PowerShell: Section 1 Transcript

Introduction

Welcome to the PowerShell module.

Throughout this module, you'll be presented with opportunities to assess and apply what you've learned.

At the end of this module, you will be able to:

- Identify PowerShell logging and audit settings,
- Use cmdlets in PowerShell,
- Create effective Windows PowerShell commands with one line of code,
- Troubleshoot PowerShell Commands,
- Interpret a functional PowerShell script,
- Execute administrative tasks using PowerShell,
- Interact with Windows Management Instrumentation (WMI) using PowerShell,
- Interact with Common Information Model (CIM) using PowerShell, and
- Differentiate PowerShell from other Command Line Interfaces.

Bypass Exam Introduction

If you are already familiar with the subject matter presented in this module, you can choose to take a Bypass Exam to skip this module.

The Bypass Exam option provides a single opportunity to successfully demonstrate your competence with the material presented within the module. If you pass, you'll receive credit for completing the module, unlocking the content within, and you will be free to proceed to the next module. If you do not pass, you will need to successfully complete the module, including all exercises and the Module Exam, to receive credit.

Click the Next Section button to continue.

PowerShell: Section 2 Transcript

PowerShell Overview

PowerShell is Microsoft's task automation and configuration management framework platform used for Windows and Windows Server. Designed to simplify and streamline system administration, it is the de facto management standard for Windows administrators. As part of the Microsoft Engineering Common Criteria, Windows PowerShell management hooks are built into all server-based products, including Microsoft SQL Server, Exchange, System Center, and SharePoint. Familiarity with, and expertise in, PowerShell is becoming imperative, as it is no longer just nice to know and will only increase in prominence and administrative necessity.

PowerShell Overview

PowerShell consists of a Command Line Interface (CLI) shell and associated scripting language built on the .NET Framework. PowerShell is very different from the Windows CLI Command Prompt cmd.exe (CMD). PowerShell uses commands known as cmdlets, which provide full, unhindered access to Component Object Model (COM) and Windows Management Instrumentation (WMI), allowing you CLI access to take care of complex tasks. Many system administration tasks not accessible from CMD are now exposed via numerous built-in Windows console applications, or cmdlets. For example, cmdlets can be used to manage the registry for controlling Windows environments. There are approximately 1,300 cmdlets in Windows 10.

The scripting language and powerful, interactive CLI are similar to Linux and other UNIX systems, making use of pipes which allow you to pass the output of one cmdlet to the input of another cmdlet. Use multiple cmdlets in sequence to manipulate the same data. Unlike the CMD shell and those within UNIX which can only pipe streams of characters (text), PowerShell pipes objects between cmdlets. This allows PowerShell to share more complex data through cmdlets, like .NET objects. This transforms tasks that were difficult, awkward, and/or complex (such as date and time parsing), into surprisingly straightforward and simple tasks.

The PowerShell console where the commands are run was purposely created to have a similar feel to CMD. Familiar commands have been incorporated through the aliases that point these old commands to the appropriate new cmdlets, and run the new cmdlets when you type the old commands. This makes PowerShell a venerable and more powerful replacement of CMD. It also has UNIX command aliases, reducing the dreaded command line faux pas that most technology professionals experience when rolling from a UNIX command line interface to a Windows command line interface. You can find the Windows PowerShell command that any alias points to from within Windows PowerShell by using the Get-Alias cmdlet. (`get-alias cls`)

PowerShell Overview

PowerShell's Integrated Scripting Environment (ISE) gives you a command line-driven cookbook with graphical user interface (GUI) interaction to facilitate its use.

PowerShell is powerful, flexible, and is updated regularly at a pace that meets or exceeds the development and delivery of versions of Windows. The updates consistently include new features that extend its use, improve its usability, and facilitate the ability to control and manage Windows-based environments. As additional features and roles are added, so are additional cmdlets. Each version is designed to be backward-compatible, which ensures cmdlets, providers, modules, snap-ins, scripts, functions, and profiles that were designed earlier will still function as expected.

PowerShell Versions

As of February 2016, PowerShell 5.0 is installed by default on Windows Server 2016 and Windows 10, whereas PowerShell 1.0 came pre-installed in Windows Server 2008; though it was released in November 2006 for Windows XP SP2, Windows Server 2003 and Windows Vista. PowerShell is available on older operating systems through Microsoft downloads; however, it is dependent on installation of the .NET framework.

The current General Availability release of PowerShell for pre-Windows 10 systems is version 4.0, due to a recall of version 5.0.

PowerShell Components

PowerShell is organized by modules, providers, and cmdlets. During this slide, pull up your Virtual Machine (VM) and open a PowerShell console to practice the commands as they are being taught.

A Windows PowerShell module is a package that can contain cmdlets, aliases, functions, variables, type/format XML, help files, providers, and other scripts; it is basically a grouping of functions and code around a central theme or object to manage. Modules are created to manage features of diverse applications such as Microsoft Azure, Exchange, Active Directory, AWS, and VMware. Modules are routinely created and hosted for downloading, though several modules are built into the default load of PowerShell.

To obtain a listing of the modules imported into the current session, use the `Get-Module` cmdlet.

A PowerShell provider allows any data store external to the shell environment to be exposed like a file system as if it were a mounted drive. As a Microsoft .NET Framework-based program, a provider translates and transposes details to provide an easy way to access this information using a common set of cmdlets designed to work with significantly different types of data in a consistent fashion. For example, the built-in Registry provider allows you to navigate the registry like you would navigate the logical drives on your computer.

To obtain a listing of the providers, use the `Get-PSProvider` cmdlet.

Cmdlets are lightweight executable programs that take advantage of the services built into Windows PowerShell. Cmdlets are not scripts (uncompiled code), because they are built using the services of a special Microsoft .NET Framework namespace.

To obtain a listing of all the Windows PowerShell cmdlets use the `Get-Command` cmdlet. To count those, use `(Get-Command).Count`. This command also includes cmdlets, aliases, functions, workflows, filters, scripts, and applications.

PowerShell Core Modules

Through a default load, core modules are installed.

- Within PowerShell Version 2.0 and 3.0 there are 11 Core Modules.
- Within PowerShell Version 4.0 there are 12 Core Modules, and
- Within PowerShell Version 5.0 there are 18 Core Modules.

With every new version of PowerShell, the core modules build on previous versions of PowerShell. There are cmdlet and function modules included with Windows PowerShell that make up these core modules. The Windows PowerShell modules in the list provide the basic functionality of Windows PowerShell; however, anyone who can write a Windows PowerShell script has the ability to create a module. We will focus on PowerShell 4.0. Click each of the 12 core modules to learn about them.

Microsoft.PowerShell.Core Module
The Core module contains cmdlets and providers that manage the basic features of Windows PowerShell.

Microsoft.PowerShell.Diagnostics Module
The Diagnostics module contains cmdlets that manage event logs, performance counters, and Event Tracing for Windows (ETW).

Microsoft.PowerShell.Host Module
The Host module contains cmdlets that manage data from host programs.

Microsoft.PowerShell.Management Module
The Management module contains cmdlets that help you manage Windows in Windows PowerShell.

Microsoft.PowerShell.Security Module
The Security module contains the Certificate Provider and cmdlets that manage the security features of Windows PowerShell, including execution policies.

Microsoft.PowerShell.Utility Module
The Utility module contains cmdlets that manage the basic features of Windows PowerShell.

Microsoft.WSMan.Management Module
The WSMan Management module contains the WSMan Provider and cmdlets that manage Web Services for Management (WS-Management) and Windows Remote Management (WinRM).

ISE Module
The Integrated Scripting Environment (ISE) module contains cmdlets that add features to Windows PowerShell Integrated Scripting Environment (ISE).

PSScheduledJob Module
The PSScheduledJob module contains cmdlets and providers that manage scheduled jobs in Windows PowerShell.

PSWorkflow Module
The PSWorkflow module includes cmdlets that support the Windows PowerShell Workflow feature.

PSWorkflowUtility Module
The PSWorkflowUtility module includes commands that manage the Windows PowerShell Workflow feature.

PSDesiredStateConfiguration Module
The PSDesiredStateConfiguration module includes the cmdlets that manage Windows PowerShell Desired State Configuration (DSC). DSC is a management platform in Windows PowerShell that enables deployment and management of configuration data for software services, and management of the environment in which these services run.

PowerShell Module Locations

There are three default locations for Windows PowerShell modules as defined in $env:PSModulePath, which is a default environmental variable. By editing this variable you can add additional default module path locations. The three locations include the user's home directory, the Windows PowerShell Program Files directory, and the Windows PowerShell home directory. Let's talk a little more about each.

The user's home directory is not present by default. It exists only if it has been created, usually when

someone has decided to create and store modules there. When it exists, it is the first place Windows PowerShell uses when it searches for a module.

The Windows PowerShell Program Files directory usually exists in this location (C:\Program Files\WindowsPowerShell).

The Windows PowerShell home directory always exists. If the user's module directory does not exist, the modules directory within the Windows PowerShell home directory is used.

Windows PowerShell modules exist in loaded and unloaded states. To display a list of all loaded modules, use the Get-Module cmdlet without any parameters. To obtain a listing of all modules that are available (loaded and unloaded) on the system, use the `Get-Module` cmdlet with the `-ListAvailable` switch parameter.

PowerShell Providers

There are six PowerShell providers within version 4.0. Click the PowerShell providers to learn about them.

Alias (Drive={Alias})
Alias provides simplified access to all aliases defined in Windows PowerShell. Use the `Set-Location` cmdlet and specify the Alias:\ drive to work with the aliases on your computer. After that, you can use the same cmdlets you would use to work with the file system.

Environment (Drive={Env})
Environment is used to provide access to the system environment variables. If you enter `set` within a CMD shell, you will obtain a listing of all the environment variables defined on the system. You can run the old-fashioned command prompt inside Windows PowerShell by typing `CMD`.

FileSystem (Drive={C, D})
FileSystem provides access to the file system, making it the easiest to comprehend. When PowerShell is started, it automatically opens on the user documents folder where you can:

- Create both directories and files
- Retrieve properties of files and directories
- Delete files and directories
- Open files and append or overwrite data to the files through inline code or by using the pipelining feature

Function (Drive={Function})
Function provides access to the functions defined in PowerShell through a file system-based model. Through the function provider, you can:

- Obtain a listing of all the functions on your system
- Add, modify, and delete functions

Registry (Drive={HKLM, HKCU})
Registry supplies access to the registry in the same manner that the filesystem provider permits access to a local disk drive. The equivalent cmdlets used to access the file system: `New-Item`, `Set-Item`, `Get-ChildItem`, `Remove-Item`, etc. also work with the registry.

The inclusion of remote computing with Windows PowerShell 2.0 made it possible to create remote registry changes as easily as changing the local registry. The registry provider supplied with PowerShell 3.0 improved with the introduction of transactions; whereas, there have been no major increases in functionality with the registry provider in PowerShell 4.0 and PowerShell 5.0.

Variable (Drive={Variable})
Variable provides access to the variables that are defined within PowerShell. You can obtain a listing of the cmdlets designed to work specifically with variables by using the `Get-Help` cmdlet and specifying `*variable` as a value for the `-Name` parameter. These variables include:

- User-defined variables, such as `$broncs`
- System-defined variables, such as `$host`

PowerShell Cmdlets

Cmdlets use a standard naming convention that follows a verb-noun pattern such as `Get-Module`, `Get-Command`, or `Get-PSProvider`. When cmdlets use the "Get" Verb pattern, they display information about the item on the right side of the dash. When cmdlets use "Set" Verb, they modify or set information about the item on the right side of the dash. An example of a cmdlet that uses Set-Verb is `Set-AppLockerPolicy`, which can be used to set the AppLocker policy for a specified Group Policy Object (GPO). All cmdlets use one of the standard verbs. To find all of the standard verbs, you can use the Get-Verb cmdlet. In Windows PowerShell 4.0 there are nearly 100 approved verbs.

PowerShell Cmdlets

Cmdlets return objects instead of string values, which provide significantly more detailed information than if you only had strings. In fact, to take information from one cmdlet and feed it to another cmdlet, you can use the pipe character (`|`). On the surface this could appear complicated, but is actually very easy, and becomes more and more natural with every use. At the most basic level, consider obtaining a directory listing; after you have the directory listing, perhaps you would like to reduce the details and format the way it is displayed as a table or a list. This can be divided into three separate operations. The second and third tasks take place on the right side of the pipe (`Get-ChildItem C:\ | select name, length, LastAccessTime | Format-Table -AutoSize`).

1. Obtain the directory information
2. Select name, length, and last access time
3. Format the output

Get-Command

The `Get-Command` cmdlet retrieves details of every command available within your PowerShell environment, including cmdlets, functions, workflows, aliases, and executable commands. It is easily one of the most useful cmdlets. Through the Get-Command (alias `gcm`) cmdlet, you can obtain a listing of all cmdlets installed; however, this is only the beginning, as it is much more versatile. For example, you can use wildcard characters to search for cmdlets.

Use the `gcm` alias to get the `Get-Command` cmdlet, and pipeline the output to the `Format-List` cmdlet. Use the wildcard asterisk (`*`) to obtain a listing of all the properties of the `Get-Command` cmdlet.

```
gcm Get-Command | Format-List *
```

Because objects, and not string data, are returned from cmdlets, you can also retrieve the definition of the `Get-Command` cmdlet by directly using the definition property. This is done by putting the expression inside parentheses and using dotted notation, which will provide results almost identical to the output seen here. `(gcm Get-Command).definition`

You can use the `gcm` alias to specify the -Verb or -Noun parameters such as `u*` for the verb and `p*` for the noun.

```
gcm -verb u*
gcm -noun p*
```

PowerShell Compared to Other CLIs

PowerShell includes aliases for commands in other well-known CLIs such as cmd.exe, command.exe, and UNIX shell, making it an easy transition for those familiar with other common shells to productivity. Here are some of those commands. Click each command to learn more.

PowerShell Compared to Other CLIs

Though PowerShell enables productivity immediately for those familiar with other CLIs, there are significant differences.

Here are some features built within PowerShell.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

PowerShell: Section 3 Transcript

PowerShell Scripts

PowerShell 5.0 on Windows 10 and PowerShell 4.0 in Windows 8 and Windows Server 2012 bring a watershed of cmdlets for operating system automation. As a result, PowerShell scripting has become less and less necessary. However, because scripts were needed to address recurring needs within PowerShell 1.0 through 3.0, they remain an important part of the PowerShell recipe for success, and at times are essential to comprehension and dissection.

PowerShell scripting is designed to implement complex operations using cmdlets through support of variables, functions, branching (if-then-else), loops (while, do, for, and foreach), and structured error/exception handling, as well as integration with .NET. For example, consider the activity of producing a directory listing. The simple Get-ChildItem cmdlet does a worthy job, but after you choose to filter for only .zip files and sort them in descending order by the time span between CreationTime and LastWriteTime, it becomes more complicated. If you do this, you will get a command like the one you see here.

```
Get-ChildItem *.zip | Sort-Object -Property
@{Expression={$_.LastWriteTime - $_.CreationTime}; Ascending=$false} |
Format-Table LastWriteTime, CreationTime
```

Say you want to run a command that displays the services on a computer in descending Status order and ascending DisplayName order. You would use the `Get-Service` cmdlet to get the services on the computer followed with a pipeline operator (`|`) to send services to the `Sort-Object` cmdlet. In order to sort one property in ascending order and another property in descending order, use a hash table for the value of the Property parameter. The hash table uses an Expression key to specify the property name, and an Ascending or Descending key to specify the sort order.

A calculated property is a hash table (`@{Name=Value; Name=Value}`). The first key is either Name or Label and the second is Expression. The value of the Name (or Label) key is the name that you want to assign to the property. The value of the Expression key is a script block (inside braces) that receives the property value.

The resulting display, which sorts the Status values in descending order, lists properties with a Status value of "Running" before those with a Status value of "Stopped".

```
Get-Service | Sort-Object -Property
@{Expression="Status";Descending=$true},
@{Expression="DisplayName";Descending=$false}
```

Even if you use tab completion, the previous commands require a significant amount of typing. This can be shortened through two means.

- A user-defined function, or a
- PowerShell script.

PowerShell Script Anatomy

Let's look at the anatomy of a PowerShell script.

Variables in PowerShell scripts have names that start with $. They can be assigned any value, including the output of cmdlets.

Strings can be enclosed either in single quotes or in double quotes. When using double quotes, variables will be expanded even if they are inside the quotation marks.

Enclosing the path to a file in braces preceded by a dollar sign (`${C:\broncs.txt}`) creates a reference to the contents of the file. If an object is assigned, it is serialized before being stored.

Object members can be accessed using "." notation, as in C# syntax. PowerShell provides special variables, such as $args, which is an array of all the command line arguments passed to a function from the command line, and $_, which refers to the current object in the pipeline. PowerShell also provides arrays and associative arrays. The PowerShell scripting language also immediately evaluates arithmetic expressions entered on the command line, and parses common abbreviations, such as GB, MB, and KB.

One reason that administrators write PowerShell scripts is to run the scripts as scheduled tasks. You can create, monitor, and delete scheduled jobs using the Win32_ScheduledJob WMI class.

The `ListProcessesSortResults.ps1` script, shown here, is a script that a network administrator might want to schedule to run several times a day. It produces a list of currently running processes and writes the results out to a text file as a formatted and sorted table.

```
$args = "localhost","loopback","127.0.0.1"

foreach ($i in $args)
{$strFile = "c:\orange\"+ $i +"bronc-processes.txt"
Write-Host "Testing" $i "coming soon to a network near you ...";
Get-WmiObject -computername $i -class win32_process |
Select-Object name, processID, Priority, ThreadCount, PageFaults,
PageFileUsage |
Where-Object {!$_.processID -eq 0} | Sort-Object -property name |
Format-Table | Out-File $strFile}
```

PowerShell Script Creation and Execution

A PowerShell script is a collection of cmdlets that you can put into PowerShell and run directly as written or save to a text file with a .ps1 extension. If you create the file in PowerShell ISE it will be saved automatically as a .ps1. By default, running a script requires updated execution policy; an attempt to run a PowerShell script generates an error message like the one you see here.

```
C:\Users\pmanning\Desktop\sb50win.ps1 : File
C:\Users\pmanning\Desktop\sb50win.ps1 cannot be loaded because running
scripts is disabled on this system. For more information, see
about_Execution_Policies at http://go.microsoft.com/fwlink/?
LinkID=135170.
At line:1 char:1
+ C:\Users\pmanning\Desktop\sb50win.ps1
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
+ CategoryInfo : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
```

By default, Windows PowerShell prohibits the execution of scripts; however, this can be controlled through Group Policy or by updating administrative rights on your computer using the Set-ExecutionPolicy cmdlet to turn on script support.

When you run Set-ExecutionPolicy it tells you what position the policy is currently in, asks for an updated value, and gives you a rare PowerShell warning. Here is an example of this exchange.

```
cmdlet Set-ExecutionPolicy at command pipeline position 1
```

```
Supply values for the following parameters:
ExecutionPolicy: 1

Execution Policy Change The execution policy helps protect you from
scripts that you do not trust. Changing the execution policy might
expose you to the security risks described in the
about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change
the execution policy?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): n
```

As you can see, the policy is currently in ExecutionPolicy 1. PowerShell provides a warning, and asks if you want to change the execution policy.

There are six levels that can be turned on by using the Set-ExecutionPolicy cmdlet, and three different scopes. Click each one to learn more.

Levels

1. Restricted - Does not load configuration files such as the Windows PowerShell profile or run other scripts. Restricted is the default setting.
2. AllSigned - Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.
3. RemoteSigned - Requires that all scripts and configuration files downloaded from the Internet be signed by a trusted publisher.
4. Unrestricted - Loads all configuration files and runs all scripts. If you run an unsigned script that was downloaded from the Internet, you are prompted for permission before it runs.
5. Bypass - Blocks nothing and issues no warnings or prompts.
6. Undefined - Removes the currently assigned execution policy from the current scope. This parameter will not remove an execution policy that is set in a Group Policy scope.

Scopes

1. Process - The execution policy affects only the current Windows PowerShell process.
2. CurrentUser - The execution policy affects only the current user.
3. LocalMachine - The execution policy affects all users of the computer. Setting the Local-Machine execution policy requires administrator rights on the local computer.

By default, non-elevated users have rights to set the script execution policy for the CurrentUser user scope that affects their own execution policies.

PowerShell Script Creation and Execution

To view the execution policy for all scopes, use the -List parameter when calling the Get-ExecutionPolicy cmdlet.

In order to run a script, enable execution and open the PowerShell console. Next, drag the .ps1 file to the console and press Enter (as long as the script name does not have spaces in the path, otherwise it will not run). You can also right-click the script file and select Copy As Path, right-click inside the Windows PowerShell console to paste the path of your script there, and press Enter, you will print out a string that represents the path. This is because the Copy As Path option will automatically surround the path with quotes.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use

your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

PowerShell: Section 4 Transcript

PowerShell ISE

1/4

PowerShell ISE is a GUI application for PowerShell. ISE lets you write, edit, run, test, and debug scripts and run commands, in an environment that displays syntax in colors and supports Unicode. ISE comes packaged with default installs of PowerShell in all Microsoft operating systems since Windows 7. ISE provides multiple execution environments, a built-in debugger, and the extensibility of the ISE object model. PowerShell ISE is an outstanding scripting tool; however, it is also a powerful command line interface as an alternative to the PowerShell console.

PowerShell ISE Benefits

2/4

Two benefits of PowerShell ISE include copying and pasting.

Within the PowerShell console, you mark text as a block instead of selecting it line by line; text cannot be selected without the use of a mouse. In the PowerShell ISE, you can highlight text as in any editor, line by line or by using the SHIFT + cursor keys. You can copy the text as you are used to, with CTRL+C.

Pasting is also difficult on the PowerShell console. The fastest way to paste is to right-click (if Quick Edit Mode is enabled); real shell connoisseurs avoid clicking whenever possible, but the alternative pasting method, ALT+SPACE+E+P, is not convenient. In PowerShell ISE, you can just paste as in any editor, with CTRL+V.

PowerShell Remoting

3/4

PowerShell remoting is a functionality that Windows administrators can use to connect to remote computers if they're running at least PowerShell version 2.0 and have this option enabled. To control who can connect to the computer and what operations can be performed on it, you'll use session configurations, known as endpoints. Let's take a look at the different types of endpoints.

When you use PowerShell remoting, you usually connect to the default endpoint, Microsoft.PowerShell. You can change this either by using the parameter ConfigurationName on *-PSSession cmdlets or by changing the preference variable `$PSSessionConfigurationName`. To work with remoting endpoints on your computer you use cmdlets with the common noun `PSSessionConfiguration`:

```
PS C:\> Get-Command -Noun PSSessionConfiguration | select Name
```

Exercise Introduction

4/4

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

PowerShell: Section 5 Transcript

Administrative Use Cases

A quick Google search of PowerShell reveals broad use of PowerShell across the spectrum of technology professionals. Within this module, we look at PowerShell from an administrator perspective.

PowerShell is valuable for

- Live incident response,
- System triage,
- Penetration testing methodology, and
- Host and network information gathering.

It is also becoming a go-to tool for hackers.

Administrative Use Cases

As the de facto Microsoft management standard from home to enterprise administrators, the user population and diversity is considerable. A few of the abundant administrative use cases include:

- Standardize and automate the installation and configuration of Microsoft networking components,
- Automate the day-to-day management of Windows networks,
- Collect information or configure settings on Windows machines,
- Administer Active Directory, and
- Administer Microsoft Exchange.

Incident Response Use Cases

Microsoft has continually evolved its technology, and has introduced some tools that can be used for incident response through intrusion analysis and system triage. PowerShell now fits into this role and incorporates powerful Linux command line functionally like parsing (i.e. equivalents of cut, sort, uniq, etc.), regular expressions, piping, and redirection. A handful of the many use cases for intrusion analysis and incident response include:

- Perform monitoring tasks of Microsoft Security technologies such as Windows Firewall, Active Directory, and Windows Event Logs,
- Perform regular expression pattern matching,
- Check the integrity of network monitoring, and
- Parse and analyze security events.

PowerShell can also:

- Execute commands,
- Reflectively load/inject code,
- Enumerate files,
- Interact with services,
- Retrieve event logs,
- Download files from the Internet,
- Interface with Win32 API,
- Interact with the registry,

- Examine processes, and
- Access .NET framework.

Penetration Testing

As with any technology created to facilitate administration and improve security through function and efficiency, PowerShell can be used for good or bad intentions. Administration services and programs typically have the highest rights on computers, so hacking through such a portal or service would be ideal for a penetration tester. If an application is trusted and simplifies administrative tasks, then it is easier for penetration testers to use it. If an application is not trusted, then it's more difficult to run.

For example: polling an entire organization for the current Common Vulnerabilities and Exposures (CVE) patch level for a security assessment, a penetration tester can gather existing CVEs available in the network for exploitation.

PowerShell fits into this category, and is a favorite tool for penetration testers. In fact, several PowerShell attack frameworks currently exist, such as Empire, Nishang, and PowerSploit.

Epic Breach: PowerShell and Logging

PowerShell is often used in attacks; the problem is that, by default, Windows only logs that PowerShell was launched, with no additional details about what exactly happened; however, if PowerShell calls additional programs and possibly opens network sessions, then these details are logged.

Security teams have made these observations and PowerShell is now becoming more regulated and monitored, with enhanced logging recognizing execution of encoded and obfuscated commands requiring an extra degree of observance. An example of an attack in 2015 that incorporated PowerShell was provided at ShmooCon 2016 during a track titled "No Easy Breach". It was based on an incident response by Mandiant who investigated and remediated an unwavering all-out, presumed advanced persistent threat (APT)/nation-state breach. Mandiant explained this was one of the largest, most advanced and consistently evolving breaches they had ever fielded. The attackers used series upon series of evolving attacks on the network using:

- PowerShell,
- WMI,
- Kerberos attacks,
- Novel persistence mechanisms,
- Seemingly unlimited Command and Control (C2) infrastructure, and
- A half-dozen rapidly-evolving malware families across a 100k-node network used to compromise the environment at a rate of 10 systems per day.

PowerShell auditing and logging was not enabled, which basically gave attackers a cloak of invisibility, creating a game of perpetual catch up with the attackers running circles around the incident responders. As soon as the response team figured out PowerShell was a mainstay tool, respective auditing and logging across the board was enabled. Incident responders were able to monitor and predict adversary attacker Tools, Techniques, and Procedures (TTPs). This is one of the many nails in the coffin which ultimately led to this APT extermination.

Epic Breach: PowerShell and Logging

A handful of attacker TTPs include:

- Using WMI to create persistent backdoors and schedule backdoors to be extracted and executed in the future, which sometimes takes months to ensure a future foothold,
- Using PowerShell to encode, obfuscate and execute backdoors and to run Invoke-MimiKatz in

order to evade Endpoint Security Products (ESP),
- Embedding PowerShell code in WMI class properties in order to execute on remote systems, and
- Attacking Kerberos tickets to make tracking of lateral movement difficult.

PowerShell Logging - Incident Response Lessons Learned

It is becoming commonplace for incident response Lessons Learned to include a reported lack of available logging to adequately show what actions the attacker performed using PowerShell.

Within an incident response, the Lessons Learned phase documents and records steps that were not taken during the incident, and catalogues additional documentation that might be beneficial in the future.

Here are some PowerShell Lessons Learned:

- Upgrade and enable Module Logging if possible,
- Baseline legitimate PowerShell usage,
- Restrict ExecutionPolicy setting,
- Understand the script naming conventions and existing paths,
- Identify if remoting is enabled,
- Understand use privilege,
- Understand the common source and destination systems, and
- Finally, recognize artifacts of anomalous usage.

PowerShell Logging - Administrative Diligence

As an administrator, it is imperative to understand logging and audit settings to protect your organization.

Increasing PowerShell logging and auditing settings could enable detection of malicious activity and provide a veritable historical record of how PowerShell was used on systems.

Let's look at various PowerShell logging options and how they can facilitate the visibility needed to better monitor, respond, investigate, and remediate attacks involving PowerShell.

PowerShell Logging - Version 2.0 Inadequacies

By default, PowerShell does not leave many artifacts of its execution in most Windows environments; however, Microsoft has been taking steps to improve the security transparency of PowerShell in recent versions.

As highlighted with the Epic Breach, the blend of functionality and stealth has made attacks leveraging PowerShell a nightmare for security.

PowerShell 2.0, which comes installed on all Windows 7 and Server 2008 R2 systems, provides very little evidence of attacker activity.

The Windows event logs show that PowerShell executed, the start and end times of sessions, and whether the session executed locally or remotely (ConsoleHost or ServerRemoteHost).

The logs reveal nothing about what was executed with PowerShell. This figure shows an example of the event log messages recorded in the PowerShell 2.0 log Windows PowerShell.evtx.

PowerShell Logging - Version 5.0 Inadequacies

Microsoft significantly enhanced logging within the PowerShell version 5.0 release in Windows 10

systems.

This enhanced logging records executed PowerShell commands and scripts, deobfuscated code, output, and transcripts of historic activity. Enhanced PowerShell logging is an invaluable resource, both for enterprise monitoring and administration.

PowerShell Logging - Installation

Windows 10 does not require any software updates to support enhanced PowerShell logging.

For Windows 7/8.1/Server 2008 R2/Server 2012 R2, updating PowerShell to enable enhanced logging requires:

- .NET 4.5
- Windows Management Framework (WMF) 4.0
- The appropriate WMF 4.0 update
    - 8.1/2012 R2: KB300850
    - 2012: KB3119938
    - 7/2008 R2 SP1: KB3109118

PowerShell Logging Configuration

Logging is configured by the Group Policy through the Local Group Policy Editor (PowerShell configuration options). The policy editor for Windows PowerShell can be found through Administrative Templates → Windows Components → Windows PowerShell.

PowerShell supports three types of logging: module logging, script block logging, and transcription. PowerShell events are written to the Application Logs and Analytic Logs.

Application Logs

- Windows PowerShell.evtx
- Microsoft-Windows-PowerShell/Operational.evtx
- Microsoft-Windows-WinRM/Operational.evtx

Analytic Logs

- Microsoft-Windows-PowerShell/Analytic.etl
- Microsoft-Windows-WinRM/Analytic.etl

We'll go over some of the specific logs found in these locations later. First, let's discuss the three different types of logging mechanisms.

PowerShell Module Logging

Module logging records pipeline execution details as PowerShell executes, including variable initialization and command invocations. Module logging will record portions of scripts, some deobfuscated code, and some data formatted for output. This logging will capture some details missed by other PowerShell logging sources, though it may not reliably capture the commands executed. Module logging has been available since PowerShell 3.0. Module logging events are written to Event ID (EID) 4103 or 800.

PowerShell Module Logging

To enable module logging:

- In the Windows PowerShell GPO settings, set Turn on Module Logging to enabled.
- In the Options pane, click the button to show Module Name.
- In the Module Names window, enter * to record all modules.
    - Optional: To log only specific modules, specify them here. (Note: this is not recommended.)
- Click OK in the Module Names window.
- Click OK in the Module Logging window.

Alternately, setting the displayed registry values will have the same effect.

```
HKLM\SOFTWARE\Wow6432Node\Policies\Microsoft\Windows\PowerShell\ModuleLogging
EnableModuleLogging = 1
```

```
HKLM\SOFTWARE\Wow6432Node\Policies\Microsoft\Windows\PowerShell\ModuleLogging
\ModuleNames * = *
```

PowerShell Script Block Logging

Script block logging records blocks of code as they are executed by the PowerShell engine, thereby capturing the full contents of code executed by a hacker, including scripts and commands.

Due to the nature of script block logging, it also records deobfuscated code as it is executed.

For example, in addition to recording the original obfuscated code, script block logging records the decoded commands passed with PowerShell's EncodedCommand argument, as well as those obfuscated with XOR, Base64, ROT13, encryption, etc.

Script block logging will not record output from the executed code. Script block logging events are recorded in EID 4104.

PowerShell 5.0 automatically logs code blocks if the block's contents match a list of suspicious commands or scripting techniques, even if script block logging is not enabled; this function is not available in PowerShell 4.0.

These suspicious blocks are logged at the warning level in EID 4104, unless script block logging is explicitly disabled.

Although not considered a security feature by Microsoft, this ensures that some forensic data is logged for known suspicious activity, even if logging is not enabled.

Enabling script block logging will capture all activity, not just blocks considered suspicious by the PowerShell process, which allows investigators to identify the full scope of hacker activity.

The blocks that are not considered suspicious will also be logged to EID 4104, but with in-depth details.

PowerShell Script Block Logging

Script block logging generates fewer events than module logging, and records valuable indicators for alerting in a Security Information and Event Management (SIEM) or log monitoring platform.

Group Policy also offers an option to "Log script block execution start/stop events". This option records the start and stop of script blocks, by script block ID, in EIDs 4105 and 4106. This option may provide additional forensic information, as in the case of a PowerShell script executing over a long period, but it generates a prohibitively large number of events and is not recommended for most environments.

To enable script block logging:

In the "Windows PowerShell" GPO settings, set "Turn on PowerShell Script Block Logging" to enabled.

Alternately, setting the displayed registry value will enable logging.

```
HKLM\SOFTWARE\Wow6432Node\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging
EnableScriptBlockLogging = 1
```

PowerShell Logging - Transcription

Transcription creates a distinctive record of every PowerShell session, including all input and output, exactly as it appears in the session. Transcripts are written to text files, broken out by user and session. Transcripts also contain timestamps and metadata for each command in order to aid analysis. However, transcription records only what appears in the PowerShell terminal, which will not include the contents of executed scripts or output written to other destinations such as the file system.

PowerShell transcripts are automatically named to prevent collisions, with names beginning with PowerShell_transcript. By default, transcripts are written to the user's documents folder, but can be configured to any accessible location on the local system or on the network. The best practice is to write transcripts to a remote write-only network share, where defenders can easily review the data and hackers cannot easily delete them. Transcripts are very storage efficient, easily compressed, and can be reviewed using standard tools like grep.

PowerShell Logging – Enable Transcription

To enable transcription:

- In the Windows PowerShell GPO settings, set "Turn on PowerShell Transcription" to enabled.
- Check the "Include invocation headers" box, in order to record a timestamp for each command executed.
- Optionally, set a centralized transcript output directory.

This directory should be a write-only, restricted network share that security personnel can access. If no output directory is specified, the transcript files will be created under the user's documents directory.

Alternately, setting the displayed registry values will enable logging.

```
HKLM\SOFTWARE\Wow6432Node\Policies\Microsoft\Windows\PowerShell\Transcription
EnableTranscription = 1
HKLM\SOFTWARE\Wow6432Node\Policies\Microsoft\Windows\PowerShell\Transcription
EnableInvocationHeader = 1
HKLM\SOFTWARE\Wow6432Node\Policies\Microsoft\Windows\PowerShell\Transcription
OutputDirectory ="" (Enter path. Empty = default)
```

PowerShell Log and Audit Settings Best Practice

Where possible, best practice includes enabling all three log sources:

- Module logging,
- Script block logging, and
- Transcription.

Each of these sources records unique data valuable to analyzing PowerShell activity. In environments where log sizes cannot be significantly increased, enabling script block logging and transcription will record most activity, while minimizing the amount of log data generated. At a

minimum, script block logging should be enabled, in order to identify hacker commands and code execution.

Ideally, the size of the PowerShell event log (MicrosoftWindowsPowerShell/Operational.evtx) should be increased to 1GB in order to ensure that data is preserved for a reasonable period.

PowerShell logging generates large volumes of data that quickly rolls the log. Up to 60MB per hour has been observed during typical admin or hacker activity.

PowerShell Log and Audit Settings Remoting

The Windows Remote Management (WinRM) log (MicrosoftWindowsWinRM/Operational.evtx) records inbound and outbound WinRM connections, including PowerShell remote connections. The log captures the source (inbound connections) or destination (outbound connections), along with the username used to authenticate. This connection data can be valuable in tracking lateral movement using PowerShell remoting. Ideally, the WinRM log should be set to a sufficient size to store at least one year of data.

Due to the large number of events generated by PowerShell logging, organizations should carefully consider which events to forward to a log aggregator. In environments with PowerShell 5.0, organizations should consider, at a minimum, aggregating and monitoring suspicious script block logging events, EID 4104 with level "warning", in a SIEM or other log monitoring tool. These events provide the best opportunity to identify evidence of compromise while maintaining a minimal dataset.

Event Logs - Local PowerShell Execution

When a hacker interacts with a target host through local PowerShell script execution or PowerShell remoting, several event logs are created when enabled.

Windows PowerShell.evtx
Event ID 400 and 403 record the start & stop times of a PowerShell session
EID 400: Engine state is changed from None to Available.r
HostName=ConsoleHost
EID 403: Engine state is changed from Available to Stopped.
HostName=ConsoleHost

Microsoft-Windows-PowerShell/Operational.evtx
(PowerShell 3.0 and greater)
Event ID 40961 records the start time of PowerShell session
EID 40961: PowerShell console is starting up
Event ID 4100 is an error message that provides path to PowerShell script
EID 4100: Error Message = File C:\temp\crazy.ps1 cannot be loaded because running scripts is disabled on this system

Microsoft-Windows-PowerShell/Analytic.etl

(Log disabled by default. Events exclusive to PowerShell 3.0 or greater)
The Event ID 7937 records executed cmdlets, scripts, or commands (no arguments)
EID 7937: Command crazy.ps1 is Started.
EID 7937: Command Write-Output is Started.
EID 7937: Command imhere.exe is Started

PowerShell Event Logs - Remoting

The event logs you see here are created when a hacker interacts with a target host through PowerShell remoting.

Windows PowerShell.evtx
Event ID 6 records the start of a remoting session (client host)
EID 6: Creating WSMan Session. The connection string is: 192.168.1.10/wsman? PSVersion=3.0
Event ID 400 and 403 record the start & stop of a PowerShell remoting session
EID 400: Engine state is changed from None to Available. HostName= ServerRemoteHost
EID 403: Engine state is changed from Available to Stopped. HostName= ServerRemoteHost

PowerShell Event Logs - Remoting (Accessed Host)

These event logs are created during access of a target host through PowerShell remoting.

Microsoft-Windows-WinRM/Operational.evtx
Event ID 169 records who connected via remoting
EID 169: User CORP\VonM authenticated successfully using NTLM
Event ID 81 and 134 record the timeframe of the remoting activity
EID 81: Processing client request for operation CreateShell
EID 134: Sending response for operation DeleteShell

Microsoft-Windows-PowerShell/Analytic.etl
The Event ID 32850 records who connected via remoting
EID 32850: Request 7775818. Creating a server remote session. UserName: CORP\JohnE
The Event ID 32867 and 32868 records the encoded contents of remoting I/O
EID 32867: Received remoting fragment […] Payload Length: 777 Payload Data: 0x020000000200010064D64FA51E7C68428483DB[…]
EID 32868: Sent remoting fragment […]Payload Length: 180 Payload Data: 0xEFBBBF3C4F626A2052657794643D2230232E3[…]

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

PowerShell: Section 6 Transcript

PowerShell WMI

WMI is a hierarchical namespace, in which the layers build on one another, like the Lightweight Directory Access Protocol (LDAP) directory used in Active Directory Domain Services (AD DS) or the file system structure on your hard drive. Although WMI is a hierarchical namespace, the term by itself does not really carry the wealth of WMI. The WMI model has three sections: resources, infrastructure, and consumers.

WMI resources include:

- Anything that can be accessed by using WMI, such as the file system, networked components, event logs, files, folders, disks, and AD DS.

WMI infrastructure is made up of three parts

- The WMI service,
- The WMI repository, and
- The WMI providers.

Of these parts, WMI providers are the most important because they provide the means for WMI to gather needed information.

WMI consumers provide a way to process prepackaged data from WMI. A consumer can be:

- A Windows PowerShell cmdlet,
- A Microsoft Visual Basic Scripting Edition (VBScript) script,
- An enterprise management software package, or
- Some other tool or utility that executes WMI queries.

PowerShell CIM

The CIM exposes an application programming interface (API) for working with WMI information. The CIM cmdlets support multiple ways of exploring WMI. Namespaces are not very discoverable, and finding them can be tedious and difficult; this is where tab completion comes into play. To simplify this process, enter enough of the namespace name to uniquely distinguish it, and then press the Tab key on the keyboard. This tab expansion expands the namespace when you use the CIM cmdlets, so you can explore namespaces easily. You can even drill down into namespaces by using this technique of tab expansion. All CIM cmdlets support tab expansion of the namespace parameter, in addition to the `-ClassName` parameter. But to explore WMI classes, you will want to use the cmdlet specifically designed for class exploration, the `Get-CimClass` cmdlet. It is notable that the default WMI namespace on all operating systems after Windows NT 4.0 is Root/Cimv2 and in turn all of the CIM cmdlets default to Root/Cimv2.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

PowerShell: Section 7 Transcript

Summary

You have completed the PowerShell module.

You should now be able to:

- Identify PowerShell logging and audit settings,
- Use cmdlets in PowerShell,
- Create effective Windows PowerShell commands with one line of code,
- Troubleshoot PowerShell Commands,
- Interpret a functional PowerShell script,
- Execute administrative tasks using PowerShell,
- Interact with Windows Management Instrumentation (WMI) using PowerShell,
- Interact with Common Information Model (CIM) using PowerShell, and
- Differentiate PowerShell from other Command Line Interfaces.

To receive credit and advance to the next module, you must achieve a passing score on the Module Exam. Click the Next Slide button to begin the Module Exam.