.count
```

To write each element of the array to a new line:

```
$array
```

Enclave Consulting LLC © 2017 - All Rights Reserved

To write all the elements of the array on one line, separated by spaces, cast the array as a *System.String* object from the .NET class library:

```
[String] $array
```

To write all the elements of the array separated by commas, use the -Join operator:

```
$array -join ","
```

The default separator when casting an array to a string is a single space character, but you can change that separator to be anything you wish. Whatever you assign to the $OFS variable will be used as the output field separator.

The elements in an array are numbered from zero to the last one. Again, the numbering starts at zero, not one; so if there are 20 items in an array, the first item is $item[0] and the last one is $item[19].

To retrieve the first and third items in an array:

```
$array[0]
$array[2]
```

To retrieve the last and next-to-last items in an array, use negative numbers:

```
$array[-1]              #Last Item

$array[-2]              #Next-to-Last Item

$array[-3]              #Next-to-Next-to-Last Item (and so on)
```

To retrieve a series of elements, of any type, from an array using a range:

```
$array[0..30]

$array[53..12000]
```

To get the last ten elements from an array using a range (last item output first):

```
$array[-1..-10]
```

To slice out both the first and last elements from an array, but nothing else:

```
$array[0,-1]
```

To slice out five particular elements and assign them to different variables:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
$v, $w, $x, $y, $z = $array[2, 30, 997, -32, -1]
```

## Nested Arrays (Array of Arrays)

To create a nested array, i.e., an array of arrays:

```
$array1 = 10,11
$array2 = 20,22
$arrayAA = $array1,$array2

$array4 = 40,44
$array5 = 50,55
$arrayBB = $array4,$array5

$bigarray = $arrayAA,$arrayBB
```

To extract the elements "10", "55" and "44" from the example above:

```
$bigarray[0][0][0]                # 10
$bigarray[1][1][1]                # 55
$bigarray[1][0][1]                # 44
```

## Associative Arrays (Also Known As "Hashtables" Or "Dictionaries")

An associative array, also known as a "hashtable" or "dictionary object", is similar to a standard array, but each element of the array is associated with a "key" instead of an index number. By analogy, the Webster's dictionary could be loaded into an associative array where each word is a key and the word's definition is paired with it.

In .NET, an associative array is of type *System.Collections.Hashtable* and its properties and methods can be listed with get-member like anything else.

To create an associative array where the two-letter U.S. state code is associated with the full name of the state (the two-letter code is the key, the full name is the value):

```
# Example: Associative_Arrays.ps1

$states = @{ "CA" = "California" ;
             "FL" = "Florida"    ;
             "VA" = "Virginia"   ;
             "MD" = "Maryland"
           }
```

To dump the entire associative array in tabular format, or just show the keys, or just show the values, or to test whether a particular key or value exists:

```
$states                                   # Dump entire hash array.
$states.keys                              # Dump keys only.
$states.values                            # Dump values only.
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
$states.containskey("VA")                    # Test if key exists.
$states.containsvalue("Virginia")            # Test if value exists.
```

To access the value associated with a key, just append ".*keyname*" (not case sensitive):

```
$states.CA
```

To add a new key-value pair, to remove a pair, or to remove everything in the array:

```
$states.add("CO", "Colorado")
$states.remove("CO")
$states.clear()
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Capturing Command Output To An Array

```
$output = get-history -count 50
$output = netstat.exe -ano
$output = python.exe myscript.py
$output = @(python.exe myscript.py)

$lines = @(get-content c:\file.txt)
$manylines = get-content *.txt,*.log,*.ini

$bytes = get-content worm.exe -encoding byte
```

SANS SEC505 | Securing Windows

# Capturing Output and File Contents to an Array

The output of cmdlets, functions, scripts and external commands can be captured to an array within PowerShell. You can also easily read the contents of a file into an array.

If you want to force data into an array, even if there is only one datum, surround that data or command with "@(…)" and then check the count property. This is very useful when the output or file may contain many or only one item at different times.

## Capturing Command Output Into An Array

To capture the output of a native binary or Python script into an array:

```
$output = python.exe .\myscript.py

$output = @(python.exe .\myscript.py)

$output = netstat.exe -ano
```

To capture the output of a cmdlet into an array, then pipe that array into get-member:

```
$output = get-eventlog -logname application -newest 10

$output | get-member
```

To run a PowerShell script and capture the objects or string data it outputs:

```
$output = .\Find_Big_Files.ps1
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
$output | format-table -auto length,fullname
```

## Reading File Contents Into An Array

When reading a file into an array, each line goes into a separate element. But if the file is non-textual, such as a DLL or EXE file, then it can be read one byte at a time, with each byte a separate item in the resulting array.

To read the line(s) of a text file into an array:

```
$lines = @(get-content c:\folder\somefile.txt)

$lines = get-content otherfile.txt

$lines = ${c:\must\use\full\path\to\file.txt}
```

To read all the lines of all the *.bat and *.ps1 files in the present working folder into an array, but exclude the files which begin with the letter "h":

```
$allfiles = get-content *.bat,*.ps1 -exclude "h*"
```

To read the first two bytes of an executable and print their ASCII characters ("MZ"):

```
$bytes = get-content file.exe -encoding byte -totalcount 2

foreach ($byte in $bytes) {[System.Convert]::tochar($byte)}
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## On Your Computer

```
cd C:\SANS\Day1-PowerShell\Examples

ise .\Windows_Event_Logs.ps1

ise .\Nmap.ps1
```

SANS SEC505 | Securing Windows

# On Your Computer

In the command shell window of PowerShell ISE, please switch to the Examples subdirectory (this must be your working folder, i.e., your present working directory):

```
cd C:\SANS\Day1-PowerShell\Examples
```

Use the "ise" alias to open the following scripts in the PowerShell ISE editor:

```
ise .\Windows_Event_Logs.ps1

ise .\Nmap.ps1
```

In PowerShell ISE, if you open a script for editing, you can highlight one or more lines from that script and execute just those highlighted lines. Once the desired lines are highlighted, click the "Run Selection" button or press F8. The "Run Selection" button in the toolbar looks like a script with a green triangle on top of it. Don't confuse this with the "Run Script" button just to the left, which looks like a big green triangle; the "Run Script" button (or F5) will run the entire script, not just the highlighted lines.

Over the next several slides we will look at example code which you can highlight and run instead of typing in by hand.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Search Event Logs

**Get-WinEvent**

- Search local or remote event logs.
- Filtering performed at the remote server, not locally.
- Can query archived log files off-line (*.evtx files).
- Supports the **XPath** query language:
  - Very flexible and precise query syntax.
  - Build your XPath query in Event Viewer.
  - Lots of XPath tutorials on the Internet.

SANS SEC505 | Securing Windows

# Search Event Logs

You can use PowerShell to query the event logs on local or remote computers. The output of the queries can then be filtered, searched, consolidated and perhaps saved to a CSV file or HTML report.

There are two main cmdlets for event log queries: get-winevent and get-eventlog. The newer one is get-winevent, and this cmdlet should be used whenever possible because of its greater power. The older cmdlet, get-eventlog, is mainly for Windows XP and Server 2003. Both will be demonstrated here.

The get-winevent cmdlet can query classic event logs (System, Security and Application) as well as the many new logs found on Server 2008 and later. Logs can also be queried after they have been exported to a file; you do not always have to query "live" logs. The get-winevent cmdlet can also query logs generated by Event Tracing for Windows (ETW), but ETW is a complex topic and cannot be discussed here.

Importantly, query filtering with get-winevent is performed at the target computer, not locally, which is much faster than downloading an entire (possibly multi-gigabyte) log and filtering it locally afterwards. We want to push the query filtering work out to the remote computer whenever possible.

## XPath Queries

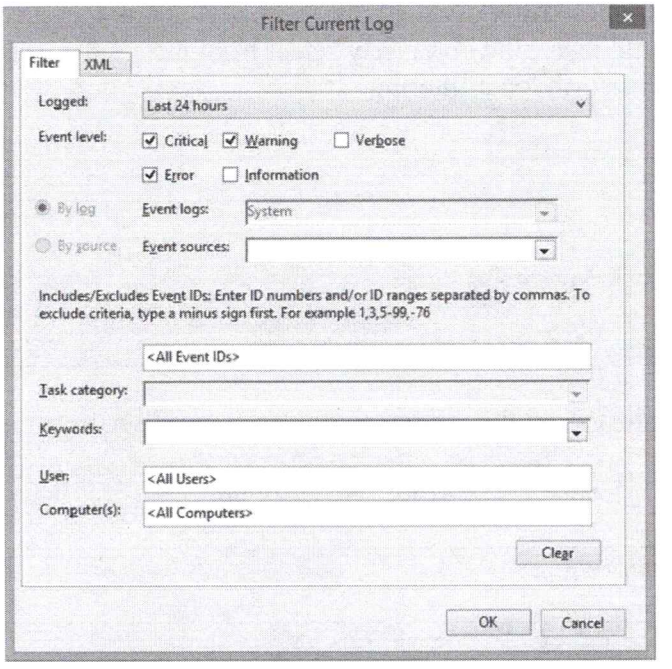
Queries with get-winevent can be defined using a simple hashtable, an XPath query, or an XML definition of one or more XPath queries performed simultaneously. The XPath query syntax is complex, but we can use the graphical Event Viewer application to build

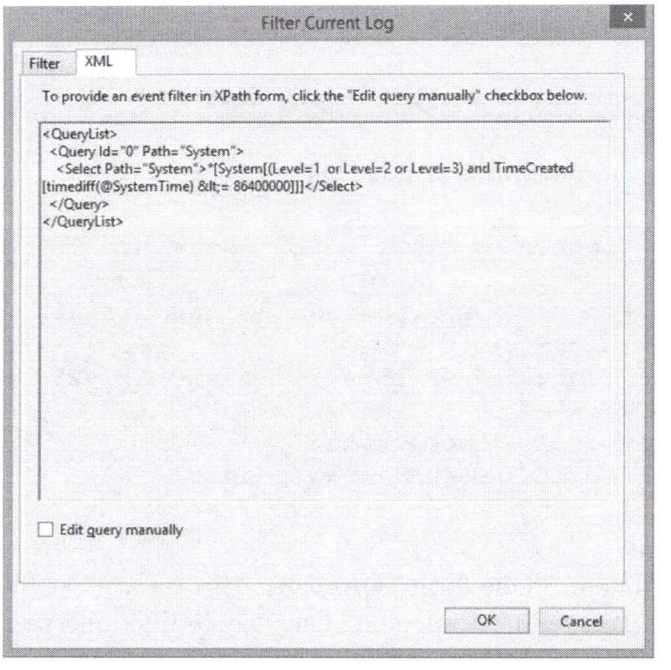
Enclave Consulting LLC © 2017 - All Rights Reserved

these queries for us. Once Event Viewer constructs the XML for the query, that XML string can simply be pasted into our scripts. Don't worry, it sounds much worse than it is.

***Try It Now!***

To construct an XPath query, open Event Viewer > right-click a log > Filter Current Log > select what you want on the Filter tab > go to the XML tab > highlight the XML text > press Ctrl-C to copy the text to the clipboard > Cancel button.




Enclave Consulting LLC © 2017 - All Rights Reserved

There are many tutorials on XPath on the Internet if you would like to edit your queries or write them from scratch. Just do a search on "xpath syntax" or "xpath examples" to get started. The syntax is complex, but, on the other hand, you will be able to perform very sophisticated queries.

Also, don't forget that after you have obtained your event log objects (with or with an XPath query), you can always filter these results locally with where-object like always. In fact, this will be a common strategy: filter out the bulk of what you're not interested in at the target server using an XPath query, then precisely select from the results just what you need using where-object to perform local filtering.

## Get-WinEvent Examples

To see a list of all local event logs:

```
# Example: Windows_Event_Logs.ps1

get-winevent -listlog *
```

To see a list of logs that begin with "s" on the remote computer named "server47":

```
get-winevent -listlog s* -computername "server47"
```

To see the details of just the Security log:

```
get-winevent -listlog security | format-list *
```

To show the last 20 events from the System log:

```
$logdata = get-winevent -logname system -maxevents 20
```

To get the last 20 events from each of the three classic logs:

```
$events  = get-winevent -logname system      -maxevents 20

$events += get-winevent -logname application -maxevents 20

$events += get-winevent -logname security    -maxevents 20

$events | sort-object -property TimeCreated |
   format-table TimeCreated,ID,Level,Message -auto
```

### Get-WinEvent -FilterHashTable

A hash table is an array of paired items in the form "*<property>* = *<value>*", with each pairing separated from the other pairs with a semicolon. One way to filter query results is to give the target computer a hash table of filters, such that only the events which match the properties and values in the filters will be returned. Here is an example.

Enclave Consulting LLC © 2017 - All Rights Reserved

To only show events with ID number 4624 from the Security log:

```
get-winevent -filterhashtable @{LogName="Security"; ID=4624}
```

> **Note:** You cannot use wildcards in -FilterHashTable query values, with the exceptions of log names and provider names. Sorry.

How would you know which properties can be used in a hash table filter? Unfortunately, not every property of an event entry displayed by get-member can be used in a hash table filter (unlike filtering with where-object). The help for get-winevent gives the ugly details, but these are the properties you will most likely wish to use in filters (they are not case sensitive):

- LogName
- ProviderName
- ID
- Level
- StartTime
- EndTime

The "Level" property is a code number for one of the following logging levels:

- Level 1 = Critical
- Level 2 = Error
- Level 3 = Warning
- Level 4 = Information
- Level 5 = Verbose

The "StartTime" and "EndTime" properties will take datetime objects, like those created by the get-date cmdlet. Here is an example.

To only show Security log events between five and three days ago:

```
$Day5Ago = (get-date).AddDays(-5)

$Day3Ago = (get-date).AddDays(-3)

get-winevent -filterhashtable `
     @{LogName="Security"; StartTime=$Day5Ago; EndTime=$Day3Ago}
```

To only show Warning events from the Application log:

```
get-winevent -FilterHashtable @{LogName="Application"; Level=3}
```

To only show 10 recent Critical, Error and Warning events from the System log on a remote computer named "server47.testing.local":

Enclave Consulting LLC © 2017 - All Rights Reserved

```
get-winevent -FilterHashtable @{LogName="System"; Level=@(1,2,3)}
   -MaxEvents 10 -ComputerName "server47.testing.local"
```

To list the last 10 user accounts created (notice the "+=" in the second command):

```
$events = get-winevent -FilterHashtable `
     @{LogName="Security"; ID=4720} -ErrorAction SilentlyContinue

$events += get-winevent -FilterHashtable `
      @{LogName="Security"; ID=624} -ErrorAction SilentlyContinue

$events | select-object -last 10
```

Why silently continue in the above example when errors occur? Unfortunately, the cmdlet returns an error when there are no matching events to a query. It should just return nothing if there are no other problems, but that's how it (doesn't) work.

**Get-WinEvent -FilterXML**

To use an XPath query constructed from Event Viewer or written by hand, the XML should be pasted into a literal here-string inside your script.

To show just Critical, Error and Warning events from the System log in the last 24 hours:

```
$CopiedFromEventViewer = @'
  <QueryList>
    <Query Id="0" Path="System">
       <Select Path="System">*
          [System[(Level=1 or Level=2 or Level=3)]]
       </Select>
    </Query>
  </QueryList>
'@

get-winevent -FilterXML $CopiedFromEventViewer |
  export-csv -Path .\searchresults.csv
```

## Writing And Clearing Events

PowerShell 2.0 and later includes other cmdlets for working with Windows event logs too, though we cannot discuss their details here, please view their help text:

- Clear-EventLog
- Write-EventLog
- Limit-EventLog
- Show-EventLog
- New-EventLog
- Remove-EventLog

Enclave Consulting LLC © 2017 - All Rights Reserved

To clear the System and Application logs on a remote computer named Server57:

```
clear-eventlog -log system,application -computername Server57
```

## Appending To Your Own Textual Logs (Add-Content)

Of course, you can also just write to your own custom comma-delimited log file instead, if you find that easier to parse, search, edit, compress, and archive. You can append text to a textual log file with the add-content cmdlet. Then, if you only append comma-delimited text, you can easily parse that logfile with the import-csv cmdlet and other tools.

## Obsolete: Get-EventLog

There is an older cmdlet named "Get-EventLog". The get-eventlog cmdlet is obsolete, but it is backwards compatible with Windows XP and Server 2003. Keep in mind that a query of a remote machine results in that entire log being downloaded before the query is performed. Any filtering being performed is done locally, not at the target system. By contrast, using get-winevent or the WMI service to query remote logs will result in the filtering being performed at the remote computer. Local filtering is slower and consumes more bandwidth than doing the filtering at the remote source.

To see a list of event logs and show their important configuration settings:

```
# Example: Event_Logs.ps1

get-eventlog -list

get-eventlog -list -computername Server57
```

To show the last 20 events from the System log:

```
get-eventlog -log system -newest 20
```

To show only warning and error events from the last 500 events in the System log:

```
get-eventlog -log system -newest 500 |
where-object {$_.EntryType -match "^Warning|^Error"}
```

The security log can be used to identify many interesting events by their unique event ID number. To list the last 10 user accounts created:

```
get-eventlog -log security |
where-object {$_.EventID -match "^624$|^4720$"} |
sort-object -property TimeGenerated |
select-object -last 10
```

To list all system log events between 72 and 48 hours ago, sorted by time:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
get-eventlog -log system |
where-object {$_.TimeGenerated -gt (get-date).AddHours(-72)
 -and
$_.TimeGenerated -lt (get-date).AddHours(-48)} |
sort-object -property TimeGenerated
```

To get the last 20 events from all three primary logs, append all these events to the $events array ("+=" appends), sort that array by time, and then show only a few essential properties in an automatically-formatted table:

```
$events  = get-eventlog -log system      -newest 20

$events += get-eventlog -log application -newest 20

$events += get-eventlog -log security    -newest 20


$events |
sort-object -property TimeGenerated |
format-table TimeGenerated,EventID,EntryType,Source -auto
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Parsing Nmap XML Output

### Nmap port scanner (www.nmap.org)

- Can identify listening ports, service versions, operating system fingerprints, and more.
- Used for inventories and penetration testing.
- Save your enterprise-wide port scans to XML files.

### Parse-Nmap.ps1 Script:

```
.\parse-nmap.ps1 samplescan.xml -runstats

dir *.xml | .\parse-nmap.ps1
```

SANS SEC505 | Securing Windows

# Parsing Nmap XML Output

The Nmap port scanner is probably the best and most popular port scanner in the world on both Windows and Linux systems (www.nmap.org). One great thing about Nmap is that it can produce XML output of the results of its scanning and OS fingerprinting work. But how can these XML files be conveniently handled from the command line when you want to query and extract specific data or to convert that data into other formats like CSV or HTML files?

PowerShell has great XML-handling capabilities, so let's look at a script which takes Nmap XML as input and converts that data into PowerShell objects where each object represents a host on the network and the properties of those objects contain the data from the scan. Once we have our scan results as an array of PowerShell objects, it becomes easy to pipe those objects into other cmdlets and scripts.

Don't forget that you must give the full path to the script at the command line, and that the current working directory is ".\", hence, the path to the script in the same folder as your present working directory would be ".\parse-nmap.ps1".

To process just one XML log file, which is on the CD-ROM by the way:

```
# Example: Nmap.ps1

.\parse-nmap.ps1 -path samplescan.xml
```

And then the output will look like the following, but remember, this output is not really text, the output is an array of .NET objects with properties!

Enclave Consulting LLC © 2017 - All Rights Reserved

```
FQDN            : srv-4mm1vr3.sans.org
HostName        : srv-4mm1vr3
Status          : up
IPv4            : 10.54.23.2
IPv6            : <no-ipv6>
MAC             : <no-mac>
Ports           : open:TCP:135:msrpc open:TCP:139:netbios-ssn
OS              : Microsoft Windows 8 <95%-accuracy>

FQDN            : wks-88d70.sans.org
HostName        : wks-88d70
Status          : up
IPv4            : 10.54.23.3
IPv6            : <no-ipv6>
MAC             : <no-mac>
Ports           : open:TCP:135:msrpc open:TCP:139:netbios-ssn
OS              : Microsoft Windows Server 2012 <93%-accuracy>

FQDN            : srv-9ryc3.sans.org
HostName        : srv-9ryc3
Status          : up
IPv4            : 10.54.23.4
IPv6            : <no-ipv6>
MAC             : <no-mac>
Ports           : open:TCP:135:msrpc open:TCP:139:netbios-ssn
OS              : Microsoft Windows XP SP2 <100%-accuracy>
```

To process a bunch of XML log files and consolidate their output:

```
dir *.xml | .\parse-nmap.ps1
.\parse-nmap.ps1 -path *.xml
```

To extract the IP addresses and hostnames of the machines fingerprinted as running XP:

```
# Example: nmap.ps1

.\parse-nmap.ps1 -path samplescan.xml |
where {$_.OS -like "*Windows XP*"} |
format-table IPv4,HostName,OS
```

To find the hosts listening on TCP/23 for telnet:

```
.\parse-nmap.ps1 -path samplescan.xml |
where {$_.Ports -like "*open:tcp:23:*"}
```

The "-like" operator is for simple wildcard matching, while the "-match" operator is for regular expression matching. Note that the Ports property is a space-character-delimited

Enclave Consulting LLC © 2017 - All Rights Reserved

list of scanned ports in the format of *state:protocol:portnumber:servicename* for each port in the list (see the Nmap documentation for more explanation at http://nmap.org).

To find the hosts listening on either TCP/80 or TCP/443 or both:

```
.\parse-nmap.ps1 -path samplescan.xml |
where {$_.Ports -match "open:tcp:[80|443]:"}
```

## Export to CSV or HTML

To export the data to a CSV file for grepping, import into a spreadsheet, etc.:

```
.\parse-nmap.ps1 -path samplescan.xml |
where {$_.Ports -match "open:tcp:80:"} |
export-csv .\weblisteners.csv
```

Then you could later re-import from the CSV file into a new array, keeping the same properties as before for the sake of further processing:

```
$data = import-csv .\weblisteners.csv
$data | where {($_.IPv4 -like "10.57.*") -and ($_.Ports -match
"open:tcp:22:")}
```

To export the processed data to an HTML file for viewing in a browser:

```
.\parse-nmap.ps1 -path samplescan.xml |
select-object IPv4,HostName,OS |
ConvertTo-Html -title "Scan Results" |
out-file \\server\sharedfolder\report.html
```

## Command-Line Only Processing (Not Using The Script)

You don't need the script to process Nmap XML output files from within PowerShell. Though it's not as easy as using the script, you do get access to all the XML data, so here's a taste of what's possible.

First, load the XML output into a variable and then show some scan stats for grins:

```
[XML] $output = get-content .\samplescan.xml

$output.nmaprun.profile
$output.nmaprun.args
$output.nmaprun.options
$output.nmaprun.startstr
$output.nmaprun.runstats.hosts
```

The <host> nodes in the XML contain the data for each scanned machine, and this is where you'll spend most of your time mining data. We can't go into a lot of detail about

Enclave Consulting LLC © 2017 - All Rights Reserved

XML handling here, there are full discussions elsewhere, so just follow along to get a feel for it.

To see the properties available on each host object:

```
$output.nmaprun.host | get-member -membertype property
```

To extract the first IP address of each host node (address[] is an array):

```
$output.nmaprun.host | foreach { $_.address[0].addr }
```

To extract the first IP address of the seventh host in the host[] array:

```
$output.nmaprun.host[6].address[0].addr
```

To extract the ports information for the seventh host in the host[] array:

```
$output.nmaprun.host[6].ports.port
```

To extract the hostname discovered for the seventh host in the host[] array:

```
$output.nmaprun.host[6].hostnames.hostname.name
```

To extract all the hostnames from all the host objects:

```
$output.nmaprun.host | foreach { $_.hostnames.hostname.name }
```

To extract all the hostnames and IP addresses and filter the output with a regular expression:

```
$output.nmaprun.host |
foreach {$_.hostnames.hostname.name + "," + $_.address[0].addr} |
select-string '^wks|^srv'
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Today's Agenda

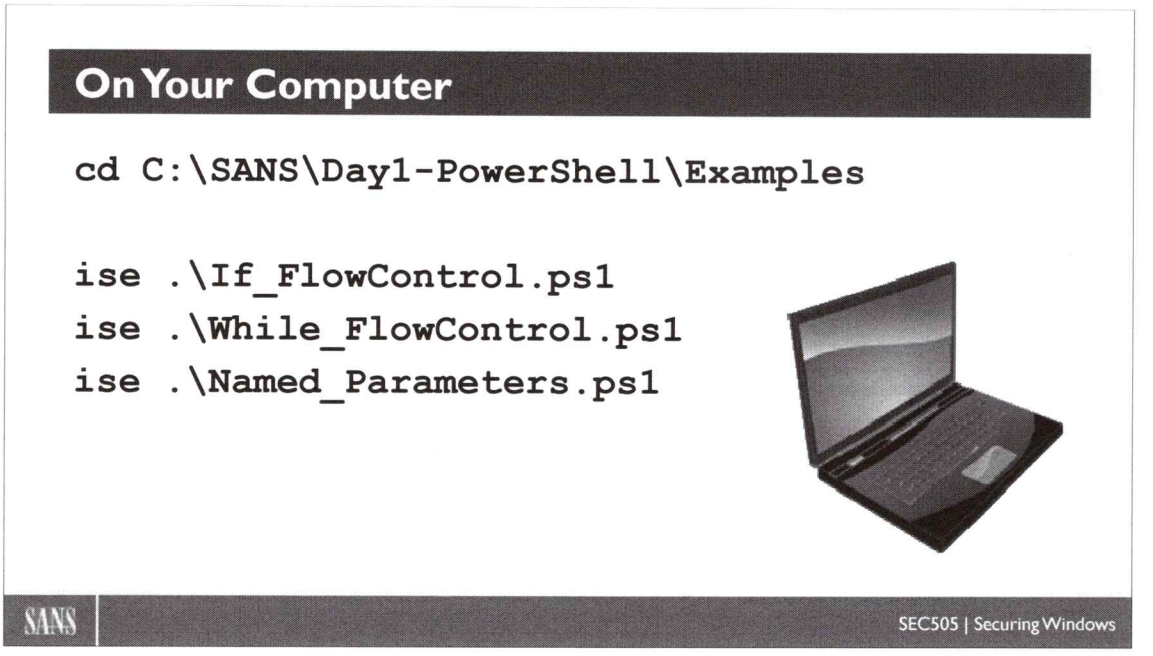
1. Setting Up Our Training VM
2. PowerShell Overview & Tips
3. Modules, Functions & PowerShell Remoting
4. Essential Commands & Scripting Examples
5. Writing Your Own Functions & Scripts

SANS | SEC505 | Securing Windows

# Today's Agenda

Having mastered the shell, you're now chomping at the bit to start writing scripts! Well, let's get to it then! In this section we'll cover the important flow-control constructs and how to write functions.

Enclave Consulting LLC © 2017 - All Rights Reserved




# On Your Computer

In the command shell window of PowerShell ISE, please switch to the Examples subdirectory (this must be your working folder, i.e., your present working directory):

```
cd C:\SANS\Day1-PowerShell\Examples
```

Use the "ise" alias to open the following scripts in the PowerShell ISE editor:

```
ise .\If_FlowControl.ps1

ise .\While_FlowControl.ps1

ise .\Named_Parameters.ps1
```

In PowerShell ISE, if you open a script for editing, you can highlight one or more lines from that script and execute just those highlighted lines. Once the desired lines are highlighted, click the "Run Selection" button or press F8. The "Run Selection" button in the toolbar looks like a script with a green triangle on top of it. Don't confuse this with the "Run Script" button just to the left, which looks like a big green triangle; the "Run Script" button (or F5) will run the entire script, not just the highlighted lines.

Over the next several slides we will look at example code which you can highlight and run instead of typing in by hand.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Flow Control: *If-ElseIf-Else*

```
If ($string -like "SANS*")
{ "It's true that it starts with SANS." }
ElseIf ($string -match "[FGH]IAC")
{
   "It matches the regular expression pattern."
}
Else
{
   "We don't know what it is, so we're giving up."
}
```

SANS SEC505 | Securing Windows

# Flow Control: *If-ElseIf-Else*

All the standard flow control statements are available in the PowerShell language, such as *if-elseif-else*, *while*, *do-while*, *for*, *foreach*, *switch*, and so on. In this section we'll look at a few examples of each, and then the rest of the course will demonstrate them.

The *if-elseif-else* statement is the most common you'll use. The test performed in the parentheses need only evaluate to either $true or $false, and can be an entire pipeline. In the simplest case, the statement(s) in the curly braces are executed only if the test resolves to $true. You can put it all on one line and, if the braces contain multiple commands, separate those commands with semicolons.

```
# Example: If_FlowControl.ps1

If ($x -eq 32) { $x = $x + 19 ; $x }
```

Or you can spread the flow control statement across multiple lines, with or without the semicolons, and optionally indent the code in the braces for easier readability:

```
If ($x -eq 32)
{
      $x = $x + 19
      $x
}
```

If you want one block of code executed if the test is $true and a different block executed if the test resolves to $false, then use the *Else* keyword.

Enclave Consulting LLC © 2017 - All Rights Reserved

```
If ( (get-date).dayofweek -eq "Monday" )
{
      "Today is " + [DateTime]::today
}
Else
{
      "Today is not Monday"
}
```

If you want additional tests performed before defaulting to the *Else* block, then add one or more *ElseIf* statements each with its own test in parentheses:

```
If ($string -like "SANS*")
{
      "It's true that it starts with SANS."
}
ElseIf ($string -match "[FGH]IAC")
{
      "It matches the regular expression pattern."
}
Else
{
      "We don't know what it is, so we're giving up."
}
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Flow Control: *While*

```
$rabbits = 2

While ($rabbits -lt 10000) {
     "We now have $rabbits rabbits!"
     $rabbits = $rabbits * 2
}
```

SANS SEC505 | Securing Windows

# Flow Control: *While*

To repeatedly execute a block of statements as long as the test in parentheses yields $true, use the *While* loop. If the first time the test in parentheses is evaluated results in $false, the entire *While* loop block in braces is skipped over and the script continues on.

```
# Example: While_FlowControl.ps1

$rabbits = 2

While ($rabbits -lt 10000) {
     "We now have $rabbits rabbits!"
     $rabbits = $rabbits * 2
}
```

You could also loop forever or until you hit Ctrl-C, whichever comes first…

```
While ($true) { ping.exe 127.0.0.1 }
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Flow Control: *Do-While*

```
$rabbits = 20000

Do {
   "We now have $rabbits rabbits!"
   $rabbits *= 2
} While ($rabbits -lt 10000)
```

SANS SEC505 | Securing Windows

# Flow Control: *Do-While*

The *While* loop won't execute its block if its test turns out $false. But what if you want the block of code to run at least once before evaluating the test condition? The *Do-While* loop is similar to the *While* loop, but the *Do-While* loop will always execute at least once. After executing the first time, the scriptblock of the *Do-While* loop will continue to be executed as long as its test yields $true.

```
$rabbits = 2

Do
{
      "We now have $rabbits rabbits!"
      $rabbits *= 2
}
While ($rabbits -lt 10000)
```

> **Note:** The "*=" operator multiplies the left operand by the right, then the contents of the left operand (always are variable) is replaced with this product.

And in all the looping flow-control statements, use of the keyword "break" will cause an immediate exit from the loop, and use of the keyword "continue" will not exit the loop but simply go to the next item through which we are looping.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Flow Control: *ForEach*, *For*, and *Switch*

```
$services = get-service

ForEach ($x in $services)
{
   $x.name + " : " + $x.status
}



get-service | ForEach { $_.name }
```

SANS SEC505 | Securing Windows

# Flow Control: *ForEach*, *For*, and *Switch*

There is a cmdlet named "foreach-object" (it has two aliases: "foreach" and "%"). That's not what we're talking about yet though. The *ForEach* loop is a syntactic programming construct, similar to but different from the foreach-object cmdlet. When the word "foreach" appears in the middle of a pipeline, it's treated as an alias for the foreach-object cmdlet. When the word "foreach" appears at the beginning of a statement, which means nothing is being piped into it, the word is treated as the beginning of a *ForEach* loop.

> **Note:** To repeat, "foreach" is both an alias for the foreach-object cmdlet and a keyword for the beginning of the *ForEach* loop construct.

The *ForEach* loop is for enumerating through all the items in an array or collection (but not a hashtable) without having to manage counters or index numbers like with the *For* loop. You'll use this loop frequently.

```
# Example: ForEach_FlowControl.ps1

$services = get-service

ForEach ($x in $services)
{ $x.name + " : " + $x.status }

"The last service is " + $x.name       # $x is not local!
```

The variable in the parentheses which is used to temporarily hold each item in the collection/array being enumerated is not a local variable. It is just another regular variable in the scope of the current session, function or script; hence, when you're done

Enclave Consulting LLC © 2017 - All Rights Reserved

enumerating, the variable will continue to exist outside the *ForEach* loop and will refer to the last item in the collection/array.

You can use a pipeline to build the collection/array to be enumerated through, and, especially if the pipeline might yield only a single object, it's generally best to wrap the pipeline with "@(...)" to ensure that the output is an array (even if it's only an array with a single element).

```
ForEach ($x in @(dir c:\ | where {-not $_.psiscontainer})) {
      $x.name + " : " + $x.length / 1024 + "KB"
}
```

And in all the looping flow-control statements, use of the keyword "break" will cause an immediate exit from the loop, and use of the keyword "continue" will not exit the loop but simply go to the next item through which we are looping.

There is also the compressed version of ForEach-Object in PowerShell 3.0 and later:

```
ps | foreach { $_.path }     #The traditional way of doing it.
(ps).path                    #Requires PowerShell 3.0 or later
ps | foreach path            #Requires PowerShell 3.0 or later
```

## Flow Control: *For*

There is another flow control statement which has a similar name: *For*. The *For* loop gives you precise control over how to initialize starting values in a loop, how many times you loop through the scriptblock, how to move from one item to the next in an array or collection, and how to exit the loop.

Here's an example of a basic *For* loop:

```
# Example: For_FlowControl.ps1

For ( $i = 0 ; $i -le 20 ; $i++ ) {
      "Now at $i"
}
```

The parentheses following the *For* has three sections separated by semicolons:

- "$i = 0" -- **Initialize Section**. The initialization section is only executed once when the *For* loop is first executed. It is often used to set the initial values of local variables. If this section is to contain multiple statements, put the section in "$( ... )" and separate the statements with semicolons. You can use pipelines. Any output produced by the statement(s) in this section will be silently discarded.

- "$i -le 20" -- **Test Section**. The test section is evaluated every time through the loop *before* the block of code is executed, even on the first run. While the test

Enclave Consulting LLC © 2017 - All Rights Reserved

section evaluates to $true, the loop will continue. If this section is to contain multiple statements, put the section in "$( … )" and separate the statements with semicolons. You can use pipelines. Any output produced by the statement(s) in this section will be silently discarded. Had the above for loop been initialized like "$i = 21", then the test would evaluate to $false and the block of code would not run even once.

- "$i++" -- **Change Section**. The change section is executed every time through the loop *after* the block of code executed. It is normally used to change a variable which is used in the test section. If this section is to contain multiple statements, put the section in "$( … )" and separate the statements with semicolons. You can use pipelines. Any output produced by the statement(s) in this section will be silently discarded.

A more complicated *For* loop might separate the three initialize/test/increment sections across multiple lines, each inside its own "$(…)" subexpression, followed by the scriptblock's curly braces on new lines too:

```
For ( $( $i=0; $j=0; $ps=@(get-process|select-object name));
      $( $ps.count -ge ($i + $j) ) ;
      $( $i += 2 ; $j++ )
    )
{
      $ps[$i]
}
```

And in all the looping flow-control statements, use of the keyword "break" will cause an immediate exit from the loop, and use of the keyword "continue" will not exit the loop but simply go to the next item through which we are looping. The "exit" keyword will terminate the script.

## Flow Control: *Switch*

*Switch* is a very powerful and useful flow-control statement. It supports wildcard matching, regular expression matching, and is very nice for processing the lines of a text file.

In a *Switch* statement, a value to be compared against is placed in parentheses, then that value is compared against the test clause in curly braces on each line. If a match is made, the code in the subsequent braces is executed. If the parentheses contain an array, or a command that produces an array, then every object in the array is evaluated one-by-one.

If the match is exact, like "58 = 58", then the condition doesn't have be placed in curly braces or use a boolean comparison operator. Inside the *Switch* statement, the value to be compared against is assigned to the default variable ( $_ ).

```
# Example: Switch_FlowControl.ps1
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
$x = 58
switch ( $x )
{
   {$_ -lt 20} {"Really Small"}

   {$_ -gt 50} {"Pretty Big"}

   58 {"It's 58"}

   default {"What was that?"}
}
```

Note in the output above that *all* matches are run, not just the first one, but the default match doesn't run if at least one other match is made. If no other matches are made, then the default entry runs.

And in all the looping flow-control statements, use of the keyword "break" will cause an immediate exit from the loop, and use of the word "continue" will not exit the loop but simply go to the next item through which we are looping. Hence, use "break" or "continue" if you only want to match one item.

**-Case**

By default, text comparisons are performed without regard to the case of the letters. If you want to perform a case-sensitive match, place the word "-case" after the "switch" keyword, e.g., "`switch -case ( $x )`".

**-Wildcard**

Another option is "-wildcard", which converts the value to be compared against to a string and then the comparisons can be performed using the "?" and "*" wildcards. In the following example, both comparisons match the string with the file path.

```
switch -wildcard ("c:\data5\archive.zip")
{
      '?:\data?\*'
      {"In some data folder."}


      '*.zip'
      {"File is a ZIP."}
}
```

**-RegEx**

Even better is the "-regex" option, which performs matching against regular expression patterns against one or more strings. If your regex pattern includes submatches, you can access them in the $matches hashtable (see the Regular Expressions section of this manual).

Enclave Consulting LLC © 2017 - All Rights Reserved

```
switch -regex ("c:\data5\archive.zip")
{
      '\\data[0-9]+\\'        {"In some data folder."}

      '\.ZIP$|\.BKF$|\.TAR$' {"File is a ZIP or BKF or TAR."}
}
```

**Processing Arrays With *Switch***

If the value to be compared against is an array or a pipeline that yields an array, the *Switch* statement will enumerate through every item in the array. This is similar to *ForEach*, but you get the additional comparison and processing power of *Switch*.

```
# Example: Processing_Arrays.ps1

$stopped = $running = $paused = 0

switch (get-service ) {
      {$_.status -like "Running"} {$running++}
      {$_.status -like "Stopped"} {$stopped++}
      {$_.status -like "Paused"}  {$paused++}
}

"Services Running = $running"
"Services Stopped = $stopped"
"Services Paused = $paused"
```

While enumerating through an array, if you want to stop comparing the current item from the array against any further conditions, use the "continue" keyword to cause *Switch* to move on to the next item in array and start comparing from the top again.

Similarly, if you have a successful match, the value of that match is stored in the "$switch.current" property, then you can call "$switch.movenext()" and retrieve the value of the *next* item with "$switch.current" again. This is handy when processing data that's formatted in a key-value fashion, such as command line parameters and their arguments.

If you want to stop processing the *Switch* statement entirely by breaking out of the loop, use the "break" keyword.

**Processing Text Files With *Switch***

The *Switch* statement can also be used to enumerate through the lines of a text file using the "-file" option. This is a very efficient way to process large log files because only 1000 lines are read into memory at a time even if the file is gigabytes in size. You can also combine the "-regex" and "-file" options together.

In the following example, an empty hashtable is created ($HashTable = @{}). An hashtable is created with curvy braces instead of parentheses. A hashtable is an array of paired items of the form "Key=Value" where the Key may be any string, such as an IP address, and the Value may be any other data, such as an integer number or another

Enclave Consulting LLC © 2017 - All Rights Reserved

string. A phonebook, for example, is a hashtable which pairs names (Keys) with phone numbers (Values).

In the example below, each line from the Windows Firewall log file (pfirewall.log) is compared against a regular expression pattern ("DROP\sTCP.+RECEIVE"). If the line from the log matches the pattern, then we split out the source IP address from the text of the line (the source IP is the fifth item in the space-delimited line).

We then check to see if the hashtable already contains a Key that is identical to that source IP address: if so, we increment the Value associated with that Key (source IP) by one; if not, then we add a new Key to the hashtable, where the Key is a new source IP address, and we set the Value of that new pairing to 1.

We examine every line from the log, either adding new source IP addresses to the hashtable or incrementing the count of dropped TCP connections to source IP addresses already seen in the log.

The output of the script (Get-DroppedSourceIP.ps1) will look something like this:

```
Name                Value
----                -----
10.6.2.44           54
192.168.1.88        42
10.17.4.9           28
```

This gives us a sorted table of source IP addresses and the total count of dropped inbound TCP connections per source IP. In real life, we might also get the count for dropped UDP connections or break it down by destination port number.

```
# Example: Get-DroppedSourceIP.ps1

$HashTable = @{}  #Empty hashtable

Switch -RegEx -File .\pfirewall.log
{
    "DROP\sTCP.+RECEIVE"
    {
       $SrcIP = ($_ -Split " ")[4]

       If ($HashTable.ContainsKey($SrcIP))
       { $HashTable.Item($SrcIP) = $HashTable.Item($SrcIP) + 1 }
       Else
       { $HashTable.Add($SrcIP, 1) }
    }
}


# Return each pairing of the hashtable separately for the sort:
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
$HashTable.GetEnumerator() | Sort-Object Value -Descending
```

The last line is weird. Each pairing inside the hashtable must be extracted separately so that Sort-Object can sort them by Value. If you piped the hashtable as a whole into Sort-Object, there would be only one thing piped in, namely, the hashtable itself. The GetEnumerator() method outputs each pairing in the hashtable separately.

Don't forget that *Switch* does not stop by default after the first match, so you can have multiple regex patterns to match against within the Switch flow control. For example, in the following, each line from the firewall log is compared against multiple regex patterns for the sake of doing a count of each type of line.

```
# Example: Processing_Text.ps1

$droptcp = $dropudp = $dropicmp = 0
$allowtcp = $allowudp = $allowicmp = 0

copy-item C:\Windows\system32\LogFiles\Firewall\pfirewall.log .

switch -regex -file .\pfirewall.log {
      'DROP TCP'   {$droptcp++}
      'DROP UDP'   {$dropupd++}
      'DROP ICMP'  {$dropicmp++}
      'ALLOW TCP'  {$allowtcp++}
      'ALLOW UDP'  {$allowudp++}
      'ALLOW ICMP' {$allowicmp++}
}

"FIREWALL LOG SUMMARY:"
"-------------------------------------------------------"
"Dropped: TCP=$droptcp, UDP=$dropudp, ICMP=$dropicmp"
"Allowed: TCP=$allowtcp, UDP=$allowudp, ICMP=$allowicmp"
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Functions

**Function = Block of code with a name.**

- Can take zero or more input parameters.
- Can return zero or more output objects.
- Can accept piped input, just like a cmdlet.
- Good way organize code into manageable units.

**Functions are kept in the function:\ drive.**

```
dir function:\
get-item function:\man | format-list scriptblock
```

SANS SEC505 | Securing Windows

# Functions

A function is a block of code in the function:\ drive which has been assigned a name. When you call or execute a function, you can optionally pass in parameters and/or have the function return a result, just like cmdlets. If written for this purpose, you can also pipe objects into functions, again just like cmdlets. Functions can be run from scripts or they can be dot-sourced into the current interactive session.

You can see all the functions available in your session by listing the contents of the function:\ drive. When you run a script, a new scope is created for that script and it can have its own functions inside it.

```
dir function:\
```

For any given function, like the *help* function, you can see the code assigned to it by selecting its scriptblock property with format-list.

```
get-item function:\man | format-list scriptblock
```

## How to Research and Write New Scripts with Functions

New coders often feel they must be able to sit down with a blank Notepad file in front of them and just start banging out code from scratch to solve whatever problem is at hand. They often feel that, if they can't do this, they don't *really* know how to program, they don't *really* understand the language they're using, and that they're somehow deficient as coders or IT experts. Nothing could be further from the truth. It is *impossible* to memorize all the properties and methods of all the .NET and COM classes you'll use to

Enclave Consulting LLC © 2017 - All Rights Reserved

solve problems at work, and if you did try to memorize it all, it would be a waste of time anyway because it's all going to change in a few years and you have other things to do at work (and at home) besides writing scripts!

When you are facing a new problem to solve through scripting, but you are unfamiliar with the products, technologies, or object models related to it, here is a rough recipe for approaching the task without feeling overwhelmed and without wasting a lot of time.

To research a new PowerShell script you need to write:

**1) Get the Microsoft Windows SDK and PowerShell Help File**

A "Software Developer's Kit (SDK)" includes help files, sample code, debugging tools and other resources for programmers. The Windows SDK includes the .NET Framework class library in a nice browseable/searchable help file. You should also get the SDKs for whatever other technologies or products you're interested in, e.g., WMI, Exchange Server, ADO.NET, etc. Start at http://msdn2.microsoft.com/downloads/ or use Google with the "site:microsoft.com" qualifier to only search Microsoft's site.

You might also prefer to have the PowerShell documentation in the form of a graphical help file (http://www.microsoft.com/technet/scriptcenter/topics/winpsh/pschm.mspx). If you are already familiar with VBScript, this help file includes a VBScript-to-PowerShell conversion guide that you'll find very useful.

**2) Set Up A Virtual Machine "Sandbox" That You Can Safely Destroy**

Don't test your scripts on production servers. Use VMware or Virtual PC to set up safe-to-destroy domain controllers, file servers, SQL Servers, Exchange Servers, etc. that you can use for debugging and trying out third-party products. If you trash your VM, shut it down, don't commit changes, and start it back up again!

**3) Break Up Your Desired Script Into Smaller Solvable Tasks**

At a high level of description, write down what you want your script to do, break those tasks into smaller subtasks, and repeat until you have a rough idea of all the little things your script will have to do which together get your big job done. You want to break it down into subtasks until you feel confident that you could write the code for each subtask by itself, knowing that you can later link them up together.

For example, you might want a script that you can schedule to run every night to remind users with old passwords to change their passwords, so this might be broken down into 1) connect to a domain controller, 2) get a list of global users, 3) get the age of each user's password, 4) test that password's age based on some definable limit, 5) get the e-mail address from the user account properties for any user with an old password, 6) send a reminder e-mail message to that user, and 7) write an event to the local Event Logs

Enclave Consulting LLC © 2017 - All Rights Reserved

recording the details of the user and message, such as whether the message was successfully sent.

**4) Find A Sample Script Similar To Your Small Task**

Search the Internet for other peoples' scripts that are related to whatever product or technology you need to write a script for. Use these as examples and to reveal clues about which objects, properties and methods you'll need to invoke and in what order. Use the "PowerShell" keyword in your Internet searches so the results will more likely to be relevant. C# examples are often useful too, if searching for PowerShell examples turns up nothing.

Don't forget to search forums or newsgroups related to your scripting language and the product(s) you need to manage, especially for tips, tricks and "gotchas" that are rarely published anywhere else. To find the Microsoft TechNet forum for PowerShell, the easiest thing is to just do a search on "technet powershell forum".

You should also get all of the sample scripts from the websites of all the book publishers who have books related to your language, product or technology that you need:

- www.wrox.com
- www.oreilly.com
- www.samspublishing.com
- www.manning.com

Dump all these scripts into a compressed NTFS folder for keyword searching later on.

Browse Microsoft's "Script Center" on MSDN for sample scripts and other resources at http://www.microsoft.com/technet/scriptcenter/hubs/msh.mspx

Microsoft MVPs and product aficionados often maintain websites and blogs with sample scripts, and these sites usually have hyperlinks to each other.

**5) Explore The Properties And Methods Of Your Misbehaving Objects**

In a separate shell or temporary script you'll often need to play around with the objects which are misbehaving. Use get-member to view their instance and static members, and format-list for property data; for example, if you were exploring get-service:

```
get-service | get-member
get-service | format-list *
```

Use your SDKs to read about and play with these properties and methods. Look at other sample scripts that invoke these properties and methods to see what they do with them. Search the Internet for the relevant class names and member names to try to find other documentation or sample scripts that will shed light on the problem.

Enclave Consulting LLC © 2017 - All Rights Reserved

### 6) Use Debugging Tricks To Shake Out The Problem And Reveal Clues

The $error array keeps a history of problems. Clear that array with "$error.clear()", rerun your troublesome function or code snippet, copy $error to a temporary variable ($temp = $error), then examine the entries in that temporary variable. You can then search error descriptions and error code numbers on the Internet.

```
$error
$temp = $error
$error.clear()
```

Sometimes it pays to deliberately cause errors in order to see the details of the messages produced. You can do this by calling methods or accessing properties that don't exist, passing in too many/few arguments, passing in the wrong types of arguments, mangling string arguments, etc.

You can also "sniff" a pipeline like you'd sniff a network using the tee-object cmdlet, which allows you to save copies of the objects coming down a pipeline to a file or variable without interrupting the flow of objects down the rest of the pipeline.

A good-old-fashioned debugging trick is to sprinkle commands at strategic locations in the script to write information about the data in variables at that point in time onto the screen, especially in local variables inside of functions.

If you wish to show detailed tracing information while the script run:

```
set-psdebug -trace 2
```

Do this in your script right before the problem area, which will show verbose debugging info, then execute "set-psdebug -off" right after the problem area to turn it off again.

If you wish to step through a problem area of your script one line at a time:

```
set-psdebug -step
```

Do this right before the problem area, then proceed by running each line separately. Execute "set-psdebug -off" again afterwards. You can also do this in the graphical PowerShell_ISE editor (see the Debug menu).

If you wish to open a new PowerShell session right in the middle of your running script so you can use that new interactive session to interrogate variables and experiment with different commands, execute "$host.EnterNestedPrompt()" to open the new session, then type "exit" to go back to your suspended script when you're done and continue running it.

Enclave Consulting LLC © 2017 - All Rights Reserved

If you think you have misspelled or undefined variables in your code causing problems, add the following line near the top of the script to raise an error if an undefined variable is encountered:

```
set-psdebug -strict
```

Finally, for even more detailed tracing of a single command, try using the trace-command cmdlet (see its help entry, we can't go into the details here).

**7) Post Your Question To A Forum**

If all else fails, you can always post a question to web-based forum:

- http://social.technet.microsoft.com/Forums/en-US/winserverpowershell/
- http://www.powershell.com
- http://www.scriptinganswers.com/forum/

**8) Put It On Ice For The Time Being…**

But when *everything* you try fails, it's time to give up. Seriously, there's no point in banging your head against a wall for days on end…however, don't delete any of your code, just set it aside. It's amazing how often you'll stumble across the solution to a hard problem a few days later. The problem will still be in the back of your mind, so when you later chance across the solution you'll recognize it!

```
$Luck = ($Opportunity + $Preparation) * $Effort
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Creating A New Function

```
function name { <your code here> }

function hi { "Hi!" }

function time { (get-date).ToLongTimeString() }

function edit-hostsfile
{
  notepad.exe $env:WINDIR\System32\Drivers\Etc\hosts
}
```

SANS SEC505 | Securing Windows

# Creating A New Function

A function does not have to be any more complicated than assigning a name to a block of code in curly braces. You simply put the keyword "function", followed by the name of the function, then the scriptblock in braces.

```
function hi { "Hi!" }

function time { (get-date).ToLongTimeString() }

function edit-hostsfile
{
     notepad.exe $env:WINDIR\System32\Drivers\Etc\hosts
}

function say-things
{
     "Hello World!"
     "7 + 3 = " + (7 + 3)
     "And now for a list of aliases!"
     get-alias
}
```

A function may have one or many lines inside it. The lines do not have to be indented, but it is easier to read if they are.

When a line in a function produces output, that output will be included in the output of the function as a whole. You do not need to use a *Return* keyword or similar to define

Enclave Consulting LLC © 2017 - All Rights Reserved

the output of the function. Nonetheless, we still talk about what the function "returns" when talking about the output of the function.

## Function Scope

Keep in mind that function parameters and other variables first-defined within the function scriptblock are local to that function. Local variables are not accessible to other functions or scripts unless they were called by the first function. If a function references a variable defined outside the function, that variable can be read or changed by the code inside the function.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Passing In Named Parameters To A Function

```
function New-User ($UserName, $Password)
{
  net.exe user $UserName "$Password" /add
}



new-user -username "Jill" -password "Sekrit"

new-user -u "Lori" -p "p@55vword"
```

SANS SEC505 | Securing Windows

# Passing In Named Parameters To A Function

A function definition can specify the number of arguments expected, their names, what their datatypes must be, as well as default values. A "parameter" is a variable that catches an "argument" passed into the function when the function is run

```
# Example: Named_Parameters.ps1

function New-User ($UserName, $Password)
{
   net.exe user $UserName "$Password" /add
}
```

The function above creates a new local user account. $UserName and $Password are variables which will catch any arguments passed into the function when the function is called. ("Calling" a function is the same as executing or running a function.) In the following, the function is called twice to create two new user accounts. If the function is not explicitly called, the function does not do anything. Notice that parameters may be abbreviated to a single letter as long as each parameter to a function begins with a different first letter, e.g., "-u" is for -UserName and "-p" is for -Password. Function names and parameter names are not case-sensitive.

```
new-user -username "Jill" -password "Sekrit"

new-user -u "Lori" -p "p@55vword"
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Passing In Positional Arguments To A Function

All the arguments passed into a function which do not go into named parameters will go into an automatic variable named "$args". Because it's an array, you can enumerate through it.

```
# Example: Positional_Args.ps1

function show-args { foreach ($x in $args) {$x} }
```

If you know that you'll pass in three arguments, you can access them by index number.

```
function show-3args {
      $computer, $user, $password = $args[0,1,2]
      "Count of arguments: " + $args.count
}


show-3args server47 admin sekrit
```

Function arguments are not passed in by specifying the name of the function followed by the comma-separated arguments in parentheses, e.g., this is wrong: "func(x,y,z)". Arguments are merely separated by spaces, not commas, and no parentheses are needed. The comma is actually an operator which binds elements together into an array, hence, if you do pass in "three" arguments separated by commas, you've actually passed in just one argument, an array of three elements, hence, $args[0] would reference an array of all the arguments and not the first argument only.

## Piping Objects Into A Function

When a function or script is run, you can pass parameters into it. But this isn't the only way to pass in data, you can also *pipe* objects into your own scripts and functions just like you can pipe into cmdlets!

When a function or script receives input in a pipeline, the objects piped into the function or script all go into an automatically-created array named "$input". Because it is an array, you can use it in a *ForEach* or *Switch* statement.

```
# Example: Piping_Input.ps1

function auf-deutsch
{
      foreach ($word in $input) { "Das " + $word + "en!"}
}

"Train" | auf-deutsch
```

The $input array is of type *System.Array+SZArrayEnumerator*, which means that the $input variable keeps track of the "current" item in the array so you can enumerate

Enclave Consulting LLC © 2017 - All Rights Reserved

through every item. A reference to "$input.current" returns the current item, while a call to "$input.movenext()" advances to the next item in the array, making *it* the current item. You can use either the *ForEach/Switch* or $input.movenext() approach when working with the $input array, your choice.

You can use the $input array in both functions and scripts, and if used in a script, you do not have to embed $input within another function. If the following line is placed in a script, data piped into the script will go into $input automatically.

```
# Following line is run in a script, not at the prompt:

while ($input.movenext()) {"Das " + $input.current + "en!"}
```

## Piping, Parameter and Positional Argument Binding

Hence, $input captures piped data and $args collects all the arguments that didn't go into named parameters. To see this in action, enter the following function and command into your shell and note the output (as shown below).

```
C:\> function thewayofdata ($p) {
>>    $p
>>    $args[0]
>>    $args[1]
>>    foreach ($x in $input) { $x }
>>}
>>

C:\> 1,2,3 | thewayofdata 4 5 -p 6

6
4
5
1
2
3

C:\>
```

Why did the numbers come out in this order? Named parameters ($p) are bound first, whatever parameters are remaining go into $args, and the data piped into $input are processed in the same order in which they were originally piped.

## Optimized Piping with the Process {...} Block

When objects are piped into a standard function, all the piped objects go into the $input array. But *when* do the piped objects go into the $input array and when is this array processed by the function? When a function is receiving objects in a pipeline, all of the objects to be piped into the function are gathered by PowerShell into one set and this set is assigned to $input in the function at a single moment in time. This means the pipeline

Enclave Consulting LLC © 2017 - All Rights Reserved

as a whole is held up until all the preceding commands before the receiving function have done 100% of their work, then all of the objects the function needs to process are placed into $input for the function to chew on. By analogy, in a factory assembly line for manufacturing toothpaste, it's as though the guy screwing caps on the bottles needs to get all 10,000 bottles before he can even start screwing on caps -- not very efficient.

Very similar to a standard function, though, is something called an "advanced function" (Get-Help about_Functions_Advanced). When objects are piped into an advanced function, the function can immediately process each object as it arrives from the pipeline even though more objects are on their way. Instead of waiting for all objects to finish coming through the pipeline, an advanced function is like the guy in the factory screwing on toothpaste bottle caps, but now he starts screwing on caps as soon the first bottle arrives at his station in real-time.

Because an advanced function runs as soon as the first object arrives from the pipeline, piped data does not go into the $input array or any other array. Instead, piped objects go into the default variable ( $_ ) inside a special block of code marked as "Process {...}". This is what is new, the Process block of code. The Process block is executed once for each object piped into the function, and that piped object goes into the $_ variable.

```
function auf-deutsch
{
     Process { "Das " + $_ + "en!" }
}
```

Advanced functions can also take named parameters with default values, emit objects, write to files, and do everything else standard functions can do. In fact, advanced functions can do quite a bit more than just process piped data, but a discussion of those details are beyond the scope of this course.

```
# Example: Translate.ps1

function Translate ([String] $Language = "German")
{
    Process
    {
        $word = $_

         switch ($Language)
         {
                 'German'  {"Das " + $word + "en!"    }
                 'French'  {"La "  + $word + "ette..."}
                 'Greek'   {"Oi "  + $word + "tai;"   }
          }
    }
}
```

Enclave Consulting LLC © 2017 - All Rights Reserved

If you expect a function to often be used in a pipeline to receive objects, consider remaking it into an advanced function instead. On the other hand, sometimes you need to have all the piped input batched up into a single complete array before processing, such as when you need to check the size of the input array first, in which it's best to leave it as standard function by not including the "Process {...}" block.

Incidentally, an advanced function can actually have three different special blocks: Begin, Process, and End.

```
function Pipe-BeginningEnding
{
    Begin   { "Run only once when the function is called" }
    Process { "For each piped object: " + $_ }
    End     { "Run only once when the function finishes" }
}
```

The "Begin {...}" block is executed only once per invocation of the function, even if 1000 objects will be piped in, and it will run before the Process block. The "End {...}" block is executed only once per invocation of the function, and this is after the Process block has finished handling any piped-in objects.

```
C:\> 1..3 | Pipe-BeginningEnding

Run only once when the function is called
For each piped object: 1
For each piped object: 2
For each piped object: 3
Run only once when the function finishes
```

To read more about it: Get-Help about_Functions_Advanced_Methods.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Constraining A Function's Parameter Types

```
function eat-lunch ([Int] $number, [String] $food)
{
   "Each day we eat $number $food pies!"
}

eat-lunch -number 3 -food pinkie
```

| Full Name | Short Name | Example |
|---|---|---|
| [System.String] | [String] | "Hi There" |
| [System.DateTime] | [DateTime] | "11/29/16 03:02:21 PM" |
| [System.Int32] | [Int] | 34 |

SANS SEC505 | Securing Windows

# Constraining A Function's Parameter Types

When a function is called and parameters are passed in, you can mandate the type of object each parameter must be. For example, you might want a function that must take two parameters, the first must be a string and the second must be an integer, and you want an error to be raised if the parameters are not type-correct (or cannot be converted on-the-fly to be of the correct type).

Just as you can cast a variable as a certain type of object, so you can cast a parameter name to be of certain type.

```
function eat-lunch ([Int] $number, [String] $food)
{
     "Each day we eat $number $food pies!"
}

eat-lunch -number 3 -food pinkie
```

The types you'll frequently want to mandate are listed in the following table. You can use either the full or short name as you prefer.

The last entry in the table, by the way, is for an array of objects (see the square brackets).

| Full Name | Short Name | Example |
|---|---|---|
| [System.String] | [String] | "Hello" |
| [System.DateTime] | [DateTime] | "2/19/16 03:02:01 PM" |
| [System.Boolean] | [Boolean] | $true |

Enclave Consulting LLC © 2017 - All Rights Reserved

| | | |
|---|---|---|
| [System.Int32] | [Int32] | 34 |
| [System.Int64] | [Long] | 9415869866 |
| [System.Decimal] | [Decimal] | 23.2927**d** |
| [System.Object[]] | [Object[]] | @("Hi",3,"Joe",4) |

Note that the "d" at the end of the decimal example above is not a typo.

You can also pass in a parameter to a function or script by reference with a cast to "[Ref]" (or to [*System.Management.Automation.PSReference*], which is the same thing).

A special type constraint is "[Switch]", which does not name a .NET class, it's for a switch parameter.

Enclave Consulting LLC © 2017 - All Rights Reserved

## Switch Parameters To Functions

**A switch parameter takes no arguments.**
**A switch can only be `$True` or `$False` (defaults to `$False`).**

```
function show-folder ([Switch] $list) {
     if ($list) {dir | format-list *}
     else {dir | format-table fullname,length}
}

show-folder                 #No switch, so $list = $False
show-folder -list           #Switch used, so $list = $True
```

SANS SEC505 | Securing Windows

# Switch Parameters To Functions

A switch-type parameter takes no argument(s) to define its contents. A switch parameter is either passed into a function or it isn't, hence, it can only be interpreted as $true (it exists as a parameter) or $false (was not passed in as a parameter). You can constrain the type of a switch parameter with a cast to [Switch]. When the function is called, the switch parameter must begin with "-", but the rest of the name of the parameter can be abbreviated, e.g., "-list" or "-li" or "-l", as long as PowerShell can figure it out.

```
# Example: Parameter_Switch.ps1

function show-folder ([Switch] $list) {
     if ($list) {dir | format-list *}
     else {dir | format-table fullname,length -autosize}
}

# Call function with or without the switch parameter:
show-folder -list

show-folder -l

show-folder
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Assigning Default Values To Function Parameters

```
function disable-admin
{
    Param ( $Password = "SEC505Gr8#4TV!" )
    net.exe user Administrator "$Password"
    net.exe user Administrator /active:no
}


disable-admin

disable-admin -password "0v3rr1d3n!"
```

SANS SEC505 | Securing Windows

# Assigning Default Values To Function Parameters

If you wish, you can assign default values to the parameters of a function. If no parameters or arguments are passed into the function, the default values win. If some items are passed in, those will be bound to the function's parameters and the rest will go to the defaults. Of the function's input parameters, it's permissible for some to have defaults defined and to leave other parameters without defaults, i.e., to require those values to be passed in. You can combine defaults with type constraints.

The following sample code shows how to assign a default value to a parameter and also how to use the "Param" keyword for defining the parameters to a function.

```
# Example: Default_Value.ps1

function disable-admin
{
    Param ($Password = "SEC505Gr8#4TV!")

    net.exe user Administrator "$Password"
    net.exe user Administrator /active:no
}


disable-admin

disable-admin -Password "0v3rr1d3n!"
```

Enclave Consulting LLC © 2017 - All Rights Reserved

If an argument is not given to the -Password parameter when the above function is called, the function will use the default password defined inside the function body.

Notice that the name of the function (disable-admin) is not followed by parentheses as with most of the other function examples so far in the manual. An alternative method of defining the parameters to a function is to include the keyword "Param" as the first executable line inside the function body, followed by parentheses with the desired parameters inside the parentheses. The syntax for parameters, constraining input types, switches, and assigning default values is exactly the same whether the parentheses immediately follow the name of the function (as shown in most examples so far) or follow the "Param" keyword inside the body of the function.

When you are reading the code of other authors on the Internet, sometimes parameters will follow the function name, sometimes they will be defined using the "Param" keyword. It is your choice for your own scripts.

## Returning Results From A Function

PowerShell functions don't really "return" anything, they simply execute lines of code in a scriptblock. Now, if those lines of code create output, you can capture that output into an array using the assignment operator ("=") just like when you capture the output of a cmdlet. But if you don't capture the output of the function, the output will be displayed in the shell just as though you had executed each line separately outside of the function by hand.

```
function dothings { dir c:\ ; "Hello" ; get-process }

$output = dothings
```

But PowerShell does have a *return* keyword. Inside a function, the *return* keyword will cause whatever command immediately follows it to be executed and then the flow of execution will break out of the function without running the rest of the lines of code inside it. If you simply wish to stop executing the rest of the commands in a function and go back to the context which called the function, just use *return* with nothing after it.

However, use of the *return* keyword is optional. Whatever output in a function is not consumed by something else will automatically be output by the function, with or without the *return* keyword anywhere. Just emit some data and that's what the function will emit. Hence, the *return* keyword is really only for exiting a function before getting to the end of its scriptblock.

```
# Example: Returning_Results.ps1

function times-ten ($number) {
      $x = $number * 10
      return $x
      "You'll never get here because of the return."
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
}


function times-eleven ($number) {
      $number * 11
      return
      "You'll never get here because of the return."
}
```

If commands within a function's scriptblock are producing output you don't want, you can cast that output to [Void] to annihilate it. You can also redirect the output to $null which does the same thing. "$null" is a built-in variable that never has any content.

```
function shout { [Void] "Hey!" ; "There!" }

function shout { "Hey!" > $null ; "There!" }
```

Enclave Consulting LLC © 2017 - All Rights Reserved

## Passing Arguments Into Scripts

```
# ise .\Examples\PingWrapper.ps1
# To pass arguments into the named
# parameters of a script, use Param(...)

Param ($computer = "localhost")

function pingwrapper ($ip) { ping.exe $ip }

pingwrapper -ip $computer
```

SANS SEC505 | Securing Windows

# Passing Arguments Into Scripts

All the arguments passed into a script go into an automatically-created array named "$args" if the arguments are not passed into named parameters. While there is nothing inherently wrong with processing $args to extract positional arguments, it's not the preferred PowerShell way.

Instead, a script can specify the number of arguments expected, their names, what their datatypes must be, as well as their default values. In short, instead of passing in simple arguments, these arguments become full-fledged named parameters in the same way that cmdlets and functions can take named parameters.

To pass in named parameters into a script when that script is executed, the first executable line in the script must begin with the "Param" keyword followed by parentheses containing the named parameters. Non-executable lines are either blank or commented out. The syntax for the named parameters in the parentheses is the same as when implementing a function, they just happen to be after the "Param" keyword.

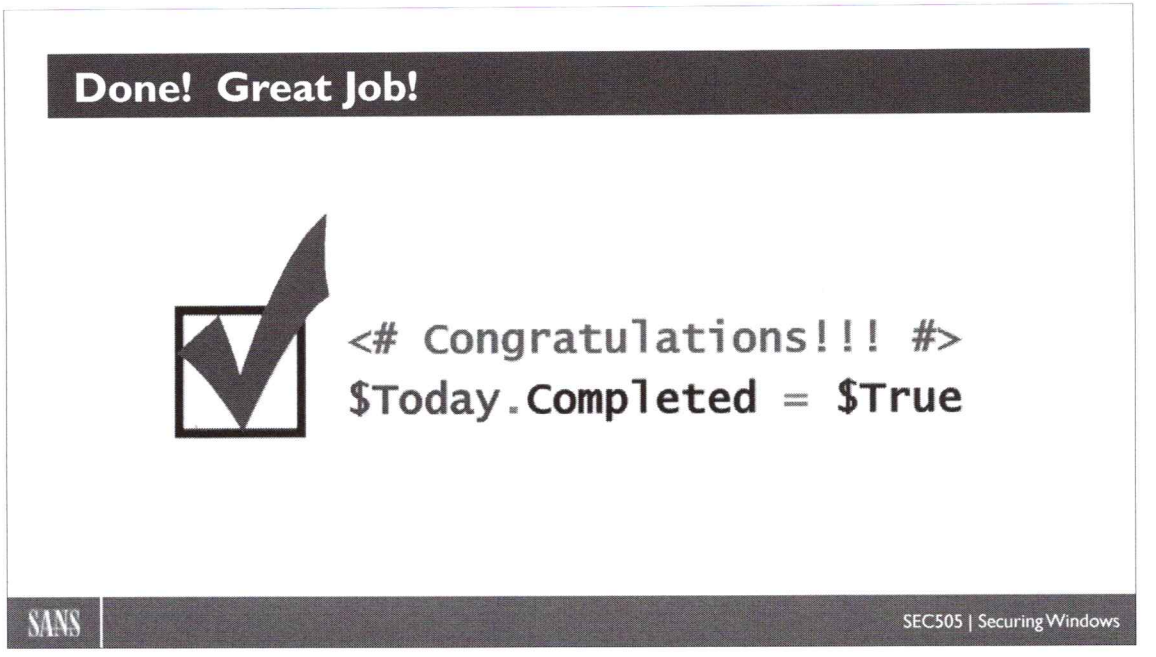
```
# Example: PingWrapper.ps1

param ($computer = "localhost")

function pingwrapper ($ip) { ping.exe $ip }

pingwrapper -ip $computer
```

Enclave Consulting LLC © 2017 - All Rights Reserved




# Congratulations!

You've finished the course! ☺

We hope you found the course useful and enjoyable. Please fill out the evaluation form, it helps us to make the course even better.

**Thank You!**

Enclave Consulting LLC © 2017 - All Rights Reserved

# Appendix A: Becoming A Domain Controller

Follow these steps to install Active Directory on your test virtual machine to make it a domain controller.

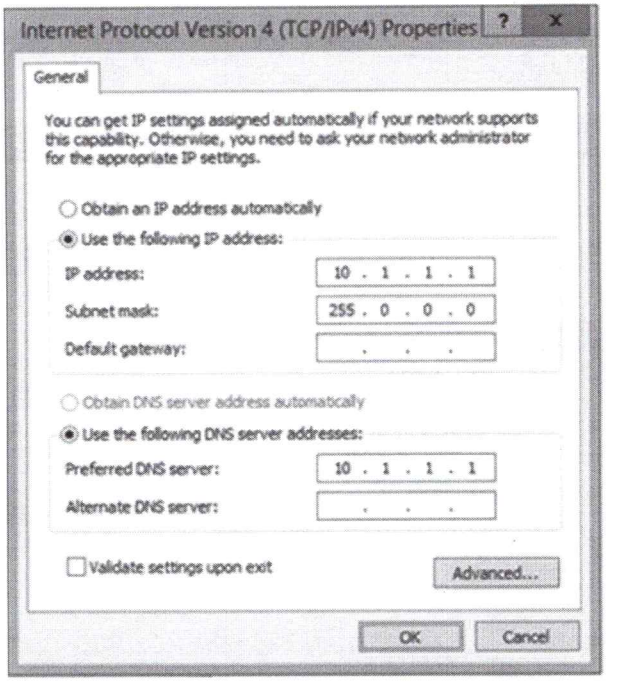
Windows Server is installed as a non-domain controller first, then Active Directory Services are installed when the machine is promoted to become a domain controller (DC). Active Directory (AD) can later be uninstalled to make the system a non-domain controller again. Promotion and demotion do not require reinstallation of the operating system.

> **Note:** If you are not installing in a virtual machine (VM) and your computer is not connected to a live network, you may need to install the "Microsoft Loopback Adapter" if you experience networking problems. Again, you do not need to do this if you are using VM software, such as Hyper-V or VMware Player.
>
> Double-click the "Device Manager" applet in Control Panel (or open it from within the System applet) > right-click your server at the top > Add Legacy Hardware > Next > select "Install the hardware that I manually select from a list (Advanced)" > select "Network Adapters" > Next > choose "Microsoft" as the manufacturer > choose "Microsoft Loopback Adapter" > Next > Next > Finish.

## Static IP Address (10.1.1.1) and Client to Your Own DNS (10.1.1.1)

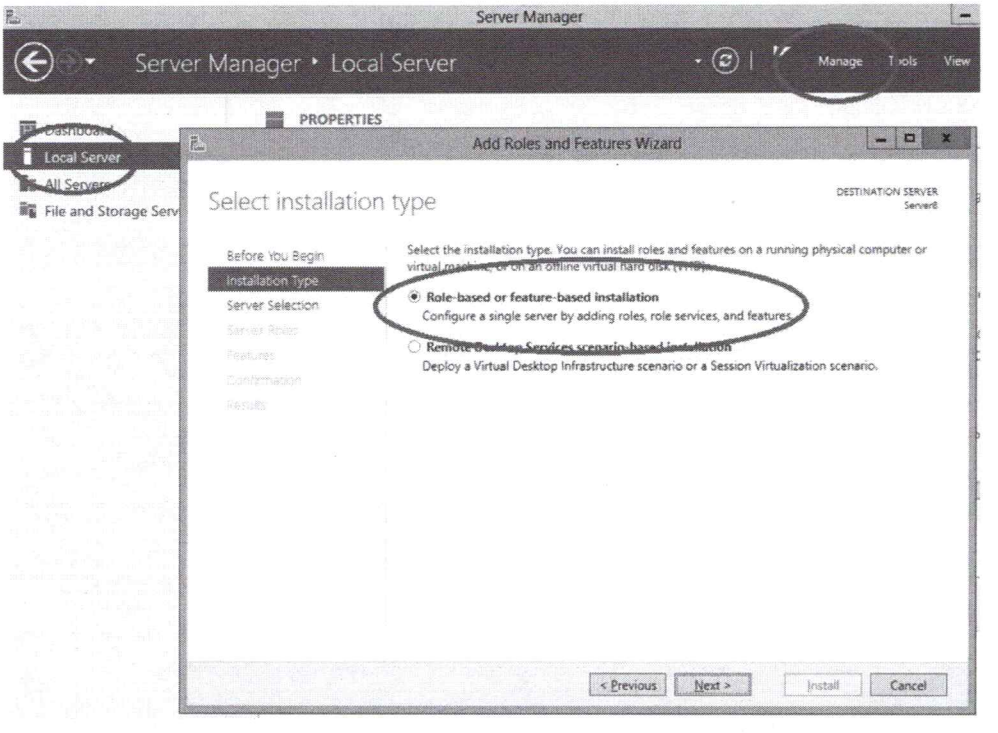
In your test VM, go to Control Panel > Network and Sharing Center > Change adapter settings > right-click your network interface > Properties > select Internet Protocol Version 4 (TCP/IPv4) > Properties > configure that adapter with a static IP address (10.1.1.1) and set the DNS server for that adapter to be your own IP address (10.1.1.1).

Enclave Consulting LLC © 2017 - All Rights Reserved



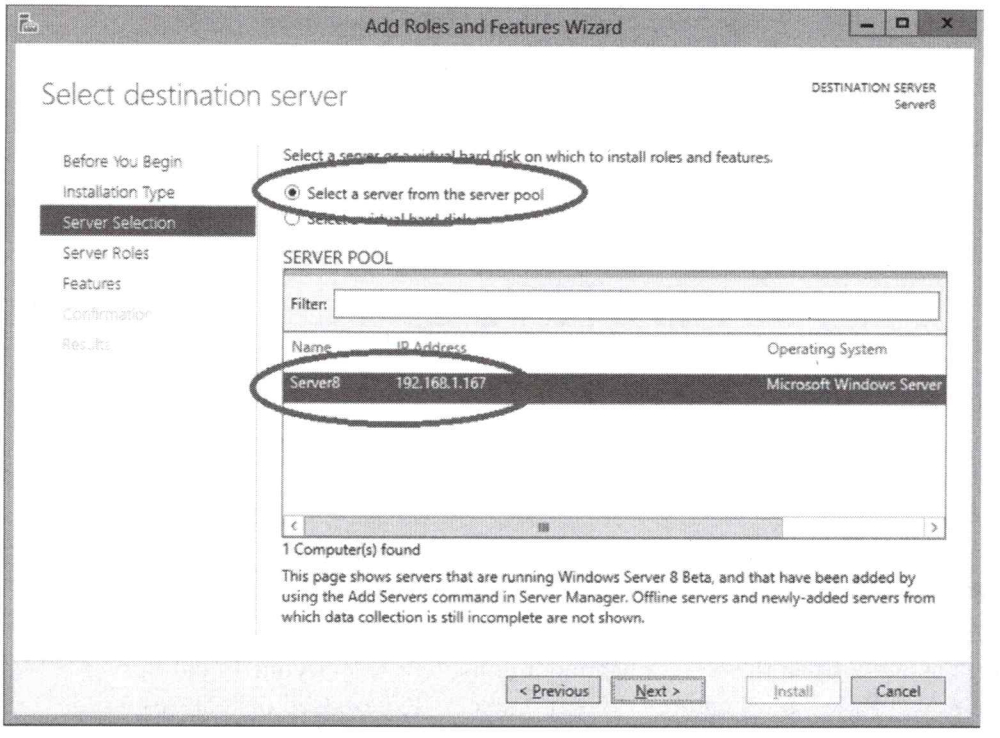

## Server Manager

Assuming you have a static IP address and your primary DNS server is your static IP address as well, open the Server Manager console > select your Local Server > Manage menu > Add Roles and Features > Next > select "Role-based or feature-based installation" > Next.



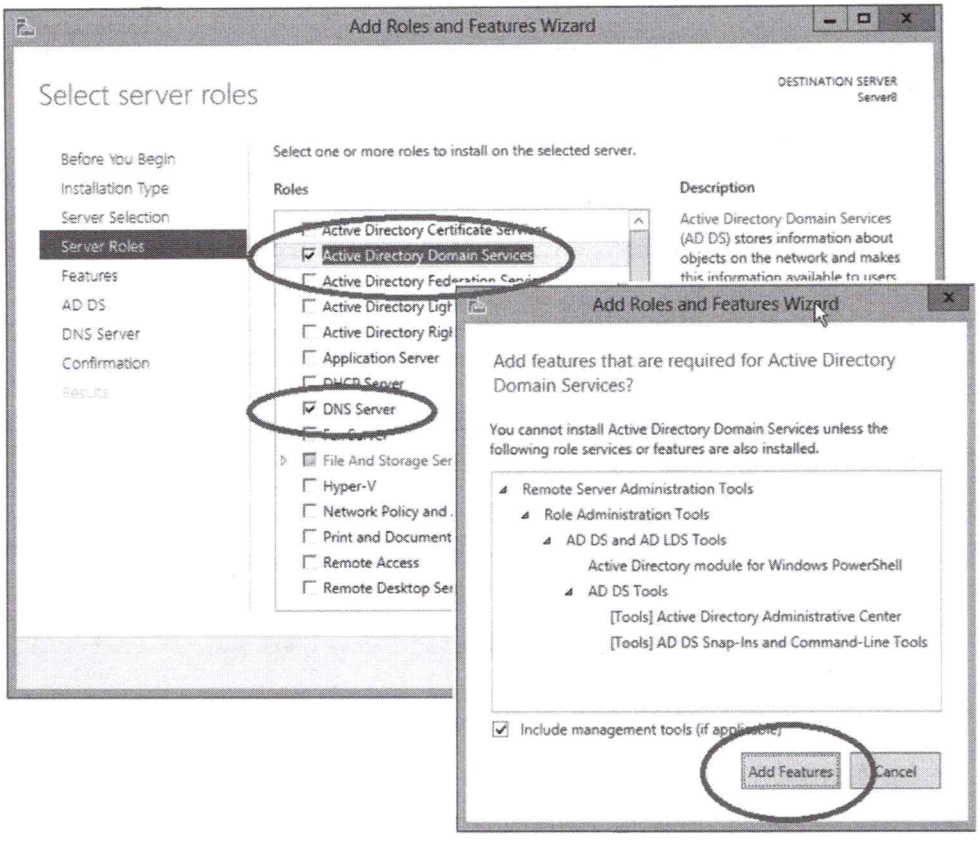

Enclave Consulting LLC © 2017 - All Rights Reserved

Next > choose "Select a server from the server pool" and make sure your own local server is highlighted > Next.

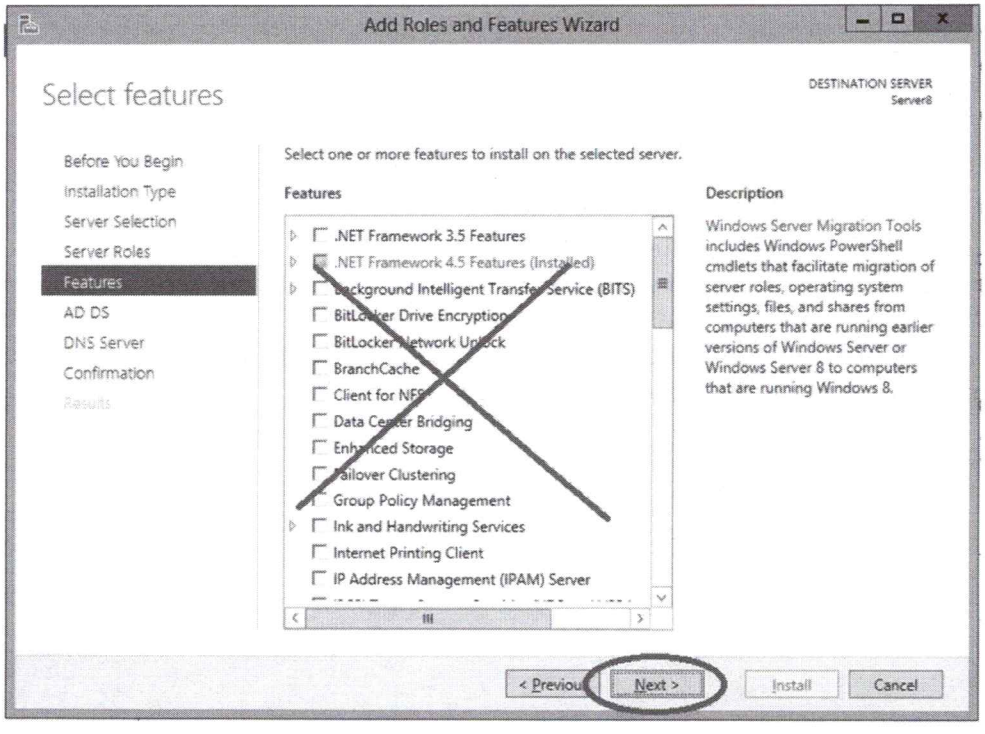



Check the box for "Active Directory Domain Services" > click the "Add Features" button > check the box for "DNS Server" > click "Add Features" button > Next.

Enclave Consulting LLC © 2017 - All Rights Reserved




Click Next again, there are no extra features to be installed now.

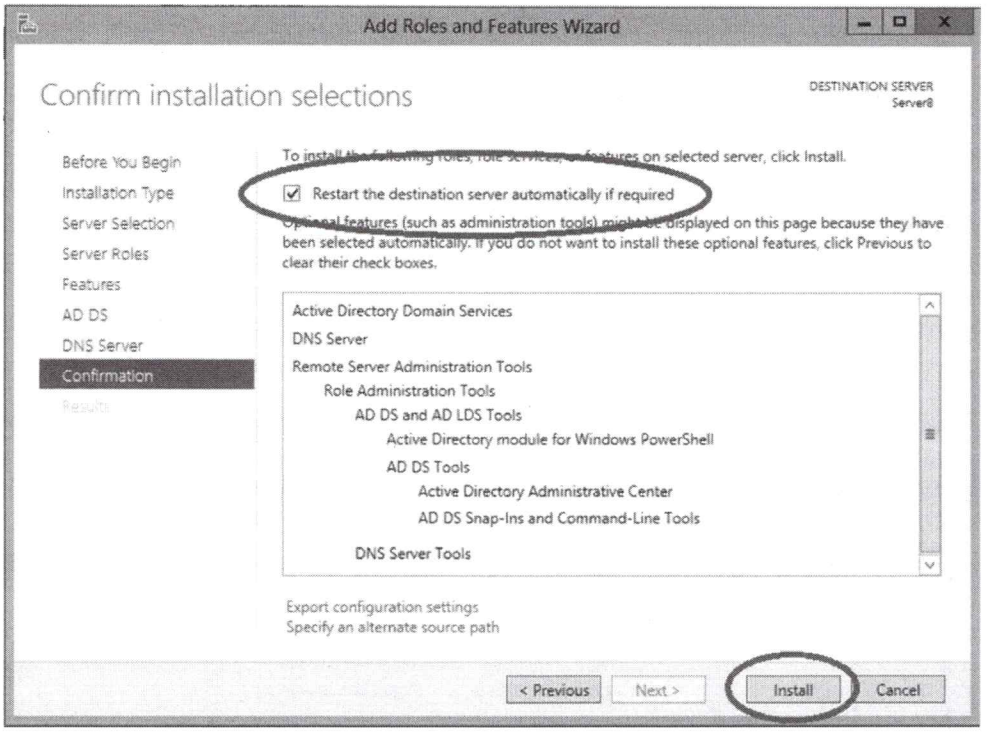



Enclave Consulting LLC © 2017 - All Rights Reserved

Next > Next > Next > check the box to "Restart the destination server automatically if required" > Install.

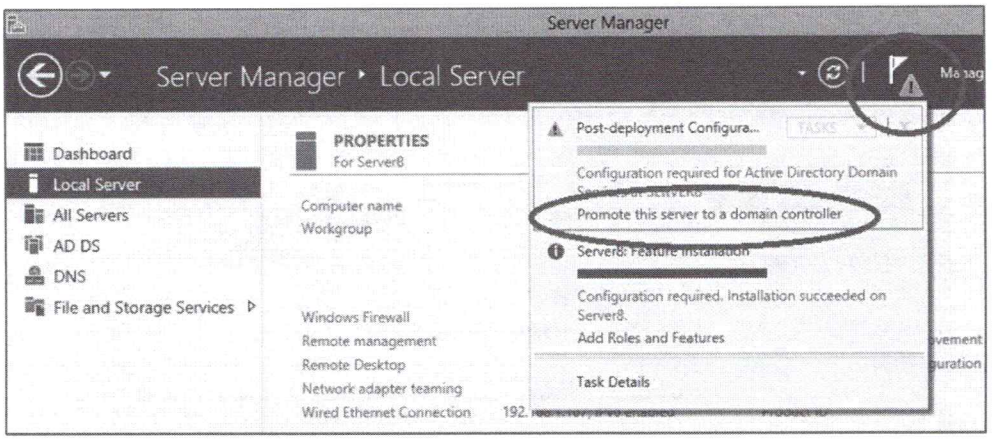
> **Important:** If you are prompted to provide the path to the installation media, and if you have mounted the DVD or ISO file on drive letter "D:", then click the link at the bottom to provide an alternate path of "**d:\sources\sxs**".

Now, even though you can click Close, do not, just wait until the progress bar shows completion (tell the instructor if you get an error message). Then Close.

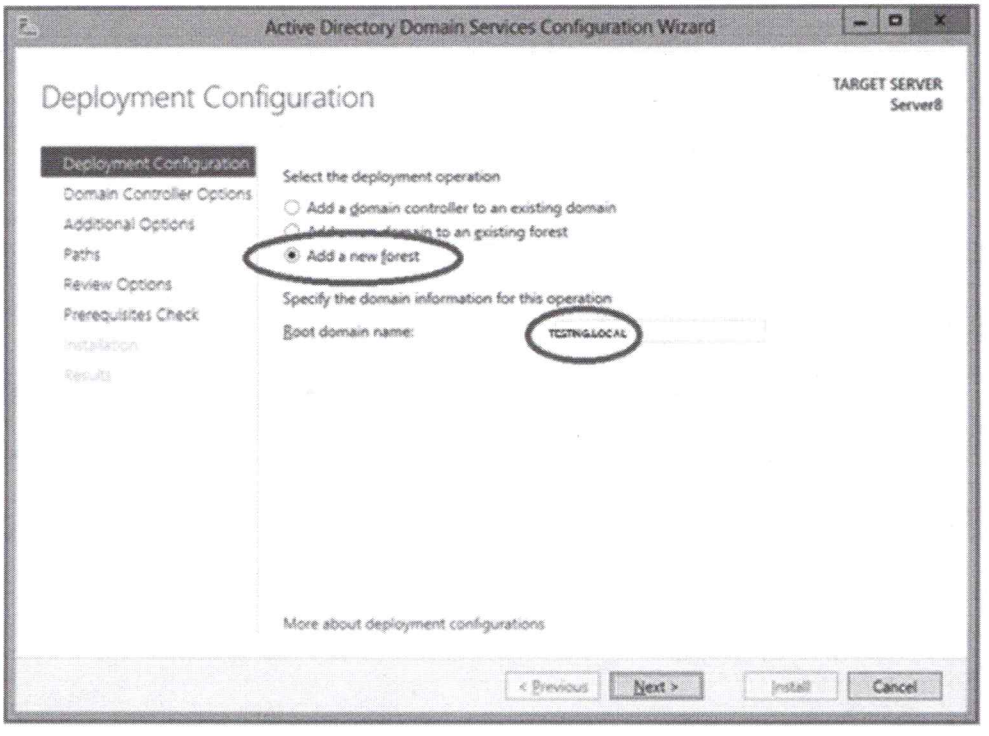



Go back to Server Manager, click the triangle notification near the flag at the top to see the progress of the installation of the role. Every minute or so, click the circular double-arrow refresh button and pull down the triangular alert menu again. Eventually, when it finishes, you will see and then click on "Promote this server to a domain controller". (There's no need to run DCPROMO.EXE anymore.)

Enclave Consulting LLC © 2017 - All Rights Reserved

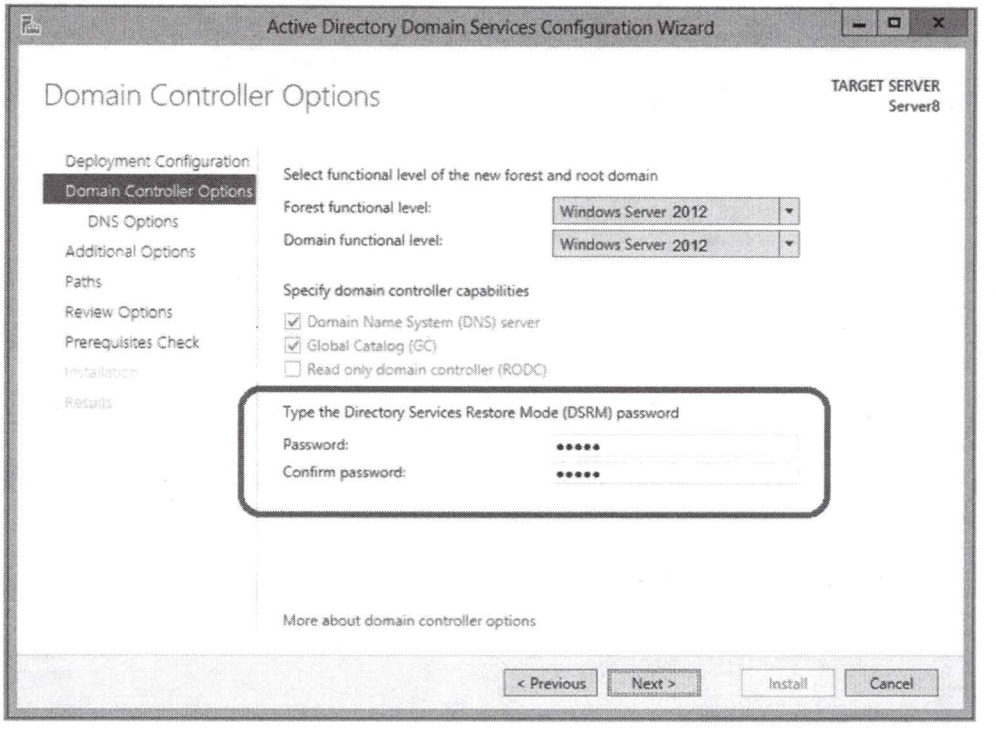



Select "Add a new forest" > enter "testing.local" as the root domain name (or anything you wish) > Next.



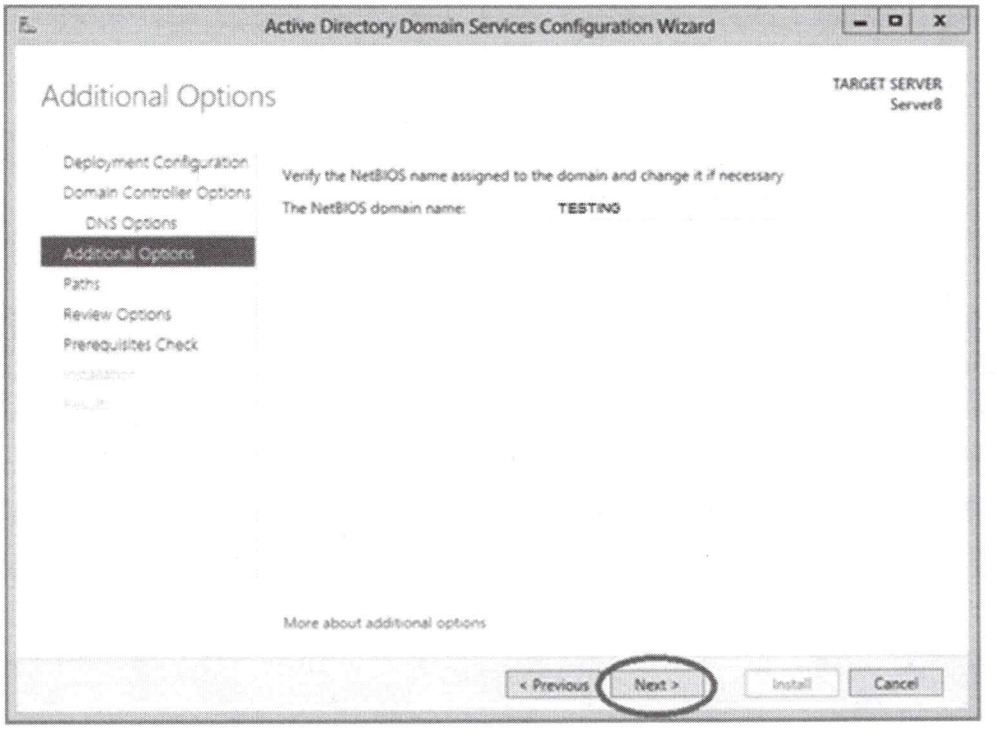

Select forest and domain functional levels of "Windows Server 2016". Enter a password of "Sans*8" for the DSRM password (or anything you'll remember) > Next.

Enclave Consulting LLC © 2017 - All Rights Reserved



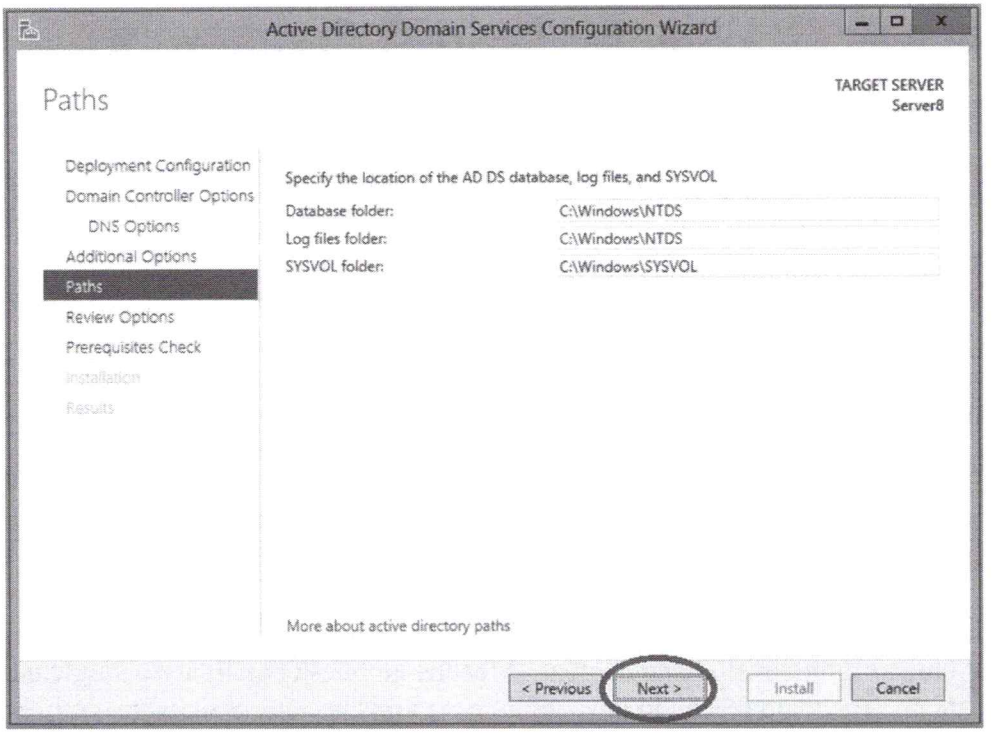

If you get an error concerning the DNS configuration, ignore it > Next.

Leave the NetBIOS name to the default > Next.




Enclave Consulting LLC © 2017 - All Rights Reserved

Leave the folder locations to their defaults > Next.




Next > Install. Ignore any error messages concerning DNS, cryptography, or anything else which does not block the installation process (if you do get a blocking error, ask the instructor). The server will restart automatically after the install is finished.

Log onto your new domain controller with the same password you had before > launch Server Manager (if it does not run automatically) > Tools menu > Active Directory Users and Computers.

Go back to Server Manager > Tools menu > Active Directory Administrative Center > go to the upper left-hand corner of ADAC and look for two tabs > select the tab on the right to enable "Tree View" of your AD domain.

Enclave Consulting LLC © 2017 - All Rights Reserved

# Appendix B: What Is The .NET Framework?

"**.NET**" is a catch-all term for a set of technologies for writing, compiling, managing, and running applications in a special way on Windows and other operating systems. These applications can be regular command shell executables, Windows GUI applications, web applications, or other network services. The "**.NET Framework**" is this programming and execution environment and everything that makes it work. If an application runs inside of the .NET Framework, it is said to be "**managed code**", i.e., managed by the .NET Framework.

A "**.NET application**" is a set of one or more assemblies that together implement that application. A single "**assembly**" is one *or more* files with either the EXE or DLL file name extension; hence, a single assembly might span multiple EXE/DLL files, though most assemblies are just single files. (Note that not all EXE and DLL files are .NET assemblies, they might also just be regular Windows API binaries.)

The interior of an assembly can be subdivided into zero or more "**modules**", which are named units of functionality and content. Most assemblies contain just a single module inside them. The main reason to have modules at all is because a single assembly may span multiple EXE/DLL files, and the main reason an assembly may span multiple files is because the source code behind each file might be written in a different language, such as VB.NET or C#.

An assembly implements one or more classes (or "types", to be more exact). A "**class**" defines and implements a set of properties and methods for the objects created from that class, i.e., instantiated from that class. An "**object**" is a named unit of properties (data) and methods (code) in memory, and by using these properties and methods an application is made to do useful things.

An assembly is described by the manifest inside it. The "**assembly manifest**" is information about the assembly, its name, version, dependency upon other assemblies, authenticating public key, culture/locale, and the classes the assembly implements. There is also a second, optional manifest, called the "**application manifest**", which is typically stored as an external XML file with the ".manifest" file name extension. This XML file contains more metadata about an assembly, mainly for use by the operating system before launching the .NET application loading the assembly. Hence, when a .NET application is said to be "**self-describing**", it means that there is the assembly manifest and possibly an XML application manifest for every assembly that comprises the application.

EXE and DLL assemblies are compiled into **Microsoft Intermediate Language** (**MSIL**). MSIL is a programming-language-neutral and CPU-instruction-set-neutral representation of data and executable code, hence, C# and VB.NET both compile to the same type of MSIL. When the modules of an application are copied to another computer

Enclave Consulting LLC © 2017 - All Rights Reserved

and run, the MSIL of these EXEs and DLLs are compiled again into the native machine language of that computer's CPUs, and these native-language compiled versions of the modules are run whenever the application is run. Modules in an assembly are not recompiled unless they are changed somehow.

Hence, .NET applications do not run in any kind of ".NET Virtual Machine", they run as native machine language instructions on the CPUs just like any other non-.NET executable. Managed code is compiled twice: first into MSIL, then again into the native machine language of whatever computer the application is running on. After being thus twice-compiled, a .NET application EXE or DLL file is a regular Portable Executable (PE) binary which can be parsed, loaded and executed by the Windows operating system loader just like any other PE binary program. Think of the .NET Framework as an automated source code auditor (actually, an automated MSIL auditor) which runs automatically the moment before the source code is compiled, and .NET will block this compile process if .NET detects anything dangerous or hinky.

What compiles modules from MSIL into native machine language? Modules in MSIL are compiled on-the-fly as needed by the **Common Language Runtime** (**CLR**). The CLR is one part of the .NET Framework. The CLR compiles MSIL modules into native code, handles memory management, enforces security, keeps track of module versions, and provides hooks for debuggers and performance monitoring. Once an application is compiled into native machine language, the CLR exists inside that application process as a set of DLLs, hence, the CLR is not external to a running process, it is a part of the process itself. If ten .NET applications are concurrently running, then there are ten living CLRs, one in each application process. Such processes, which have the CLR in them, are called "**runtime hosts**".

While running, a runtime host process will load assemblies into an "**application domain**", which is an in-memory container for assemblies and configuration settings. So, while an assembly is a file on the hard drive, an application domain is an in-memory runtime construct. A host process must have one application domain, but it may create many application domains in order to provide protection and management separation for sets of assemblies and configuration settings.

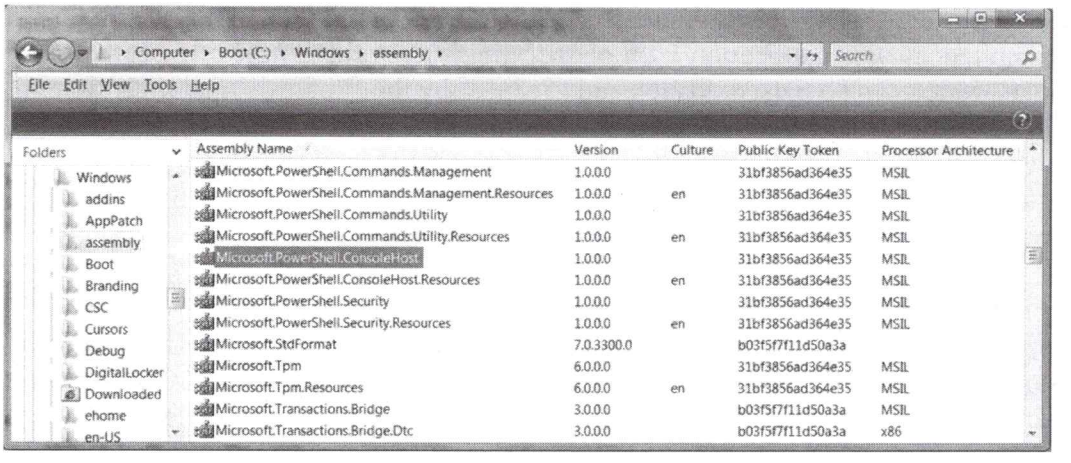
Another important part of the .NET Framework is the **.NET class library**. The class library is a huge collection of classes with many properties and methods. Even a simple type of data, like a string of characters or an integer number, is actually an object with properties and methods, hence, is an object derived from a class in the .NET class library. All .NET applications use the .NET class library. Almost all differences between languages, like C# and VB.NET, are removed when they are compiled into MSIL because MSIL uses the .NET class library to represent data types.

In the .NET class library you'll find classes related to file management, databases, GUI applications, ASP.NET web applications, XML, Windows services, the registry, and very many other technologies. Eventually, when the .NET class library is "complete", every Microsoft software product will be implemented and/or manageable through classes in

Enclave Consulting LLC © 2017 - All Rights Reserved

the library. As new versions of the .NET Framework come out, the class library gets bigger.

Physically, the .NET Framework is installed in a version folder underneath %SystemRoot%\Microsoft.NET\Framework\v.*VersionNumber*. However, there is also the **Global Assembly Cache** (**GAC**) under %SystemRoot%\Assembly, which stores assemblies that are available to all .NET applications.




When you view the GAC folder in Windows Explorer, the display window changes to show version information about each assembly. PowerShell has its own assemblies in the GAC too.

## The .NET Class Library

PowerShell is itself a .NET executable program and through it you have access to the .NET class library. To browse the .NET class library on Microsoft's web site, go to http://msdn2.microsoft.com/library/, click on ".NET Development" and go the "Class Library" section. The classes are organized into "namespaces"; for example, if you browse to the *System.Text.RegularExpressions* namespace, you'll find the *RegEx* class underneath it, along with documentation for all the properties and methods of that class.

Most of the .NET classes (that is to say, most of the assemblies) you'll need are loaded automatically when PowerShell starts, but you can load more assemblies as needed to get access to the classes they implement.

## Browse Assemblies, Classes, and Class Members

To see a list of the assemblies currently loaded into PowerShell:

```
# Example: Creating_NET_Objects.ps1

[System.AppDomain]::CurrentDomain.GetAssemblies() |
format-list FullName,Location
```

Enclave Consulting LLC © 2017 - All Rights Reserved

In .NET, an "application domain" is similar to a scope in PowerShell: it's a context in which assemblies are loaded, protected, run, and unloaded. The "default application domain" is the first domain created for the initial process that represents the application running, which in this case is `powershell.exe` itself. The default application domain ceases to exist when the initial process is terminated, but a process could spawn off new application domains as desired just as `powershell.exe` can launch other programs and scripts. The "current domain" is the application domain of the currently-running thread, which, for PowerShell in this example, is in the default/initial application domain.

To see a list of all the classes from all the assemblies currently loaded into PowerShell:

```
[System.AppDomain]::CurrentDomain.GetAssemblies() |
foreach-object { $_.GetExportedTypes() } |
format-list fullname,assembly
```

To see the assembly which implements a *currently-loaded* class (in square brackets):

```
[System.String].Assembly | format-list

[System.Management.Automation.ErrorRecord].Assembly | format-list
```

To see all the properties and methods of a class (we've seen this before):

```
[System.Net.WebClient] | get-member | format-list
```

To see all the *static* properties and methods of a class (we've seen this before):

```
[System.String] | get-member -static | format-list
```

But besides the get-member cmdlet, you can also query a class with its own methods:

```
[System.String].GetMembers()

[System.DateTime].GetMembers()

[System.Net.WebClient].GetMembers()
```

## Loading Assemblies And Classes Into PowerShell

To load a class you want, you have to know the name of the assembly that provides that class. If that assembly is in the Global Assembly Cache (GAC), you don't have to know the full path to the file, just the name of the class you want (the .NET Framework will handle the rest for you). The name used to load the assembly can either be its partial name or its full name (partial names are also called "short" or "simple" names, while full names are also called "strong" names).

The short name for an assembly in the GAC might be:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
System.Windows.Forms
```

While the full/strong name for that same assembly is:

```
System.Windows.Forms, Version=2.0.0.0,
Culture=Neutral, PublicKeyToken= b77a5c561934e089
```

To load a .NET class into PowerShell from its assembly stored in the Global Assembly Cache (GAC) using the short/simple/partial name of that class:

```
[System.Reflection.Assembly]::LoadWithPartialName("System.Windows
.Forms")
```

The *System.Reflection.** classes all relate to the use of metadata to query and manipulate the assemblies, modules, classes, properties and methods of .NET applications. Recall that this metadata is part of each of the modules (EXE or DLL) which comprise a .NET application.

To load a .NET assembly into PowerShell using the full/strong name of the assembly:

```
 [System.Reflection.Assembly]::Load("System.Windows.Forms,
Version=2.0.0.0, Culture=Neutral, PublicKeyToken=
b77a5c561934e089")
```

On the other hand, if the assembly is not in the GAC, you'll have to load the file with its full hard drive path.

To load a .NET assembly into PowerShell using the full path to the assembly file:

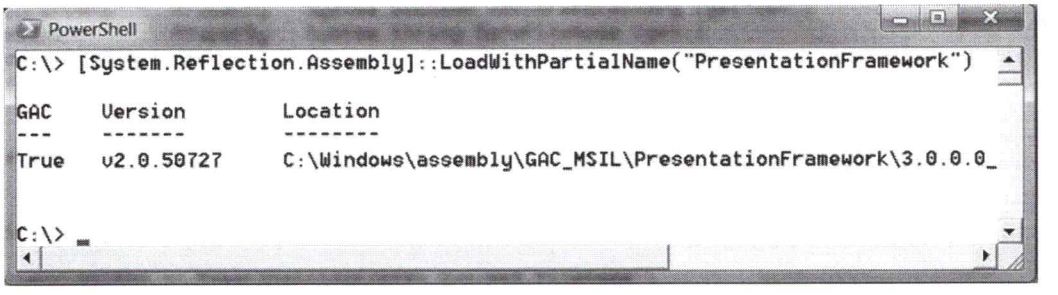
```
[System.Reflection.Assembly]::LoadFrom('C:\folder\module.dll')
```

A problem you'll sometimes face is when you know the name of the class you want to use, but you don't know the name of the assembly for it. If the code sample you're looking at doesn't reveal the assembly name, find the class name on Microsoft's MSDN web site (http://msdn2.microsoft.com/library/) and it will give the name of the assembly to load. (If you try "[ClassName].Assembly" it won't work because the assembly for that class hasn't been loaded yet…chicken-and-egg problem.)

Another problem is when you want to use a class that only exists in a particular version of the .NET Framework, and you don't happen to have that version installed. For example, there is a class named *Microsoft.Win32.OpenFileDialog* provided by the presentationframework.dll assembly, but this assembly and class only exists if you have the .NET Framework 3.0 or later.

Enclave Consulting LLC © 2017 - All Rights Reserved

When you try to load an assembly, how do you know if it worked? If it fails to load, *nothing* will be output (and even $? will still be true). If the assembly loads successfully, the command will output information about the assembly, such as whether it is currently in the Global Assembly Cache (GAC) and its version number. The following screenshot shows a successfully loaded assembly.




You will also know when the assembly load has failed because your attempt to create an instance of a class from that assembly will cause an error message: "Cannot find type… make sure the assembly containing this type is loaded." You'll have to test for this in your code when doing error handling.

## Creating .NET Objects

PowerShell creates, uses and destroys .NET objects almost every moment you're doing something with it. Many types of .NET objects will be created for you on-the-fly without any special syntax or creation formalities. Just simply use the built-in cmdlets and create strings/integers/arrays/etc as normal.

## You Don't Have To Know How To Do This To Use PowerShell!

But the entire .NET class library is available *if* you want to use it. However, you do not have to master the entire class library or even look at it to productively use PowerShell. Just because you *can* create exotic .NET objects doesn't mean you *must* create them to get your work done. New programmers often shriek in Kafkaesque horror when they see the .NET class library for the first time. But don't worry, the best way to learn is to look at other peoples' code after Googling the problem you're trying to solve. You can ponder the .NET class library and your metamorphosis as a coder *later*…

To create an object whose parent assembly is already loaded:

```
# Example: Creating_NET_Objects.ps1

$object = new-object System.Int32
$object = new-object System.DateTime
$object = new-object System.Net.WebClient
$object = new-object System.Net.Mail.SmtpClient
```

But if the assembly containing the class hasn't been loaded, you'll have to load it first:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
[System.Reflection.Assembly]::LoadWithPartialName("PresentationFr
amework")

$object = new-object Microsoft.Win32.OpenFileDialog

$object.ShowDialog()

$object.FileName
```

## Constructor Arguments (-ArgumentList)

Some objects must be created with more information than just the name of the .NET class. These additional arguments are required by the constructor methods of the object. When you pass in this additional information, put all the arguments in a single comma-delimited array (or no comma if there's only one argument). And, since we're passing in multiple parameters to new-object now, we should spell the parameters out more explicitly.

```
$s = new-object -type System.String -argumentlist "Hello"
```

To get the full documentation on what constructor arguments can or must be passed in, consult the .NET class library at http://msdn2.microsoft.com/library/.

To get *hints* about the possible constructor arguments for a type of object, such as for a *System.String* object:

```
[System.String].GetConstructors() |
foreach-object { $_.getparameters() } |
select-object  name,member |
format-table -autosize
```

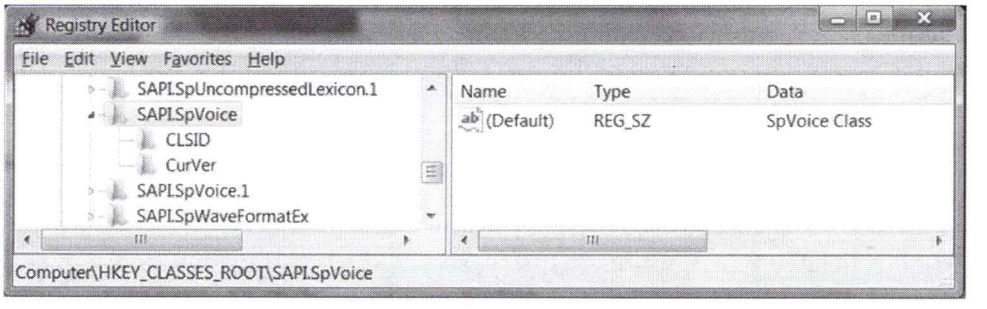
Enclave Consulting LLC © 2017 - All Rights Reserved

# Appendix C: Creating COM Objects

PowerShell is a .NET application. This means it can access .NET classes fairly well, but going through the .NET Framework to access COM objects isn't always perfect. Component Object Model (COM) has been the dominant standard since 1993 for the creation of language-independent objects on Windows that have well-defined and discoverable properties and methods. The mainstay of VBScript/JScript scripting, for example, is the creation and use of COM objects. PowerShell can access COM objects too, just not as well as VBScript/JScript (for example, PowerShell 1.0 lacks a "GetObject()" method like in VBScript).

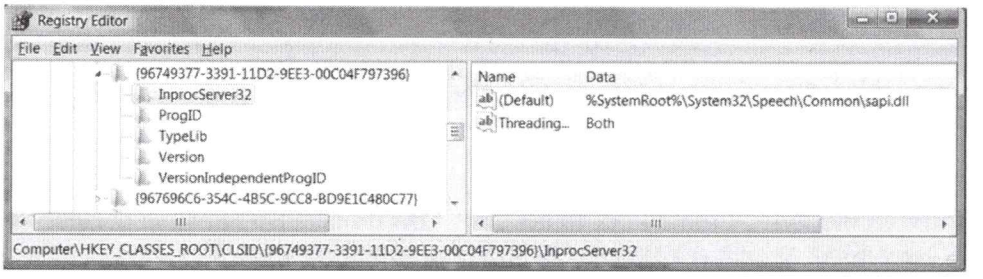
COM objects are typically loaded into your script by their programmatic identifier (ProgID) strings. The ProgID identifies a type of COM object in a version-independent way; for example, "SAPI.SpVoice" is a ProgID for an object related to the voice narrator in Windows, and it will still be a valid ProgID even if Microsoft updates the version of the DLL which implements that object. The ProgID string is mapped in the HKEY_CLASSES_ROOT branch of the registry to a unique class identifier (CLSID) number for that type of COM object, and the CLSID is mapped to the full path of the file on the drive which implements the object, usually a DLL.

In the screenshot below, "SAPI.SpVoice" is a programmatic identifier and key located under HKEY_CLASSES_ROOT. In the CLSID subkey, the default value contains a long string, called a Globally Unique ID (GUID) number, which is also the name of a key found under HKEY_CLASSES_ROOT\CLSID.




The GUID-number key contains another subkey, InprocServer32, which contains the full path to the DLL which implements the SpVoice class (sapi.dll). The GUID-number key also has a TypeLib subkey with another GUID number, this time for identifying the DLL which contains the type library for the class, which has information about the properties and methods of that class. Don't worry, you don't have to know any of these details, PowerShell already knows how to navigate and use this COM information.

Enclave Consulting LLC © 2017 - All Rights Reserved




Once the DLL for the COM object is loaded into memory, PowerShell calls a constructor method to create and initialize the object. Once created, the properties and methods of the object can be called.

If the COM object comes with a type library (TypeLib), the library can be queried to enumerate the members of the object ("type" is roughly synonymous with "class" here). Microsoft also provides documentation on many of the most important COM objects on http://msdn.microsoft.com/developer-resource-downloads-msdn/ and in downloadable Software Developer's Kits (SDKs). If there's no TypeLib available, you'll have to research the object's members manually through Google and MSDN.

When you run "`regsvr32.exe schmmgmt.dll`" to register the DLL for the Active Directory Schema snap-in, you're creating the necessary registry keys to find that DLL.

## "DLL Hell"

All the registry and operating system "plumbing" that makes COM work, though, is somewhat fragile, prone to corruption, and difficult for COM object writers to implement correctly. There's even a special term, "DLL Hell", for when DLLs are missing, corrupt, the wrong version, in the wrong location, or lacking correct registry information. Much of the flakiness of Windows has stemmed from COM and "DLL Hell" issues, and this was part of the motivation to create the .NET Framework in the first place.

## Creating COM Objects

COM objects are created in PowerShell with the new-object cmdlet. Once created, you can pipe the COM object into get-member just like .NET objects, but be aware that not all COM objects have type libraries to query, so you'll have to manually discover their properties and methods through other documentation sources or Google.

```
# Example: Creating_COM_Objects.ps1

$voice = new-object -comobject "sapi.spvoice"

$voice | get-member

$voice.speak("I have a type library!")
```

Enclave Consulting LLC © 2017 - All Rights Reserved

To launch Notepad and wait one second, and then to create the famous *WScript.Shell* COM object and start typing text into Notepad using that COM object:

```
notepad.exe

start-sleep -seconds 1

$WshShell = new-object -comobject "WScript.Shell"

$result = $WshShell.AppActivate("Untitled")

$WshShell.SendKeys("I'm sending keystrokes to notepad.exe!")
```

## -Strict Switch

When you create a COM object in PowerShell, you might not get the raw COM object back, but something that mostly looks and acts like that COM object. The .NET Framework represents data types and handles object creation/tracking/destruction differently than COM, but it's still possible to access COM objects from within .NET, and vice versa, because the .NET Framework provides "interop" services to manage the necessary .NET-to-COM translation. When a Runtime Callable Wrapper (RCW) exists for a COM object and you create that object in PowerShell, you might not get the raw COM object, you might get the interop assembly for it instead. This would be fine, except that the RCW version of the COM object sometimes will have different properties and methods than the raw COM object, hence, your code might fail because it assumes it's dealing with the original COM object.

What to do about it? PowerShell doesn't do anything different when the RCW version of the COM object is returned instead, but if you would like to be notified that this has happened, you can use the -strict switch with the new-object cmdlet. The -strict switch will raise a non-terminating error (and set $? to false) if a .NET-wrapped version of the COM object is returned instead of the raw COM object. In your code, you could test $? to see if it is false, then perhaps throw an exception or deal with the RCW object differently.

If you have Microsoft Excel installed, you can see the non-terminating error:

```
$excel = new-object -com "Excel.Application" -strict
```

The error message printed by PowerShell reads:

```
The object written to the pipeline is an instance of
type "Microsoft.Office.Interop.Excel.ApplicationClass"
from the component's primary interop assembly. If this
type exposes different members than the IDispatch
members, scripts written to work with this object
might not work if the primary interop assembly is not
installed.
```

Enclave Consulting LLC © 2017 - All Rights Reserved

But the script continues to run. If you want to suppress the error message and query the $? error variable yourself, destroy the output of the new-object command: " > `$null` " or use "`| out-null`".

Enclave Consulting LLC © 2017 - All Rights Reserved

# Appendix D: Operators and Strings

An "operator" is a command that performs an action upon one or more other items (these items are the operator's "operands"). Usually, a test is performed which yields $true or $false, or one of the operands is somehow changed by the other.

If the operands on the left-hand and right-hand sides of the operator are of different class types, PowerShell will try to convert the item on the right-hand side to match or be compatible with the item on the left-hand side. In other words, the left-hand operand stays as it is or dominates the operand on the right-hand side.

## Mathematical Operators

All the standard mathematical operators are built into PowerShell, and there are also the static methods of the *System.Math* class too.

| Mathematical Operators | Name/Description |
|---|---|
| + | Addition, plus, string concatenation. |
| - | Subtraction, minus. |
| / | Division. |
| * | Multiplication, times. |
| % | Remainder, modulus. |

The "+" operator is special. It is for arithmetic addition, but it can also concatenate (stick together) strings and append items to the end of an array, collection or hashtable.

The "*" operator is special. It is for arithmetic multiplication, but it can also be used to repeat strings and arrays. If you "multiply" a string or array by a number (the string or array must be the left-hand operand of "*" and the number must be on the right) then the string or array is expanded in size and filled with repetitions of itself.

## Assignment Operators

Assignment operators put data into the left-hand operand, which is usually a variable.

| Assignment Operators | Name/Description |
|---|---|
| = | Left operand made equal to right operand. |
| += | Adds right operand to current value of left operand. |
| -= | Subtracts right operand from current value of left. |
| *= | Multiplies left operand by the right, assigns back to left. |
| /= | Divides left operand by the right, assigns back to left. |
| %= | Divides left by right, assigns remainder only to left. |

Enclave Consulting LLC © 2017 - All Rights Reserved

The "=" assignment operator can be used to perform multiple assignments simultaneously on a single line; hence, the following are all valid statements.

```
$x = $y = $w = 383
$name = $item = $what = "SANS Institute"

# The following assigns $x = 3, $y = 7, and $w = 9:
$x, $y, $w = 3, 7, 9
```

## Comparison Operators

There are many comparison operators which yield $true or $false. By default, a comparison operator is case-insensitive, but there are case-sensitive versions. The case-sensitive operators begin with the letter "c", while the case-insensitive versions begin with "i" or don't begin with either.

Why doesn't PowerShell use comparison operators like ">", "<", ">=", "<=" and so on? Because PowerShell is not just a language, but also a command shell, and ">" and "<" are used for redirection.

| Comparison Operators | Name/Description |
|---|---|
| -eq | Equals (not case-sensitive) |
| -ieq | Equals (not case-sensitive) |
| -ceq | Equals (case-sensitive) |
| -ne | Not equals (not case-sensitive) |
| -ine | Not equals (not case-sensitive) |
| -cne | Not equals (case-sensitive) |
| -gt | Greater than (not case-sensitive) |
| -igt | Greater than (not case-sensitive) |
| -cgt | Greater than (case-sensitive) |
| -ge | Greater than or equal (not case-sensitive) |
| -ige | Greater than or equal (not case-sensitive) |
| -cge | Greater than or equal (case-sensitive) |
| -lt | Less than (not case-sensitive) |
| -ilt | Less than (not case-sensitive) |
| -clt | Less than (case-sensitive) |
| -le | Less than or equal (not case-sensitive) |
| -ile | Less than or equal (not case-sensitive) |
| -cle | Less than or equal (case-sensitive) |
| -contains | Right operand in left collection/array (not case-sensitive) |

Enclave Consulting LLC © 2017 - All Rights Reserved

| | |
|---|---|
| -icontains | Right operand in left collection/array (not case-sensitive) |
| -ccontains | Right operand in left collection/array (case-sensitive) |
| | |
| -notcontains | Right operand not in collection/array (not case-sensitive) |
| -inotcontains | Right operand not in collection/array (not case-sensitive) |
| -cnotcontains | Right operand not in collection/array (case-sensitive) |
| | |
| -in | Like -contains, but operands swapped (v.3.0+) |
| -notin | Like -notcontains, but operands swapped (v.3.0+) |

## Wildcard Matching Operators

Wildcard matching evaluates to $true or $false if the string being tested matches another string that may have simple wildcards in it. Wildcards include "*", which matches anything, and "?", which matches any single character. You can also match against any single character from a range of characters, like "[a-f]", or any single character from an explicit list of characters, like "[abcdef]" (note that "[a-f]" and "[abcdef]" match against the exact same things).

| **Match Operators** | **Name/Description** |
|---|---|
| -like | Left operand matches the pattern (not case-sensitive) |
| -ilike | Left operand matches the pattern (not case-sensitive) |
| -clike | Left operand matches the pattern (case-sensitive) |
| | |
| -notlike | Left operand does not match (not case-sensitive) |
| -inotlike | Left operand does not match (not case-sensitive) |
| -cnotlike | Left operand does not match the pattern (case-sensitive) |

In truth, the wildcard matching operators are converted into regular expressions under the hood, but that's another story.

## Logical and Bitwise Operators

Logical operators allow you to combine multiple boolean statements together to produce just one overall $true or $false result.

| **Logical Operators** | **Name/Description** |
|---|---|
| -and | If both operands are $true, yields $true. |
| -or | If one or both operands are $true, yields $true. |
| -xor | If only one operand is $true, not both, yields $true. |
| -not | Flips the boolean value of the right operand. |

Bit operators can be used to query or modify data at the bit level, such as for testing or changing bitmask fields.

Enclave Consulting LLC © 2017 - All Rights Reserved

| Bit Operators | Name/Description |
|---|---|
| -band | Binary AND: 0x1 -band 0x1 = 0x1 |
| -bor | Binary OR: 0x0 -bor 0x0 = 0x0 |
| -bxor | Binary XOR: 0x1 -bxor 0x1 = 0x0 |
| -bnot | Binary NOT: -bnot 0x1 = 0x0 |
| -shl | Shift bits left (v.3.0+) |
| -shr | Shift bits right (v.3.0+) |

## Split and Join

PowerShell 2.0 and later include the -split and -join operators for manipulating strings and arrays. The -split operator will divide a string into smaller strings using a delimeter you specify, including a delimeter defined with a regular expression pattern. The -join operator will combine multiple strings into one larger string. For examples, see the about_Split and about_Join help files.

## Redirection (>, >>)

PowerShell 3.0 and later included better support for redirection. Here are examples:

```
get-process >  output.txt      #Creates or overwrites output.txt
get-process >> output.txt      #Creates or appends to output.txt
get-process 1> output.txt      #Same as >
get-process 2> error.txt
get-process 3> warning.txt
get-process 4> verbose.txt
get-process 5> debug.txt
get-process 5>&1                 #Redirects debug to output.
get-process *> everything.txt
```

## [System.Math] and [System.Numerics.BigInt]

PowerShell can be used as a command-line calculator instead of the graphical calc.exe utility. In addition to the mathematical operators listed above, the static methods and properties of the [System.Math] class may be used for common functions. This class may be abbreviated as just [Math].

To see the mathematical functions supported by the [Math] class:

```
[Math] | get-member -static
```

To compute 4 to the 3rd power ($4^3$) or "4 * 4 * 4":

```
[Math]::Pow(4,3)
```

To round a number to two decimal places:

Enclave Consulting LLC © 2017 - All Rights Reserved

```
[Math]::Round(352.6182, 2)
```

To retrieve the value of Pi:

```
[Math]::Pi
```

To see the maximum sizes of various number types:

```
[Int32]::MaxValue          #Can be a positive or negative number.
[Int64]::MaxValue          #Can be a positive or negative number.
[Uint64]::MaxValue         #Unsigned, must be a positive number.
[Double]::MaxValue         #Uses scientific notation.
```

When dealing with *very* large integers, place the number in quotes and cast as [BigInt]:

```
[BigInt] '72992343083248709238740898308239729782393458203493845 9'
```

When using a [BigInt] in a calculation, no information is lost like when a [Double] is used with scientific notation. On the other hand, a [BigInt] cannot have a decimal point at all. So what's the point? It's at the end.

## Byte Multiplication (KB, MB, GB, TB, PB) and Scientific Notation

When working with numbers that represent bytes of data, such as the size of a file on a hard drive, you may place "KB", "MB", "GB", "TB" or "PB" immediately after a number to multiply that number by a factor of 1024 ("EB" is not supported yet). Hence, 1KB is equal to 1024 and 2KB is equal to 2048. Note that "KiB" is not supported when you wish to multiply by 1000 instead of 1024.

```
1KB   # = 1024
1MB   # = 1024 * 1024 = 1048576
1GB   # = 1024 * 1024 * 1024 = 1073741824
1TB   # = 1024 * 1024 * 1024 * 1024 = 1099511627776
1PB   # = 1024 * 1024 * 1024 * 1024 * 1024 = 1125899906842624
```

Another shorthand trick to represent large numbers is to use scientific notation for powers of ten. This is also useful when you wish to multiply numbers by a combination of 1000 instead of 1024. In scientific notation, "$3.7 * 10^3$" is the same as "3.7 * 10 * 10 * 10", which is the same as "3.7 * 1000", which is equal to 3,700. The exponent can also be a negative number in order to represent very small numbers; for example, "$3.7 * 10^{-3}$" is equal to 0.0037, and "$9.1 * 10^{-31}$" is the mass of one electron measured in kilograms.

In PowerShell, a number may be followed by "E+" or "E-" and then another number to represent a power of ten in scientific notation:

```
1E+2          # = 100 = 1 * 10^2
1E+3          # = 1000 = 1 * 10^3
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
1E+4            # = 10000 = 1 * 10^4
9.452E+14       # = 945200000000000 = 9.452 * 10^14
1E-2            # = 0.01 = 1 * 10^-2
2.74E-3         # = 0.00274 = 2.74 * 10^-3
```

In the above, you can also use a lowercase "e" instead of an uppercase "E", and the plus symbol ("+") isn't mandatory either, i.e., "2E+3" is the same as "2E3" or "2e3".

## Double-Quoted vs. Single-Quoted Strings

If a string is defined using double-quotes ( "…" ), when that string is used or assigned somewhere, PowerShell will search for and expand any variables inside that string.

If a string is defined using single-quotes ( '…' ), when that string is used or assigned somewhere, PowerShell will not search for and expand any variables inside that string, hence, PowerShell interprets that string as-is or literally. Single-quoted strings are called "literal strings" because of this.

```
$animal  = "antelope"
$string  = "An $animal ate my pizza!"
$literal = 'An $animal ate my pizza!'

$string         # Prints: An antelope ate my pizza!
$literal        # Prints: An $animal ate my pizza!
```

## Comments and Here-Strings

Single-line comments begin with a hashmark ("#") and continue to the end of the line. With PowerShell 2.0 and later, you can also have multi-line comments enclosed within "<# … #>".

```
# This line is commented out!

<# This is a
   multi-line
   comment. #>
```

A here-string literal can contain multiple lines of text and the interpreter will not try to parse or perform variable substitution within it.

Here's an example of a here-string.

```
# Example: Here-Strings.ps1

$text1 = @'
All this text, across multiple lines,
will be treated by PowerShell as one big
literal string.  $Any variable-looking text
will $be $left alone.
```

Enclave Consulting LLC © 2017 - All Rights Reserved

```
'@
```

If you want to use a here-string as a regular (non-literal) string object in which variable substitution can take place, use double-quotes instead of single-quotes.

```
$text2 = @"
What's true is $true, and
my home is folder is $home,
since my username is
$env:username
"@
```

In general, text inside of single-quotes is interpreted as literal text where no variable substitutions should occur. Text in double-quotes, on the other hand, is fair game for search-and-substitution of variables wherever they may be found.

## Casting

To "cast" a variable is to change it to a specific class type. The .NET class name is placed in square brackets. This may look like we're declaring the variable, but in PowerShell there's no need to declare variables before using them (just use a new variable whenever and wherever you need it). When you cast a variable, you can do it with the fully qualified class name, such as "[System.String]", or with the short name, such as "[String]", just as long as PowerShell can figure out what you want.

```
$x = 55    # An Int32 already by default.
[System.String] $x
[Int32] $x
$x = [System.DateTime] "18-Jun-2010"
$x = [String] "18-Jun-2010"
```

When a variable of one type is recast as a different type, PowerShell will rather aggressively try to convert the data for you, but PowerShell will avoid performing a conversion that results in a loss of data or precision. If a recast isn't possible, an error will be raised.

Enclave Consulting LLC © 2017 - All Rights Reserved