Fundamentals of Leveraging PowerShell

By
Carlos Perez

Instructor

- Carlos Perez (Twitter @carlos_perez)
- Day job is Director of Reverse Engineering at a security vendor.
- Microsoft MVP on Cloud and Server Management
- Metasploit contributor
- Co-Host in the Paul's Security Weekly Podcast

Why PowerShell?

Why PowerShell?

- Red:
  - Installed by default on Windows 7/2008 R2 and above.
  - Many defenders have not kept up to date on how to configure and control it.
  - Rarely upgraded to the latest versions do to legacy infrastructure or lack of skill/knowledge on how to do it.
  - Provides access to all the APIs available on a Windows system.

Why PowerShell?

- Blue:
  - Installed by default on Windows 7/2008 R2 and above.
  - Provides access to all the APIs available on a Windows system.
  - Better controls and logging that other scripting engines.
  - Its use and abuse can be tracked across the entire enterprise with centralized logging.

Why PowerShell?

Type	.Net	Other COM	ADSI	WMI	Win32 API	System Executables
PowerShell	Yes	Yes	Yes	Yes	Yes	Yes
VBScript	No	Yes	Yes	Yes	No	Yes
JScript	No	Yes	Yes	Yes	No	Yes
WSH	No	Yes	Yes	Yes	No	Yes
HTA	No	Yes	Yes	Yes	No	Yes
CMD/BAT	No	No	No	No	No	Yes

Why PowerShell?

Type	Event Log	Transcript	Enhanced Logging	AppLocker	CodeSign Enforcement	API Constraint	AMSI Inspected	Command Filtering
PowerShell	Yes	Yes**	Yes*	Yes	No	Yes **	Yes	Yes
VBScript	No	No	No	Yes	Yes	No	Yes	No
JScript	No	No	No	Yes	Yes	No	Yes	No
WSH	No	No	No	Yes	Yes	No	Yes	No
HTA	No	No	No	No	No	No	No	No
CMD/BAT	No	No	No	Yes	No	No	No	No

* PSv3 and above

** PSv5 and above

What is PowerShell?

What is PowerShell?

- PowerShell is an automation and configuration tool/framework that works well with your existing tools and is optimized for dealing with structured data (e.g. JSON, CSV, XML, etc.), REST APIs, and object models.
- There are 2 main versions of PowerShell:
  - **Windows PowerShell**: Ships with Windows and it is based on a full .Net framework CLR.
  - **PowerShell Core**: Open Source version that is multiplatform and it is based on .Net Core CLR

What is PowerShell?

- As of version 6.x PowerShell is a cross-platform (Windows, Linux, and OS X) It includes a command-line shell, an associated scripting language and a framework for processing cmdlets, it is called PowerShell Core.
- It is licensed under MIT License for versions after 6.0
- Source code and build instructions at https://github.com/PowerShell/PowerShell

What is PowerShell?

- On Windows:
  - PowerShell is not **powershell.exe**!
  - PowerShell is not **powershell_ise.exe**!
  - Those are host program that run the engine.
  - **System.Management.Automation.dll** and all its copies in the system are engine of Windows PowerShell.
  - There are multiple copies of it in your systems :)

Problems with PowerShell

- Fragmentation
  - After Windows 7 and Windows 2008 R2 PowerShell comes as part of the OS and not updated by Windows Update.
  - PowerShell comes as part of the Windows Management Framework and several old servers tools are incompatible with new versions on Windows.
  - The latest versions of WMF do not appear in the WSUS Catalog making deployment of newer versions a manual and complex process.
  - Some cmdlets and modules are only available on newer versions of Windows even when they all run the latest version of WMF

Handling Caveats

	2.0	3.0	4.0	5.x	Core 5.x	6.x
Windows 7/2008 R2	Default	Supported	Supported	Supported	Not Supported	Not Supported
Windows 8/2012		Default	Not Supported	Supported	Not Supported	Not Supported
Windows 8.1/2012R2			Default	Supported	Not Supported	Supported
Windows 10/2016				Defuault	Not Supported	Supported
Windows Nano					Default	Supported
Windows IoT					Default	Supported
Linux/OS X						Supported

Handling Caveats

- Use PowerShell 2.0 as the lowest denominator since it is the first version of PowerShell included by default on Windows.
- Use .NET Framework 3.5 as the the lowest denominator when selecting .NET API Calls.

Cmdlet

Cmdlet

- PowerShell specific commands are called cmdlets.
- They are in the form of a **`<verb>-<noun>`**
- **`Get-Verb`** will display the standard PowerShell verbs.
- The verbs are grouped for the tasks of:
  - –Common
  - –Communication
  - –Data
  - –Diagnostic
  - –Lifecycle
  - –Other
  - –Security

Cmdlet

- **Cmdlets** are written in .Net Framework Language, most are in C#.
- **Functions** are like cmdlets but they are written in PowerShell.
- **Applications** are any type of executable that can be ran from the shell.

Cmdlet

- The **`Get-Command`** cmdlet will allow for the searching of Cmdlet, Alias and Function using wild cards.
- A recommended method for using **`Get-Command`** or its alias **`gcm`** is to use the **`-noun`** and/or **`-verb`** option so as to filter none cmdlets or use **`-CommandType cmdlet`**

Wildcard Characters

Many of the cmdlet option accept wildcards characters. In PowerShell the Wildcards Characters are:

Wildcard Character	Description	Example
*	Matches zero or more characters, starting at the specified position	a*
?	Matches any character at the specified position	?n
[<start>-<end>]	Matches a range of characters	name[1-20]
[ ]	Matches the specified characters	[ab]jhones

Cmdlet

- PowerShell supports Aliases for cmdlets. This are like shortcuts that can be used.
  - To get a full list of existing aliases in the current shell the **`Get-Alias`** cmdlet can be used.
  - They should be avoided in Scripts or Functions since they may change or be overwritten by accident.

Script Block

- In PowerShell it interprets a new line or ; as the end of a command.
- Script-block is a special structure that contains a command or a ordered collection of commands
- a Script-block is declared by using `{ <command>; command}`
- It can be passed to cmdlets or structures that accept them (More on this later)

Modules and PSSnapin

Extending the Shell

- PowerShell provides to ways to expand the number of cmdlets, functions and providers available to a user. These are:
  - **PSSnapins** - They are written in a .Net Language and are packaged as DLLs that get registered with the systems. MS Recommends to developers to not use this method anymore.
  - **Modules** - They where introduced in v2 of PowerShell and are mainly self contained in and can be copied to system to system if dependencies are included. On v3 they added the capacity for Autoloading.

Extending the Shell

- On v2 modules need to be loaded by hand to be able to see the commands it contains unless the loading of the modules is placed in the users PowerShell profile (more on them later)
- On v3 and above the commands available in modules that are located in the **$env:PSModulePath** variable can be listed and seen without loading the module explicitly and when the command is ran it autoloads the module.
- Modules can be DLL with classes that inherit from PSCmdlet class and/or PowerShell PSM1 file with functions with cmdlet attribute

Extending the Shell

- Discovering new commands from PSSnapins:
  - For all available PSSnapins **Get-PSSnapin –Registered**
  - For currently loaded PSSnapins **Get-PSSnapin**
  - For listing commands from a **loaded** PSSnapin **Get-Command -PSSnapin <PSSnapin Name>**
- Discovering new commands from Modules:
  - For listing all available modules **Get-Module –ListAvailable**
  - For Currently loaded modules **Get-Module**
  - For listing commands from a module **Get-Command -Module <module Name>** (On v2 only loaded ones)

Extending the Shell

- Loading Extensions:
  - PowerShell will load modules located on any of the folders listed in the **$env:PSModulePath** variable folder.
  - If the module is not on any of those paths then the full path would be specified with the module name to load it.
  - **Add-PsSnapin <Name>** will load a PSSnapin.
- Removing Extensions:
  - **Remove-Module <name>** to unload a module.
  - **Remove-PSSnapin <name>** to unload a PSSnapin

Extending the Shell

- Managing autoloading of modules is done by setting the $PSModuleAutoloadingPreference variable:
  - **All** - Modules are imported automatically on first-use.
  - **ModuleQualified** - Modules are imported automatically only when a user uses the module-qualified name of a command in the module **<Module Name>\<Cmdlet Name>**
  - **None** - Automatic importing of modules is disabled in the session. To import a module, use the Import-Module cmdlet.

Extending the Shell

- Name conflicts may happen when importing new commands from extensions. PowerShell will Hide or Replace commands.
- To minimize risk of this happening import new modules with either the **-NoClobber** parameter or the **-Prefix <prefix>** parameter
- One can also select what import by passing the names to the parameters -**Alias <String[]>, -Cmdlet <String[]>, -Function <String[]>, and -Variable <String[]>**

CmdletBinding

- We can make our Script and Functions have features of cmdlets compiled in C# by using the **`[cmdletbinding()]`** attribute set before the parameter **`Param()`** definition in the Script or Function.
- The help topic is `about_Functions_CmdletBinding_Attribute`

```
[CmdletBinding(ConfirmImpact=<String>,
     DefaultParameterSetName=<String>,
     HelpURI=<URI>,
     SupportsPaging=<Boolean>,
     SupportsShouldProcess=<Boolean>,
     PositionalBinding=<Boolean>)]
```

Functions

- All that we have covered for scripts applies to functions.
- To turn the script in to a function we just need to place it in the process script block of a function

```
function <name> {
    [CmdletBinding()]
    Param ($Parameter1)
    Begin{}
    Process{}
    End{}
}
```

- In the **Begin** block we put code we need to be present before executing any action.

Functions

- The **`Process`** block will execute for each object passed thru the pipeline
- The **`End`** block executes once all pipeline object are processed.
- This includes the Parameter Definitions and help based text.
- To load functions in to our current session we just define them in one of the profile files or we do what is called dot sourcing a file where the definition of a file are loaded in to the current session by placing a dot before the full path to the file

```
. .\Get-Driver.ps1
```

Using Help

Help

- To access the help system we use the **`Get-Help`** cmdlet also aliased in the shell as **`help`** and also aliased as **`man`**
- The help command can be used to get help on cmdlets and topics
- If the author included the proper comments in his/her code help can also be used against user written functions.

Updating Help

- PowerShell also provides ways to get the latest Help information.
  - The **`-online`** option will open the default web browser showing the help page for the selected cmdlet or topic.
  - On PowerShell v3 and above the **`Update-Help`** cmdlet was added and it will update the help files for PowerShell. It must be ran as Administrator.

Updating Help

- When you install PowerShell v3/v4/v5 for the first time it will come with no help files and these must be downloaded from the internet the first time help is ran.
- The language of the help files that will be downloaded will depend on the Culture setting for your shell.
  - To retrieve the culture setting the **`Get-UICulture`** cmdlet is used.
- **`Update-Help`** will use the proxy configured in Internet Explorer to connect to the internet if one is needed.

Updating Help

- In those cases that the machines are completely isolated from the internet the **Update-Help** cmdlet allows to load them from an alternate path.
  - To save the help files to an existing path

```
Save-Help -DestinationPath .\PSHelp -UICulture "en-US"
```

  - To import the help files as Administrator

```
Update-Help -UICulture "en-US" -SourcePath .\PSHelp -Force
```

- Update-Help will only update the help once very **24 hours** unless the -**Force** parameter is used.

Using Help

- Show all PowerShell conceptual topics areas. This topics cover several topics and general areas related to using PowerShell that are not related to an specific cmdlet

```
help about_*
```

- **`help <wildcard expression>`** will look for the word or expression in the titles of the help files, if none is found it will look in the content of the help for it.
- To search for all cmdlets with the word service

```
help -Category Cmdlet -Name *service*
```

Using Help

- One can select what parts of a help file we want to see.
  - Wen used against a cmdlet with no options it will show Name, Synopsis, Syntax, Description, Related Links and Remarks.
  - When the **`-Detailed`** option is given it will show Parameter Information and Examples.
  - When the **`-Full`** option is given it will show a more detailed list of info for Parameters.
  - When the **`-Examples`** option is given only examples are shown.

Using Help - Reading Syntax

- A cmdlet can have more than one way for it to be invoked and this can be seen in the syntax

```
PS C:\> help get-service

NAME
    Get-Service

SYNOPSIS
    Gets the services on a local or remote computer.


SYNTAX
    Get-Service [[-Name] <string[]>] [-ComputerName <string[]>] [-DependentServices] [-Exclude <str
    ing[]>] [-Include <string[]>] [-RequiredServices] [<CommonParameters>]

    Get-Service -DisplayName <string[]> [-ComputerName <string[]>] [-DependentServices] [-Exclude <
    string[]>] [-Include <string[]>] [-RequiredServices] [<CommonParameters>]

    Get-Service [-InputObject <ServiceController[]>] [-ComputerName <string[]>] [-DependentServices
    ] [-Exclude <string[]>] [-Include <string[]>] [-RequiredServices] [<CommonParameters>]
```

- They will typically have one or more Parameter Sets that will differ from syntax to syntax.

Using Help - Reading Syntax

- Required for required options or values they will not be enclosed in any bracket.
- Options or values enclosed in **[ ]** are optional
- Values are represent with the type they take between **< >**
- Those values that can be lists are represented as **<type[ ]>**
- Those that have a predefined list of options it can take are represented as **< option1 | option2 | option3>**

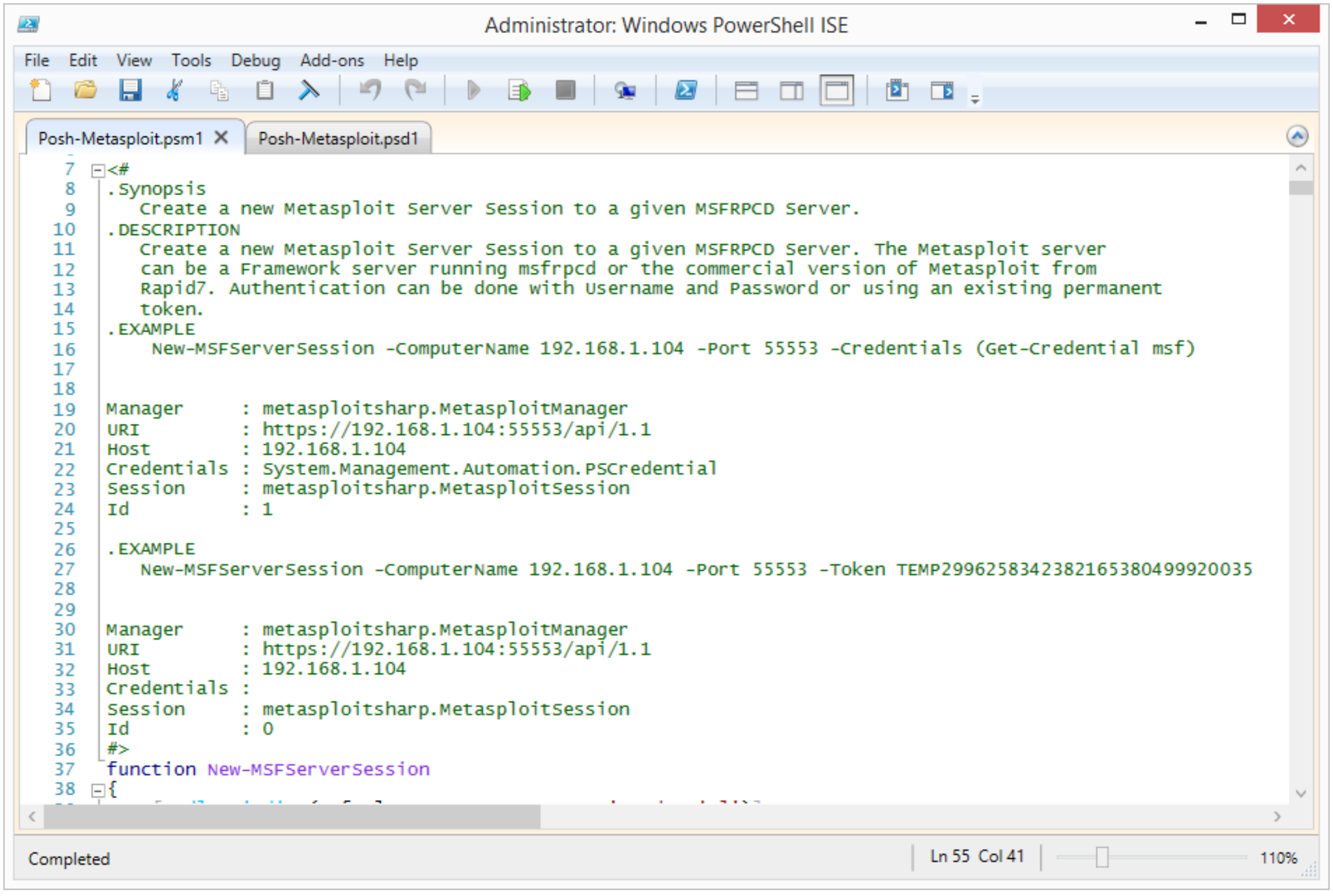
Using Help - Reading Syntax

- When the help cmdlet is used with the **`-full`** option is used we get additional information on the parameters:
  - **`required?`** - specifies if the option is required or not.
  - **`position?`** - specified if the position is a named one or an order one. For ordered one it will give the number of the position for the value it will map to it.
  - **`Default value`** - Default value the option has. (Some times on PSv2 it does not display properly)
  - **`Accept pipeline input?`** - specified if the option accepts input from the pipeline and if the input is by value type or by property name.
  - **`Accept Wildcard Characters?`** - specifies if wildcard characters can be used.

Using Help - Reading Syntax

- As parameters are defined in script and advanced function the help information is generated based on the settings by PowerShell.
- Help information on the advanced modules can be comment based or in a XML based file following the MAML (Microsoft Assistance Markup Language) Specification.
- MAML is used for binary modules, when providing help information in multiple languages and when offering updatable help.

Using Help - Comment Help



Using Help - Comment Help

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Posh-Metasploit.psm1 | Posh-Metasploit.psd1

```
37  function New-MSFServerSession
38  {
39      [CmdletBinding(DefaultParameterSetName = 'Credential')]
40      Param
41      (
42          # Metasploit Server FQDN or IP.
43          [Parameter(Mandatory=$true,
44                     Position=0)]
45          [Parameter(ParameterSetName = 'Credential')]
46          [Parameter(ParameterSetName = 'Token')]
47          [string[]]$ComputerName,
48
49          # Credentials for connecting to the Metasploit RPC Server
50          [Parameter(Mandatory=$true,
51                     Position=1,
52                     ParameterSetName = 'Credential')]
53          [Management.Automation.PSCredential]$Credentials,
54
55          # Port of the Metasploit RPC server. Use 55553 for Framework and 3790 for commercial versions.
56          [Parameter(Mandatory=$false,
57                     Position=2)]
58          [Int32]$Port = 55553,
59
60          # Version of API to use depending on target server.
61          [validateset('Pro','Framework')]
62          [string]$Version = 'Framework',
63
64          [validateset('Pro','Framework')]
65          [switch]$DisableSSL,
66
67          # Specify a existing permanent token to use.
68          [Parameter(Mandatory=$false, ParameterSetName = 'Token')]
```

Completed | Ln 1 Col 1 | 110%

Using Help - XML Help

Administrator: Windows PowerShell ISE

File Edit View Tools Debug Add-ons Help

Untitled1.ps1* | Posh-Shodan.Help.xml

```
<?xml version="1.0" encoding="utf-8"?><helpItems xmlns="http://msh" schema="maml">
<command:command xmlns:maml="http://schemas.microsoft.com/maml/2004/10" xmlns:command="http://schemas.microsoft.com/
<!--Generated by PS Cmdlet Help Editor-->
    <command:details>
        <command:name>Get-ShodanAPIInfo</command:name>
        <maml:description>
            <maml:para>Get features and information for a given API Key.</maml:para>
        </maml:description>
        <maml:copyright>
            <maml:para />
        </maml:copyright>
        <command:verb>Get</command:verb>
        <command:noun>ShodanAPIInfo</command:noun>
        <dev:version />
    </command:details>
    <maml:description>
    <!--This is the Description section-->
        <maml:para>Get features and information for a given API Key.</maml:para>
    </maml:description>
    <command:syntax>
        <command:syntaxItem>
            <maml:name>Get-ShodanAPIInfo</maml:name>
            <command:parameter required="false" variableLength="false" globbing="false" pipelineInput="false" posit
                <maml:name>APIKey</maml:name>
                <maml:description>
                    <maml:para>Shodan API Key to get information on.</maml:para>
                </maml:description>
                <command:parameterValue required="true" variableLength="false">String</command:parameterValue>
            </command:parameter>
            <command:parameter required="false" variableLength="false" globbing="false" pipelineInput="false" posit
                <maml:name>CertificateThumbprint</maml:name>
                <maml:description>
                    <maml:para>Specifies the digital public key certificate (X509) of a user account that has permi

Certificates are used in client certificate-based authentication. They can be mapped only to local user accounts; th
                </maml:description>
                <command:parameterValue required="true" variableLength="false">String</command:parameterValue>
            </command:parameter>
        </command:syntaxItem>
```

Completed | Ln 1 Col 1 | 110%

Object Basics

PowerShell Objects

- Every action taken inside of PowerShell is done in the context of objects.
- Data is moved from one cmdlet to another as a single object or collection of objects.
- Objects are composed of:
  - **Method** - Action that can be taken on the object.
  - **Property** - Information about the state of an object.
  - **Event** – An action we can monitor for.
- Even the data returned by a regular command is returned as an object.

PowerShell Objects

- To get a list of the methods and properties an object has the **`Get-Member`** cmdlet is used.
- One can use the **Pipe** to pass an object or a collection of objects to **`Get-Member`**
- If a collection is given it will return the information for each unique type in the collection.

PowerShell Objects

- Instance Methods and Properties can be accessed directly from each instance.
  - Method `<object>.<method>(Param List)`
  - Property `<object>.<property>`
- Classes also provide their own set of Methods and properties. To access these:
  - Method `[classname]::<method>(Param List)`
  - Property `[classname]::<property>`

PowerShell Objects

- For the manipulation of objects we will cover first the Operators in PowerShell since they are used against Objects and the Properties of objects.
- PowerShell operators differ from the operators of other scripting and programing languages, the design reasons where to mimic those found in Shell Languages found on *nix systems.
- When comparisons are done PowerShell has the special variables **$True** and **$False** to represent Boolean values

Boolean Operators

Operator	Description
-and	Return True if all sub-expressions are True
-or	Return True if any sub-expression is True
-not	Return the opposite
-xor	Return True if one sub-expression is True, but not if both are True

Boolean Operators

- Boolean Operators are used to combine several comparison subexpressions.
- Subexpressions can be parenthetical or cmdlets that return a boolean.

```
PS C:\> ((1 -eq 1) -or (15 -gt 20)) -and ("running" -like
"*run*")
True
```

Comparison Operators

Operator	Description
-eq	Equal to
-ne	Not Equal to
-gt	Greater than
-lt	Less than
-le	Less or Equal to
-ge	Greater or Equal to

Comparison Operators

Operator	Description
-contains -notcontains	Collection of element contains a specific element.
-in -notin	A specific element is present in a collection of elements.
-like -notlike	Wildcard string comparison
-match	Matches a regular expression

Comparison Operators

- In PowerShell comparisons are not case sensitive for string comparison

```
PS > "hello" -eq "HELLO"
True
```

- To make a comparison be case sensitive one only need to add a “c” to the comparison.

```
PS > "hello" -ceq "HELLO"
False
```

- PowerShell will try to convert the types of the element for evaluation by analyzing them.

```
PS > 1 -eq "1"
True
```

Comparison Operators

- Many times -contains and -in operators are used by mistake to search in strings, this is a common mistake. Their use is for Arrays or Hash lists

```
PS > "a","b","c" -contains "b"
True

PS > "b" -in "a","b","c"
True
```

Type Operators

Operator	Description
-is	Return True when an input is of the specified .Net type
-isnot	Return False when an input is of the specified .Net type
-as	Converts the input to a specified type

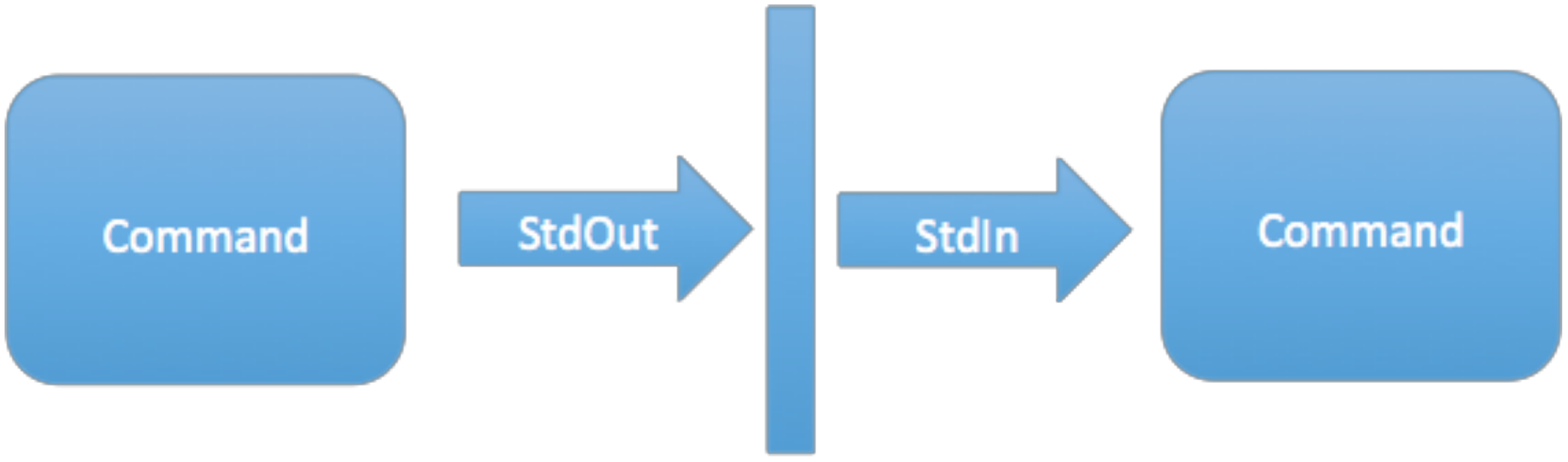
Type Operators

- Type operators are mostly used to make sure the proper type is used in scripts

```
C:\PS> (get-date) -is [datetime]
True

C:\PS> (get-date) -isnot [datetime]
False

C:\PS> "9/28/12" -as [datetime]
Friday, September 28, 2012 12:00:00 AM
```

Pipeline

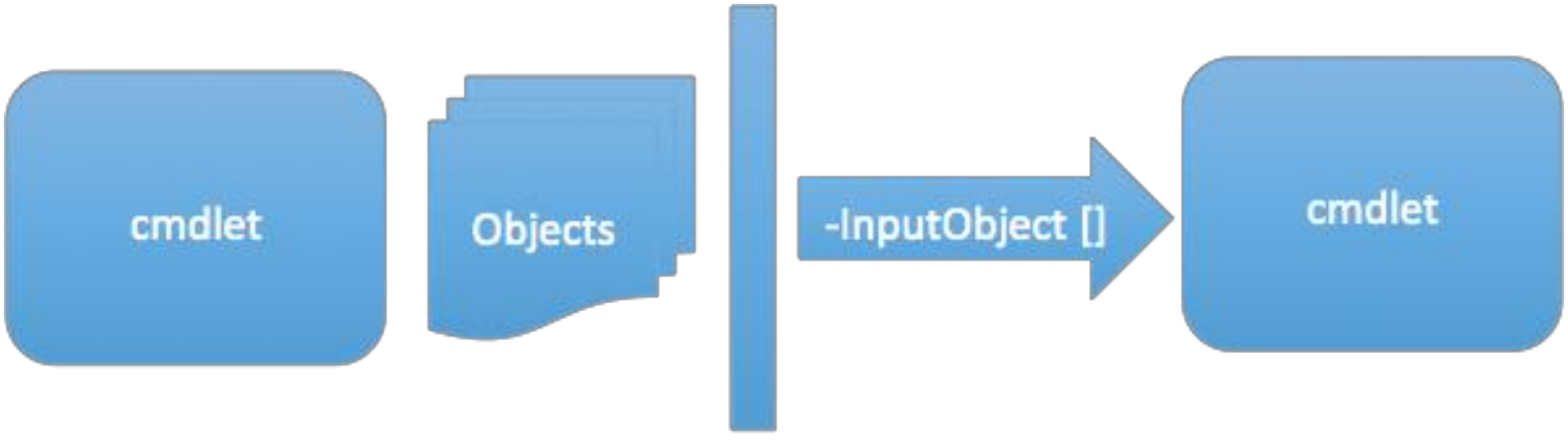
Pipeline On Other Shells



The Pipeline

- The pipeline is what makes PowerShell so powerful as a shell.
- It ties commands and cmlets together in ways a regular shell can not.
- Mastery of the Pipeline is what makes the difference in mastering or not mastering PowerShell

Pipeline ByValue



Pipeline ByValue

- A cmdlet's receiving Object Type must be same as the Object Type output by the originating cmdlet.
- The Parameter must accept input from the pipeline and it must also accept a collection (Array of Objects)

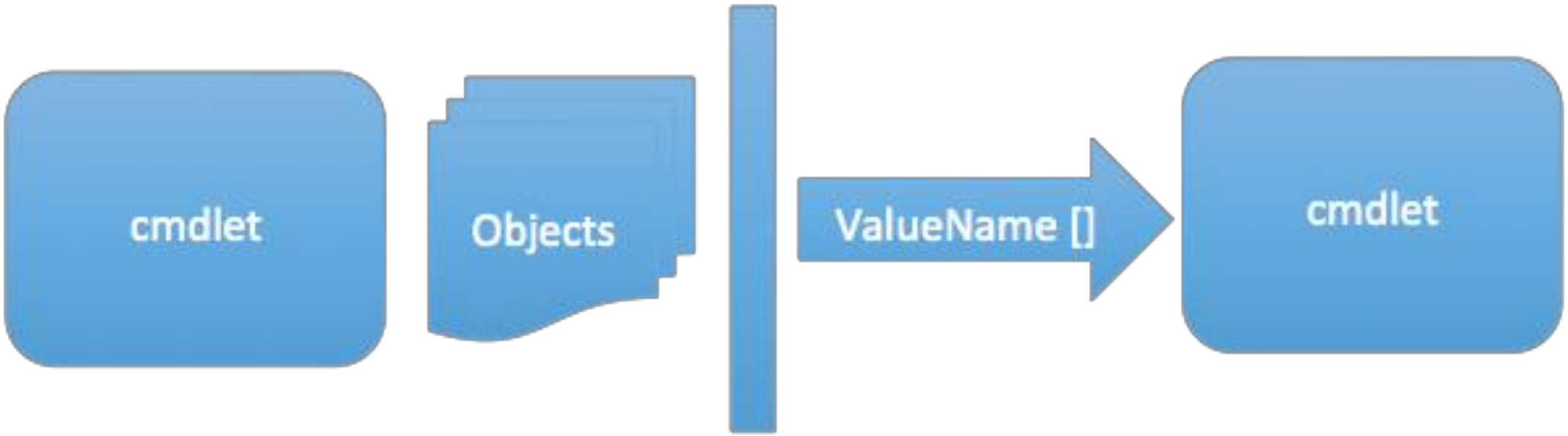
```
-InputObject <ServiceController[]>
    Specifies ServiceController objects representing the services to be stopped. Enter a variable that contains
    the objects, or type a command or expression that gets the objects.

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

Pipeline ByValue

- In advanced functions withe the cmdlet binding a parameter can be made to accept objects by value in its parameter definition.

```
[Parameter(Mandatory=$true,
           ValueFromPipeline=$true,
           Position=0)]
[sting[]]
$Param
```

Pipeline ByPropertyName



Pipeline ByPropertyName

- The Object has to have a property which name matches the Parameter name
- Te Parameter must accept input from the pipeline and it must also accept a collection

```
-Name <String[]>
    Specifies the service names of the services to be stopped. Wildcards are permitted.

    The parameter name is optional. You can use "Name" or its alias, "ServiceName", or you can omit the parameter
    name.

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

Pipeline

- When an object collection is sent thru the pipeline to another cmdlet that takes a collection each object is referred to as **$_** giving us access to the object properties, methods and events.

```
Get-Service | where-object { $_.Status -eq "Running" }
```

Working with Objects

Filtering Objects

- For filtering objects in a collection in PowerShell the **`Where-Object`** cmdlet is used since it allows to filter by property value.
- On PowerShell v2 this is done with a Script Block

```
Get-Service | where-object { $_.Status -eq "Running" }
```

- On PowerShell v3 this can be done with a Script Block or by Specifying the property and value as parameters.

```
Get-Service | Where-Object -Property Status -eq -Value Running
```

```
Get-Service | Where-Object Status -eq Running
```

Selecting Objects

- The **`Select-Object`** cmdlet allows for:
  - Selecting specific objects or a Range of objects from an ordered list objects that contains specific properties.
  - Selecting a given number from the beginning or end of a ordered list of objects.
  - Select specific properties from objects.
  - Creation of new object with properties
  - Renaming object properties

Selecting Objects

- Selecting specific Objects from a list

```
PS > Get-Process |Sort-Object workingset -Descending |Select-Object -Index 0,1,2,3,4
```

- Selecting a range of objects from a list

```
PS > Get-Process | Sort-Object workingset -Descending | Select-Object -Index (0..4)
```

- Select the first 5 from a list

```
PS > Get-Process | Sort-Object workingset -Descending | Select-Object -first 5
```

- Creating/Renaming a property

```
PS > Get-Process |  Select-Object -Property name,@{name = 'PID'; expression = {$_.id}}
```

Iterating Objects

- Iteration is the method by which several objects in a collection are processed one by one and actions are taken against them.
- In PowerShell, there are 2 methods for iterating thru objects and are often confused:
  - **`ForEach-Object`** cmdlet and its aliases **`foreach`** and **`%`**.
  - **`foreach(<variable> in <collection>){}`** statement.
- Each method will take a collection and process the objects in a Scriptblock but each behaves differently, however and it's use will vary case by case.

Iterating Objects

- The ScriptBlocks parameters are also positional

```
PS C:\> 1..5 | ForEach-Object { $Sum = 0 } { $Sum += $_ } {
$Sum }

15
```

- To skip to the next object to be process in **ForEach-Object** the keyword **return** is used.
- For exiting the loop inside of a **ForEach-Object** the **break** keyword is used.
- For Next value in the loop inside of a **ForEach-Object** the **return** keyword is used inside the comparing script block.

Iterating Objects

- The **`foreach`(<variable>** **`in`** **<collection>){}** statement places on each iteration an element of a collection in to memory first and then processes each. (Not good for very large collections)
- Since the collection being worked on is loaded in to memory it tends to be faster than the ForEach-Object cmdlet.
- To skip to the next object to be process in **foreach statement** the keyword **`continue`** is used.
- For exiting the loop inside of a **foreach statement** the **`break`** keyword is used.

Iterating Objects

- The foreach statement can be used in the shell as well as in scripts

```
 PS >foreach ($i in (1..10)){
>>      if ($i -gt 5){
>>          continue
>>      }
>>      $i
>> }
>>

1
2
3
4
5
```

Measuring Objects

- For performing metrics on objects the **`Measure-Object`** cmdlet is used.
- It can:
  - Count objects
  - Calculate the minimum, maximum, sum, and average of the numeric values.
  - For text objects, it can count and calculate the number of lines, words, and characters.

Measuring Objects

- Measuring object count

```
PS C:\> Get-Service | Measure-Object
```

- Measuring text

```
PS > Get-Content C:\Windows\WindowsUpdate.log | Measure-Object -
Line -Word -Character | Format-Table -AutoSize
```

- Measuring a numeric property

```
 PS C:\> get-process | measure-object -property workingset -
minimum -maximum -average
```

Grouping Objects

- PowerShell allows the grouping of object based on a key property using the **`Group-Object`** cmdlet

```
PS > Get-Service | Group-Object status

Count Name                      Group
----- ----                      -----
  103 Stopped                   {AeLookupSvc, ALG...}
   55 Running                   {AudioSrv, BFE, BITS...}
```

- For each group, it return an object called `Microsoft.PowerShell.Commands.GroupInfo` with properties:
  - Count
  - Group
  - Name
  - Values

Formatting

- Formatting is how the information from the Objects returned from cmdlets are shown on the screen.
- When PowerShell default to showing all set properties for an object unless a **format.pmxml** files in the PowerShell object being displayed exists in the installation folder.
- If there is no formatting predefined if 4 or less properties exist it will use a table and if it is 5 or more a list.

Formatting

- The formatting cmdlets are:
  - **`Format-Custom`** Used for the testing and creation of custom format files (not covered in this class).
  - **`Format-Table`** formats the output of a command as a table of properties in which each property is displayed on a separate column.
  - **`Format-List`** formats the output of a command as a list of properties in which each property is displayed on a separate line.
  - **`Format-Wide`** formats the objects in a screen wide table showing only one property of the object.

Formatting

- **`Format-Table`** will try to fill the whole with of the screen.
- Great for showing a selected number of properties in a format that uses the most of the screen.
- Properties can be selected with the **`-property`** parameter.
- If the content of the property is longer than the column with it will be truncated by default.
- Column titles can be changed with custom expression for the property.
  `@{name=”name”;Expression={<expression>}}`

Formatting

- **`Format-List`** will only show the default properties for the view configured.
- **`Format-List`** is a recommended cmdlet for listing all properties and their values for discovery and troubleshooting.

Formatting

- When a format cmdlet is used no object is returned, only the formatted text so no cmdlet that expects objects should be used at the extreme right of the command.
- The only cmdlets that can use the formatted output are:
  - **Out-File** - Saves output shown on screen to a file.
  - **Out-Printer** - Send output shown on screen to a printer.
  - **Out-Host** - Sends output to the stdout of the session.