# GitHub - PowerShellMafia/PowerSploit: PowerSploit - A PowerShell Post-Exploitation Framework

By HarmJ0y

Archived: 2026-04-05 17:47:59 UTC

## This project is no longer supported

**PowerSploit is a collection of Microsoft PowerShell modules that can be used to aid penetration testers during all phases of an assessment. PowerSploit is comprised of the following modules and scripts:**

## CodeExecution

**Execute code on a target machine.**

`Invoke-DllInjection`

Injects a Dll into the process ID of your choosing.

`Invoke-ReflectivePEInjection`

Reflectively loads a Windows PE file (DLL/EXE) in to the powershell process, or reflectively injects a DLL in to a remote process.

`Invoke-Shellcode`

Injects shellcode into the process ID of your choosing or within PowerShell locally.

`Invoke-WmiCommand`

Executes a PowerShell ScriptBlock on a target computer and returns its formatted output using WMI as a C2 channel.

## ScriptModification

**Modify and/or prepare scripts for execution on a compromised machine.**

`Out-EncodedCommand`

Compresses, Base-64 encodes, and generates command-line output for a PowerShell payload script.

`Out-CompressedDll`

Compresses, Base-64 encodes, and outputs generated code to load a managed dll in memory.

`Out-EncryptedScript`

Encrypts text files/scripts.

`Remove-Comment`

Strips comments and extra whitespace from a script.

## Persistence

**Add persistence capabilities to a PowerShell script**

`New-UserPersistenceOption`

Configure user-level persistence options for the Add-Persistence function.

`New-ElevatedPersistenceOption`

Configure elevated persistence options for the Add-Persistence function.

`Add-Persistence`

Add persistence capabilities to a script.

`Install-SSP`

Installs a security support provider (SSP) dll.

`Get-SecurityPackages`

Enumerates all loaded security packages (SSPs).

## AntivirusBypass

**AV doesn't stand a chance against PowerShell!**

`Find-AVSignature`

Locates single Byte AV signatures utilizing the same method as DSplit from "class101".

## Exfiltration

**All your data belong to me!**

`Invoke-TokenManipulation`

Lists available logon tokens. Creates processes with other users logon tokens, and impersonates logon tokens in the current thread.

`Invoke-CredentialInjection`

Create logons with clear-text credentials without triggering a suspicious Event ID 4648 (Explicit Credential Logon).

`Invoke-NinjaCopy`

Copies a file from an NTFS partitioned volume by reading the raw volume and parsing the NTFS structures.

`Invoke-Mimikatz`

Reflectively loads Mimikatz 2.0 in memory using PowerShell. Can be used to dump credentials without writing anything to disk. Can be used for any functionality provided with Mimikatz.

`Get-Keystrokes`

Logs keys pressed, time and the active window.

`Get-GPPPassword`

Retrieves the plaintext password and other information for accounts pushed through Group Policy Preferences.

`Get-GPPAutologon`

Retrieves autologon username and password from registry.xml if pushed through Group Policy Preferences.

`Get-TimedScreenshot`

A function that takes screenshots at a regular interval and saves them to a folder.

`New-VolumeShadowCopy`

Creates a new volume shadow copy.

`Get-VolumeShadowCopy`

Lists the device paths of all local volume shadow copies.

`Mount-VolumeShadowCopy`

Mounts a volume shadow copy.

`Remove-VolumeShadowCopy`

Deletes a volume shadow copy.

`Get-VaultCredential`

Displays Windows vault credential objects including cleartext web credentials.

`Out-Minidump`

Generates a full-memory minidump of a process.

`Get-MicrophoneAudio`

Records audio from system microphone and saves to disk

## Mayhem

**Cause general mayhem with PowerShell.**

`Set-MasterBootRecord`

Proof of concept code that overwrites the master boot record with the message of your choice.

`Set-CriticalProcess`

Causes your machine to blue screen upon exiting PowerShell.

## Privesc

**Tools to help with escalating privileges on a target.**

`PowerUp`

Clearing house of common privilege escalation checks, along with some weaponization vectors.

## Recon

**Tools to aid in the reconnaissance phase of a penetration test.**

`Invoke-Portscan`

Does a simple port scan using regular sockets, based (pretty) loosely on nmap.

`Get-HttpStatus`

Returns the HTTP Status Codes and full URL for specified paths when provided with a dictionary file.

`Invoke-ReverseDnsLookup`

Scans an IP address range for DNS PTR records.

`PowerView`

PowerView is series of functions that performs network and Windows domain enumeration and exploitation.

## Recon\Dictionaries

**A collection of dictionaries used to aid in the reconnaissance phase of a penetration test. Dictionaries were taken from the following sources.**

- admin.txt - http://cirt.net/nikto2/
- generic.txt - http://sourceforge.net/projects/yokoso/files/yokoso-0.1/
- sharepoint.txt - http://www.stachliu.com/resources/tools/sharepoint-hacking-diggity-project/

## License

The PowerSploit project and all individual scripts are under the BSD 3-Clause license unless explicitly noted otherwise.

## Usage

Refer to the comment-based help in each individual script for detailed usage information.

To install this module, drop the entire PowerSploit folder into one of your module directories. The default PowerShell module paths are listed in the $Env:PSModulePath environment variable.

The default per-user module path is: "$Env:HomeDrive$Env:HOMEPATH\Documents\WindowsPowerShell\Modules" The default computer-level module path is: "$Env:windir\System32\WindowsPowerShell\v1.0\Modules"

To use the module, type `Import-Module PowerSploit`

To see the commands imported, type `Get-Command -Module PowerSploit`

If you're running PowerShell v3 and you want to remove the annoying 'Do you really want to run scripts downloaded from the Internet' warning, once you've placed PowerSploit into your module path, run the following one-liner: `$Env:PSModulePath.Split(';') | % { if ( Test-Path (Join-Path $_ PowerSploit) ) {Get-ChildItem $_ -Recurse | Unblock-File} }`

For help on each individual command, Get-Help is your friend.

Note: The tools contained within this module were all designed such that they can be run individually. Including them in a module simply lends itself to increased portability.

## Contribution Rules

We need contributions! If you have a great idea for PowerSploit, we'd love to add it. New additions will require the following:

- The script must adhere to the style guide. Any exceptions to the guide line would need an explicit, valid reason.
- The module manifest needs to be updated to reflect the new function being added.
- A brief description of the function should be added to this README.md
- Pester tests must accompany all new functions. See the Tests folder for examples but we are looking for tests that at least cover the basics by testing for expected/unexpected input/output and that the function exhibits desired functionality. Make sure the function is passing all tests (preferably in mutiple OSes) prior to submitting a pull request. Thanks!

## Script Style Guide

**For all contributors and future contributors to PowerSploit, I ask that you follow this style guide when writing your scripts/modules.**

- Avoid Write-Host **at all costs**. PowerShell functions/cmdlets are not command-line utilities! Pull requests containing code that uses Write-Host will not be considered. You should output custom objects instead. For more information on creating custom objects, read these articles:
    - http://blogs.technet.com/b/heyscriptingguy/archive/2011/05/19/create-custom-objects-in-your-powershell-script.aspx
    - http://technet.microsoft.com/en-us/library/ff730946.aspx
- If you want to display relevant debugging information to the screen, use Write-Verbose. The user can always just tack on '-Verbose'.
- Always provide descriptive, comment-based help for every script. Also, be sure to include your name and a BSD 3-Clause license (unless there are extenuating circumstances that prevent the application of the BSD license).
- Make sure all functions follow the proper PowerShell verb-noun agreement. Use Get-Verb to list the default verbs used by PowerShell. Exceptions to supported verbs will be considered on a case-by-case basis.
- I prefer that variable names be capitalized and be as descriptive as possible.
- Provide logical spacing in between your code. Indent your code to make it more readable.
- If you find yourself repeating code, write a function.

- Catch all anticipated errors and provide meaningful output. If you have an error that should stop execution of the script, use 'Throw'. If you have an error that doesn't need to stop execution, use Write-Error.
- If you are writing a script that interfaces with the Win32 API, try to avoid compiling C# inline with Add-Type. Try to use the PSReflect module, if possible.
- Do not use hardcoded paths. A script should be useable right out of the box. No one should have to modify the code unless they want to.
- PowerShell v2 compatibility is highly desired.
- Use positional parameters and make parameters mandatory when it makes sense to do so. For example, I'm looking for something like the following:
    - `[Parameter(Position = 0, Mandatory = $True)]`
- Don't use any aliases unless it makes sense for receiving pipeline input. They make code more difficult to read for people who are unfamiliar with a particular alias.
- Try not to let commands run on for too long. For example, a pipeline is a natural place for a line break.
- Don't go overboard with inline comments. Only use them when certain aspects of the code might be confusing to a reader.
- Rather than using Out-Null to suppress unwanted/irrelevant output, save the unwanted output to $null. Doing so provides a slight performance enhancement.
- Use default values for your parameters when it makes sense. Ideally, you want a script that will work without requiring any parameters.
- If a script creates complex custom objects, include a ps1xml file that will properly format the object's output.

---