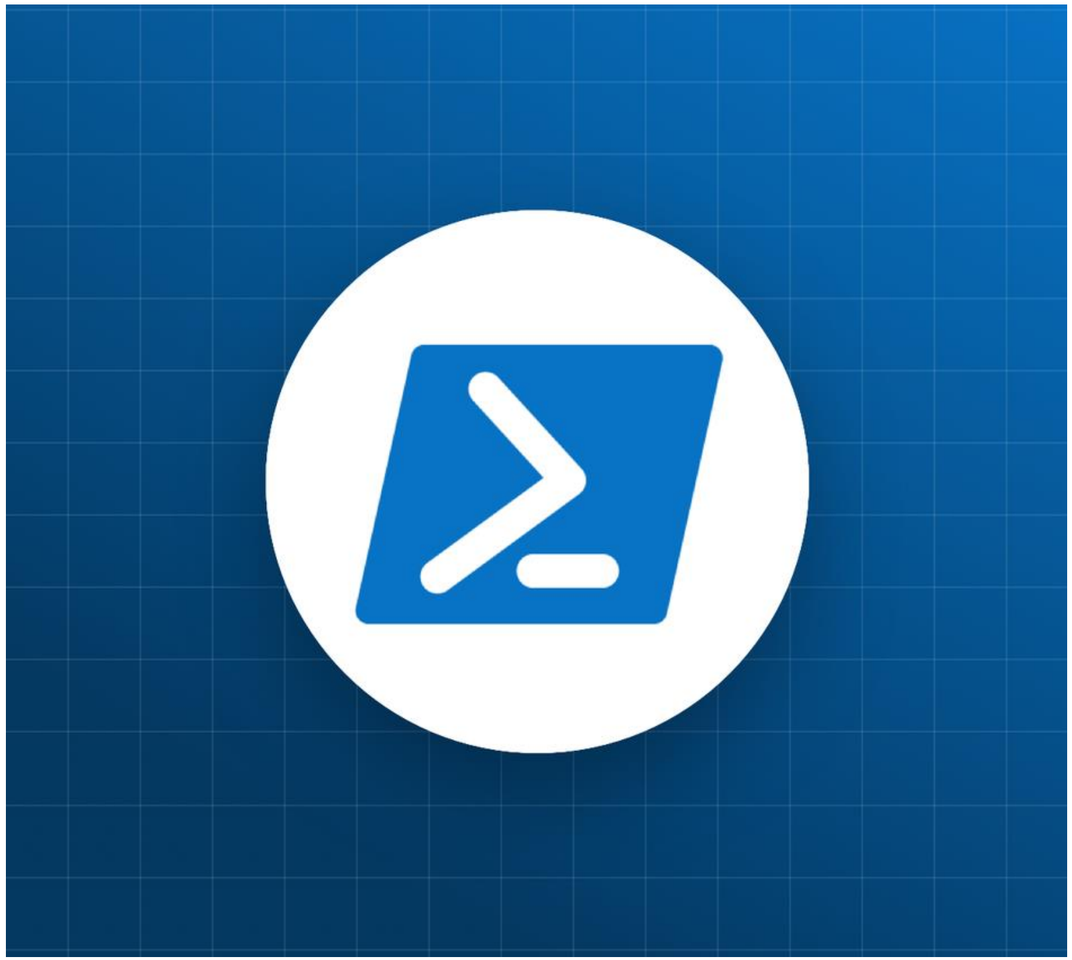
Detecting Offensive PowerShell Tools



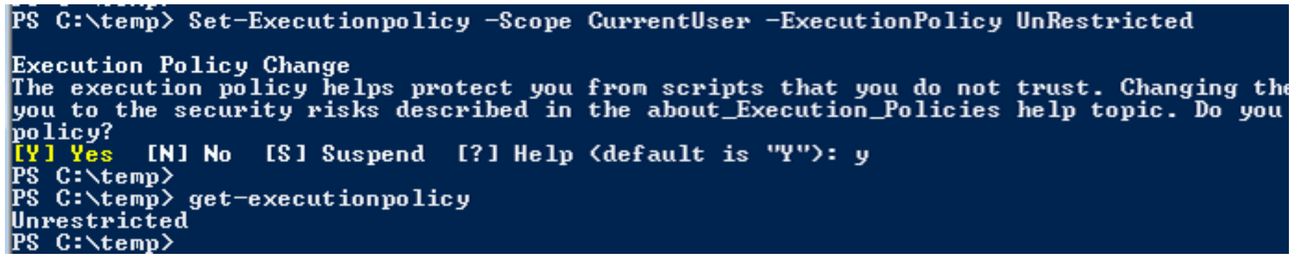
Why do attackers love PowerShell?

- It is a built-in command line tool
- It provides unprecedented access on Windows computers
- It’s enabled on most computers, as system administrators use PowerShell to automate various tasks
- Its malicious use is often not stopped or detected by traditional endpoint defenses, as files and commands are not written to disk

It is hard to block access to PowerShell

The reality is that PowerShell is more than a single executable. PowerShell exists in the **System.Management.Automation.dll** dynamic linked library file (DLL) and can host different runspaces which are effectively PowerShell instances.

PowerShell Execution Policies aren’t about security



The PowerShell Execution Policy is set to restricted by default so .PS1 files don’t auto-execute.This means all script execution is disabled by default, though one can still type in the commands by hand. Bypassing the PowerShell Execution Policy is as easy as asking.

PowerShell.exe can be instantiated with no execution policy to run the script of choice.

PowerShell as an attack platform

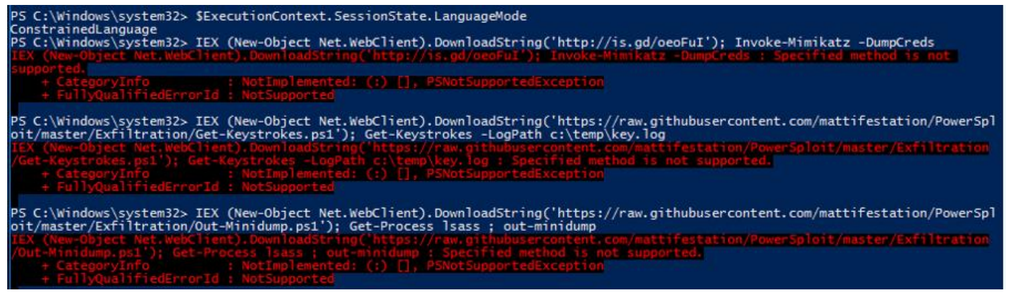
- Run code in memory without touching disk
- Download & execute code from another system
- Interface with .Net & Windows APIs
- Most organizations are not watching PowerShell activity
- CMD.exe is commonly blocked, though not PowerShell
- By default, PowerShell does not leave many artifacts of its execution in most Windows environments. The combination of impressive functionality and stealth has made attacks leveraging PowerShell a nightmare for enterprise security teams

PowerShell attack code can be invoked by

- Microsoft Office Macro (VBA)
- WMI
- HTA Script (HTML Application – control panel extensions)
- CHM (compiled HTML help)
- Java JAR file
- Other script type (VBS/WSH/BAT/CMD)
- Typically an Encoded Command

Limiting PowerShell Attack Capability with Constrained Language Mode

Constrained language mode limits the capability of PowerShell to base functionality removing advanced feature support such as .NET & Windows API calls and COM access.The lack of this advanced functionality stops most PowerShell attack tools since they rely on these methods.

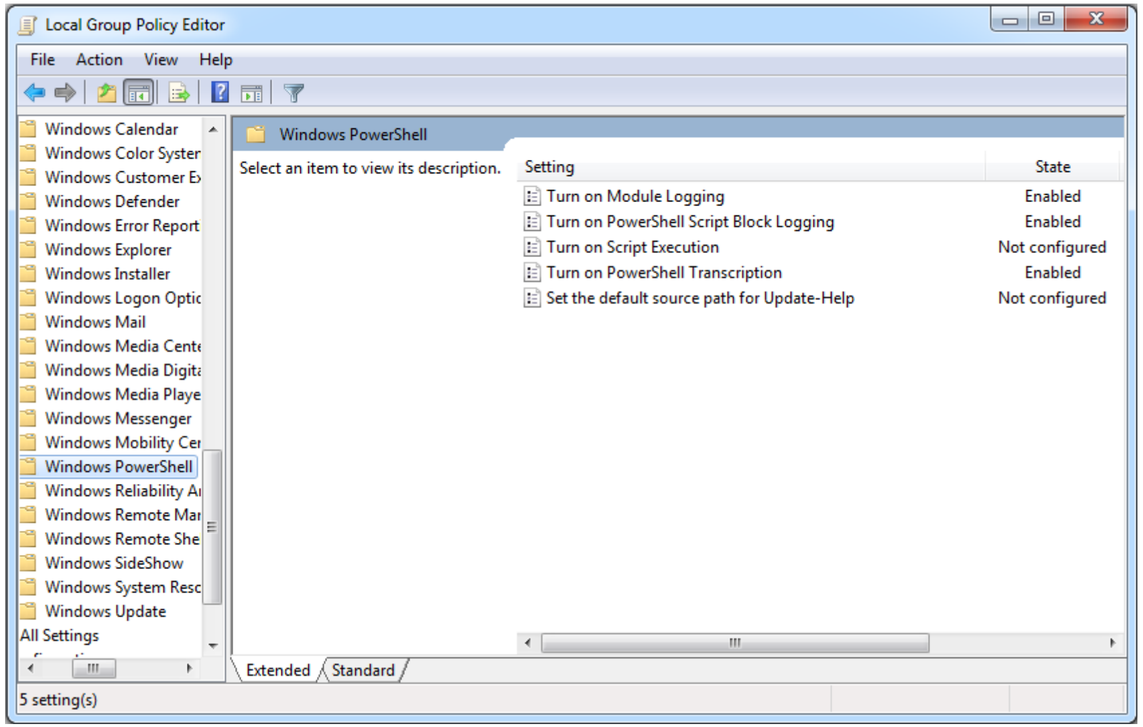


The drawback to this approach is that in order to configured PowerShell to run in constrained mode, an environment variable must be set, either by running a command in PowerShell or via Group Policy.

Enable Constrained Language Mode:
[Environment]::SetEnvironmentVariable(‘__PSLockdownPolicy‘, ‘4’, ‘Machine‘)

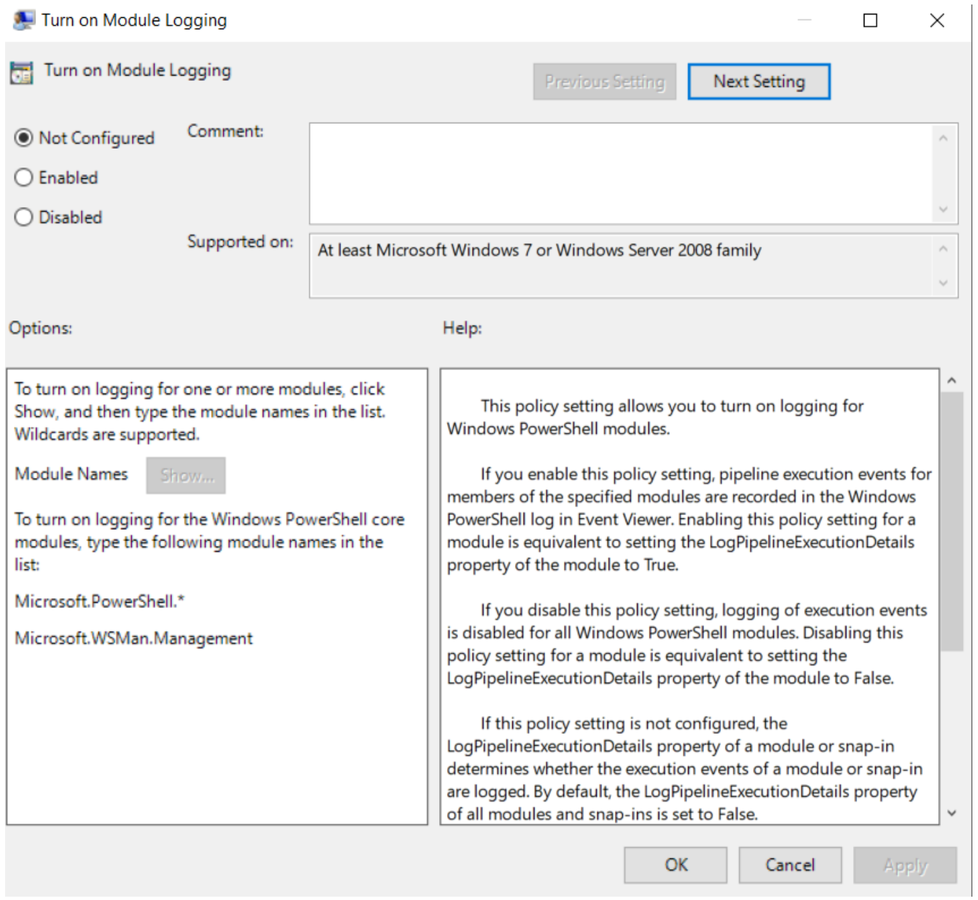
Enable via Group Policy:
Computer Configuration\Preferences\Windows Settings\Environment

Keep in mind that bypassing Constrained PowerShell is possible and not all PowerShell “attack scripts” will be blocked. This environment variable can be modified by an attacker once they have gained control of the system. they would have to spawn a new PowerShell instance to run code in full language mode after changing the environment. These changes would be logged and could help the defender in identifying unusual activity on the system.



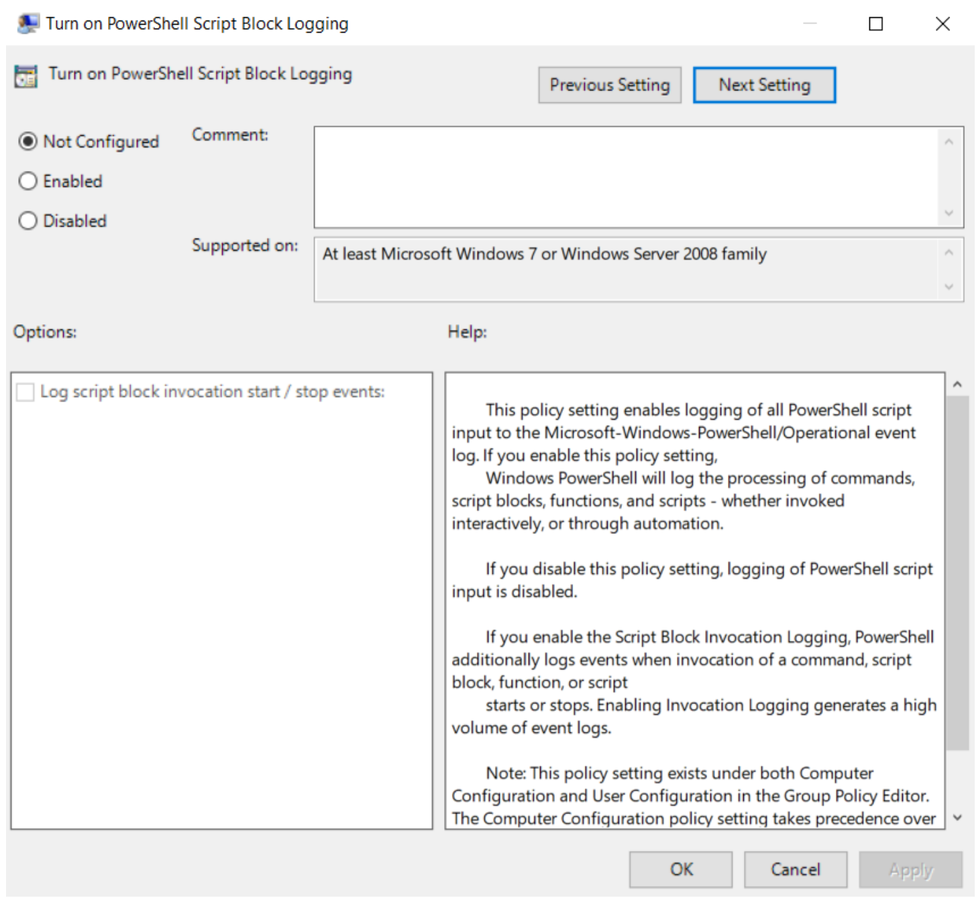
PowerShell Logging

- Logging must be configured through Group Policy as follows:
  - Administrative Templates → Windows Components → Windows PowerShell
- PowerShell supports three types of logging:
  - Module Logging
  - script block logging
  - transcription
- PowerShell events are written to :
  - Microsoft-Windows-PowerShell%4Operational.evtx



Module Logging

- Module logging records pipeline execution details as PowerShell executes, including variable initialization and command invocations.
- Module logging will record portions of scripts, some de-obfuscated code, and some data formatted for output.
- This logging will capture some details missed by other PowerShell logging sources, though it may not reliably capture the commands executed.
- Module logging events are written to Event ID (EID) 4103.



Script Block Logging

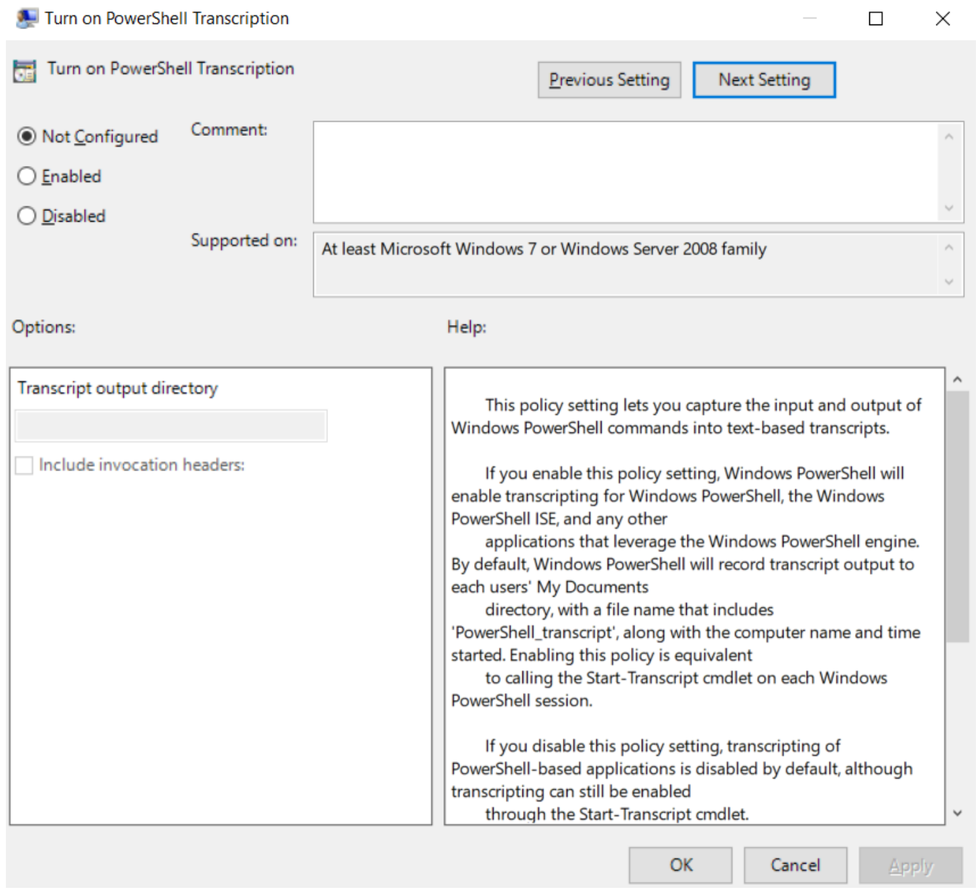
- Script block logging records blocks of code as they are executed by the PowerShell engine, thereby capturing the full contents of code executed by an attacker, including scripts and commands.

- Due to the nature of script block logging, it also records de-obfuscated code as it is executed. For example, in addition to recording the original obfuscated code, script block logging records the decoded commands passed with PowerShell's -EncodedCommand argument, as well as those obfuscated with XOR, Base64, ROT13, encryption, etc.

- Script block logging events are recorded in EID 4104.

- PowerShell 5.0 will automatically log code blocks if the block's contents match on a list of suspicious commands or scripting techniques, even if script block logging is not enabled. These suspicious blocks are logged at the "warning" level in EID 4104, unless script block logging is explicitly disabled.

- This allows investigators to identify the full scope of attacker activity. The blocks that are not considered suspicious will also be logged to EID 4104, but with "verbose" or "information" levels.



Transcription

- Transcription creates a unique record of every PowerShell session, including all input and output, exactly as it appears in the session.
- transcription records only what appears in the PowerShell terminal, which will not include the contents of executed scripts or output written to other destinations such as the file system.
- PowerShell transcripts are automatically named to prevent collisions, with names beginning with “PowerShell_transcript”. By default, transcripts are written to the user’s documents folder, but can be configured to any accessible location on the local system or on the network.

Detecting offensive PowerShell tools

- The best method to detect PowerShell attack code is to look for key indicators – code snippets required for the code to run correctly

- Invoke-Mimikatz Event Log Keywords:
  - “System.Reflection.AssemblyName”
  - “System.Reflection.Emit.AssemblyBuilderAccess “
  - “System.Runtime.InteropServices.MarshalAsAttribute”
  - “TOKEN_PRIVILEGES”
  - “SE_PRIVILEGE_ENABLED“

- Invoke-TokenManipulation Event Log Keywords:
  - “TOKEN_IMPERSONATE”
  - “TOKEN_DUPLICATE”
  - “TOKEN_ADJUST_PRIVILEGES”

- Invoke-CredentialInjection: Event Log Keywords:
  - “TOKEN_PRIVILEGES”
  - “GetDelegateForFunctionPointer”

- Invoke-DLLInjection Event Log Keywords:
  - “System.Reflection.AssemblyName“
  - “System.Reflection.Emit.AssemblyBuilderAccess“

- Invoke-Shellcode Event Log Keywords:
  - “System.Reflection.AssemblyName“
  - “System.Reflection.Emit.AssemblyBuilderAccess”
  - “System.MulticastDelegate”
  - “System.Reflection.CallingConventions”

- Invoke-TokenManipulation Event Log Keywords:
  - “TOKEN_IMPERSONATE”
  - “TOKEN_DUPLICATE”
  - “TOKEN_ADJUST_PRIVILEGES”

- Get-GPPPassword Event Log Keywords:
  - “System.Security.Cryptography.AesCryptoServiceProvider”
  - “0x4e,0x99,0x06,0xe8,0xfc,0xb6,0x6c,0xc9,0xfa,0xf4”
  - “Groups.User.Properties.cpassword”
  - “ScheduledTasks.Task.Properties.cpassword”

- Out-MiniDump Event Log Keywords:
  - “System.Management.Automation.WindowsErrorReporting”
  - “MiniDumpWriteDump”