EMOTET: a State-Machine reversing exercise

cecio

cecio/**EMOTET-2020-Reversing**

a State-Machine reversing exercise

1 Contributor 0 Issues 7 Stars 2 Forks

Intro

Around the 20th of December 2020, there was one of the "usual" EMOTET email campaign hitting several countries. I had the possibility to get some sample and I decided to make this little analysis, to deep dive some specific aspects of the malware itself.

In particular I had a look to how the malware has been written, with an analysis of the interesting techniques used.

There is a very good analysis done by Fortinet in 2019, where the also the first stage has been analyzed. My exercise is more focused on the second stage on a recent sample.

In this repository you will find all the DLLs, scripts and tools used for the analysis, with the **annotated Ghidra project file,** with all the mapping to my findings (API calls, program logic, etc). You can use this as starting point for additional investigation on it. Enjoy ;-)

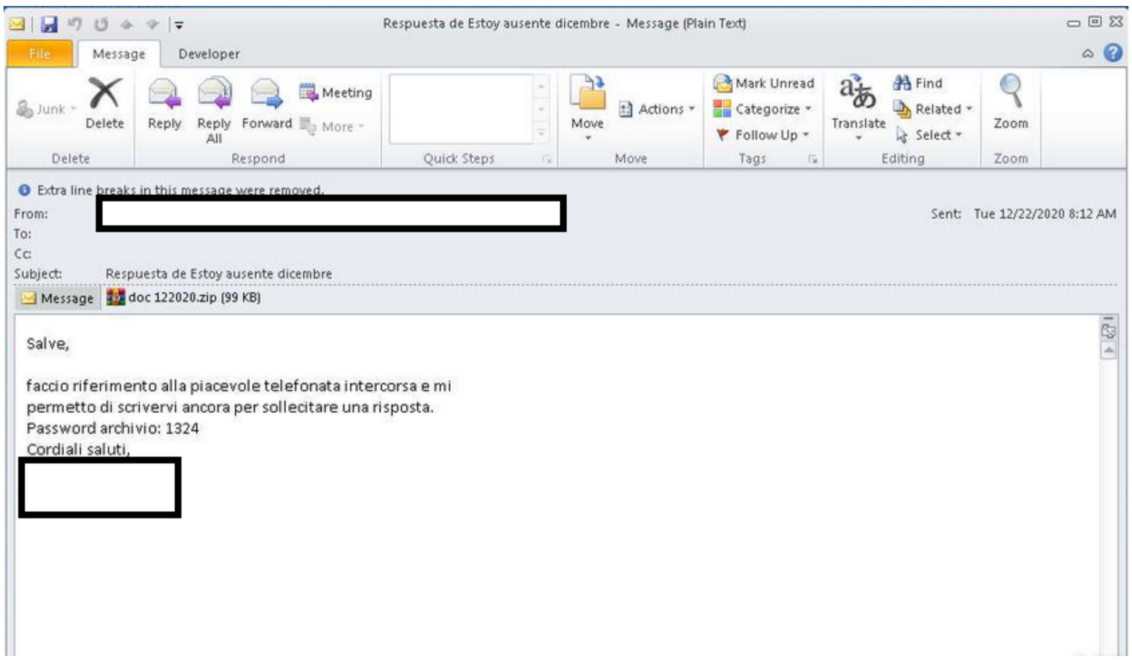
The Tools

- FireEye Speakeasy

- [Ghidra]
- [x64dbg]
- [PE Bear]
- time :-)

The infection chain

EMOTET is usually spread by using e-mail campaign (in this case in Italian language)



This particular sample is coming from what we can call the usual infection chain:

1. delivery of an e-mail with a malicious zipped document
2. once opened, the document runs an obfuscated powershell script and downloads the 2nd stage
3. the 2nd stage (in form of a DLL) is then executed
4. the 2nd stage establish some persistence and try to connect a C2

The initial triage

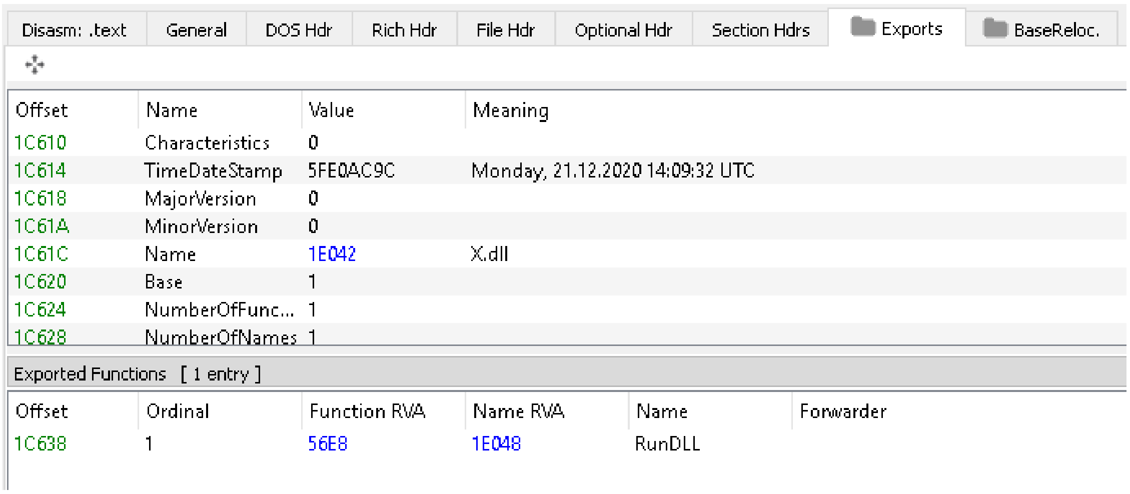
All the files used for this analysis are in the repository. The "dangerous" ones are password protected (with the usual pwd).

The DLL (`sg.dll`) has the following characteristics:

```
File Name: sg.dll
Size:      340480
SHA1:      b08e07b1d91f8724381e765d695601ea785d8276
```

This DLL exports a single function named `RunDLL` : once executed, it decrypts "in-memory" an additional DLL. This one, dumped as `dump_1_0418.bin` , is the target of my analysis:

```
File Name:  dump_1_0418.bin
Size:       122880
SHA1:       57cd8eac09714effa7b6f70b34039bbace4a3e23
```



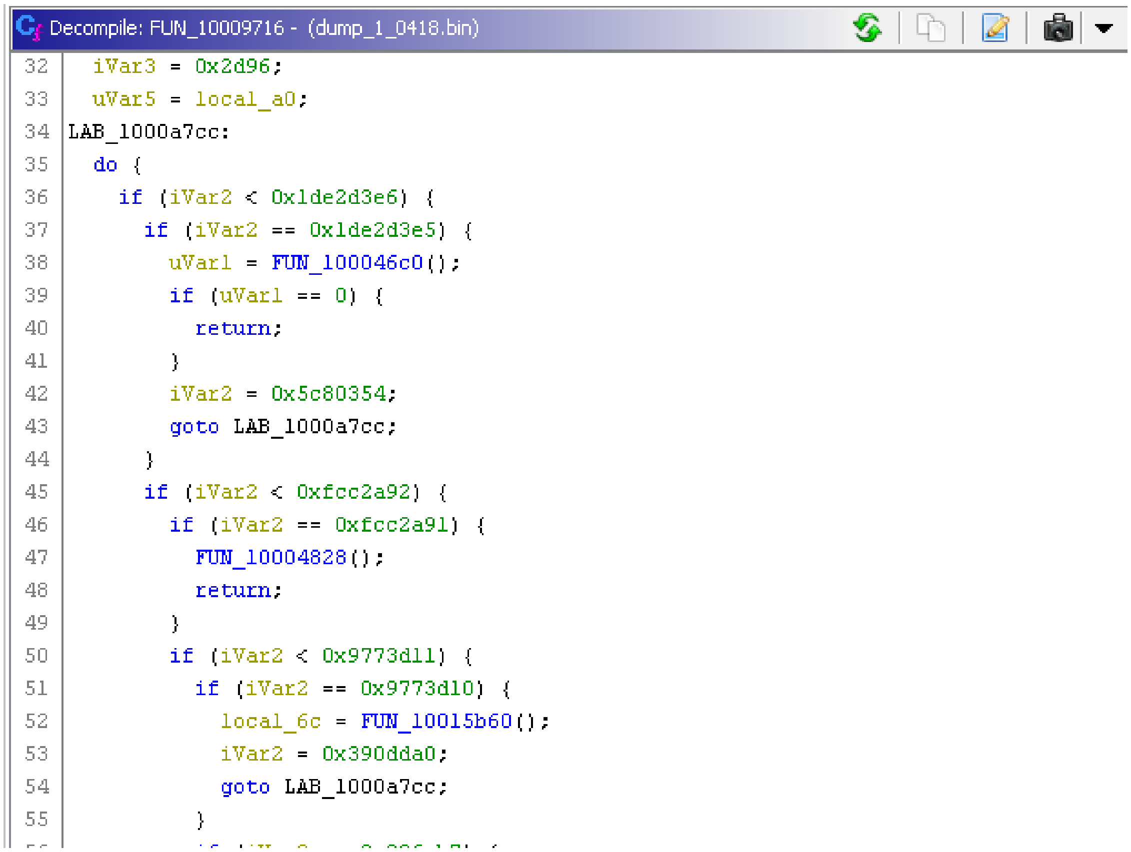
Offset	Name	Value	Meaning
1C610	Characteristics	0	
1C614	TimeDateStamp	5FE0AC9C	Monday, 21.12.2020 14:09:32 UTC
1C618	MajorVersion	0	
1C61A	MinorVersion	0	
1C61C	Name	1E042	X.dll
1C620	Base	1	
1C624	NumberOfFunc...	1	
1C628	NumberOfNames	1	

Exported Functions [1 entry]

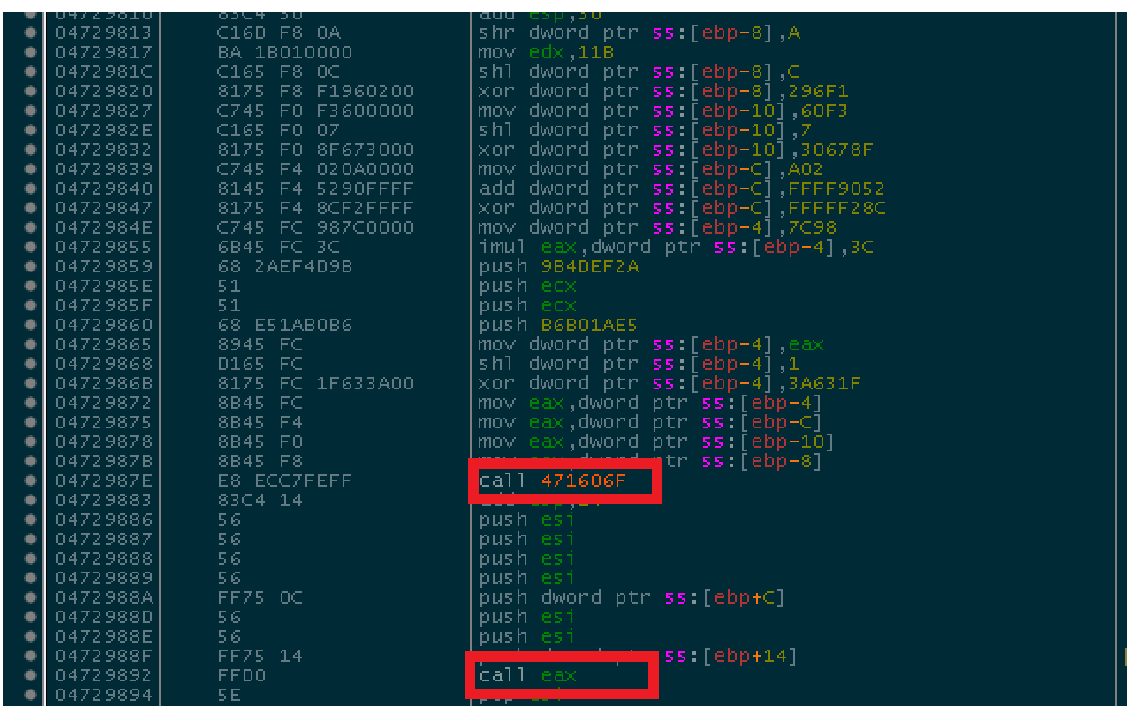
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
1C638	1	56E8	1E048	RunDLL	

An initial overview of the dumped DLL, shows immediately that we don't have any string visible in it, no imports and a first look to the disassembly shows a heavily obfuscated code. We need to do some work here.

I fired up **Ghidra** and started to snoop around. Starting from the only exported function `RunDLL` you quickly end up to `FUN_10009716` where you can spot a main loop with a kind of "State-Machine":



It looks like that a given double-word (stored in `ECX`) is controlling what the program is doing. But this looks convoluted and not very easy to unroll, since nothing is really in clear. For example, if you try to isolate the library API call in **x64dbg**, you will face something like this:



Every single API call is done in this way: there is a bunch of `MOV, XOR, SHIFT and PUSH` followed by a call to `xxx606F` (first red box), which decode in EAX the address of the function (called by the second red box). The number of `PUSH` just before the `CALL EAX` are the parameters, which could be worth to inspect.

The same "state" approach is also used in several sub-functions, not only in the main loop. So, everything looks time consuming, and I'd like to find a way to get the high level picture of it.

Speakeasy

This tool is a little gem: **Speakeasy** can emulate the execution of user and kernel mode malware, allowing you to interact with the emulated code by using quick Python scripts. What I'd like to do was to map every single state of the machine (`ECX` value of the main loop), to something more meaningful, like DLL API calls.

I had to work a bit to get what I wanted:

- the emulation was failing in more than one point, with some invalid read. I investigated a bit the reason, and I saw that sometimes the `CALL EAX` done in some location was not valid (`EAX` set to 0). I decided to get the easy way and just skip these calls
- I had to modify the call to a specific API (`CryptStringToBinary`)
- I mapped the machine state
- added a `--state` switch to control the flow of the emulation. You can use it to explore all the states (ex. `--state 0x167196bc`). You may encounter errors if needed parts are not initialized, but you can reconstruct the proper flow by looking at the Ghidra decompilation
- in a second iteration, knowing where strings are decrypted, I added a dump of all the strings in clear (see following sections)

Then the execution of the final script (`python emu_emotetdll.py -f sg.dll`) gave me something very interesting. The list of the imported DLLs (with related addresses):

```
0x10017a4c: 'kernel32.LoadLibraryW("advapi32.dll")' -> 0x78000000
0x10017a4c: 'kernel32.LoadLibraryW("crypt32.dll")' -> 0x58000000
0x10017a4c: 'kernel32.LoadLibraryW("shell32.dll")' -> 0x69000000
0x10017a4c: 'kernel32.LoadLibraryW("shlwapi.dll")' -> 0x67000000
0x10017a4c: 'kernel32.LoadLibraryW("urlmon.dll")' -> 0x54500000
0x10017a4c: 'kernel32.LoadLibraryW("userenv.dll")' -> 0x76500000
0x10017a4c: 'kernel32.LoadLibraryW("wininet.dll")' -> 0x7bc00000
0x10017a4c: 'kernel32.LoadLibraryW("wtsapi32.dll")' -> 0x63000000
...
```

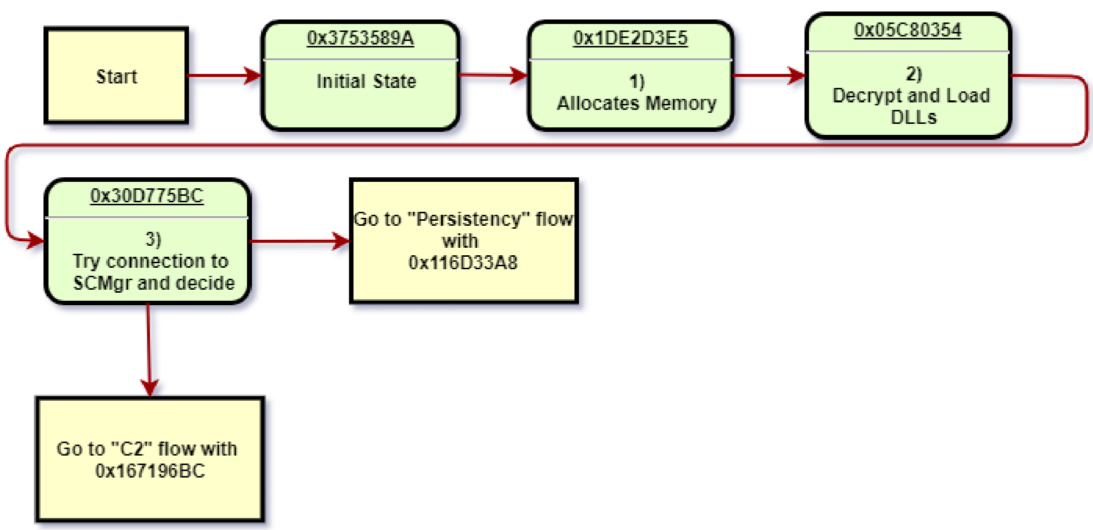
and a lot of API calls, mapped to the machine state:

```
[+] State: 1de2d3e5
0x10010ba0: 'kernel32.GetProcessHeap()' -> 0x7280
0x10018080: 'kernel32.HeapAlloc(0x7280, 0x8, 0x4c)' -> 0x72a0
[+] State: 5c80354
0x10010ba0: 'kernel32.GetProcessHeap()' -> 0x7280
0x10018080: 'kernel32.HeapAlloc(0x7280, 0x8, 0x20)' -> 0x72f0
0x10017a4c: 'kernel32.LoadLibraryW("advapi32.dll")' -> 0x78000000
0x10010ba0: 'kernel32.GetProcessHeap()' -> 0x7280
0x10014b3a: 'kernel32.HeapFree(0x7280, 0x0, 0x72f0)' -> 0x1
0x10010ba0: 'kernel32.GetProcessHeap()' -> 0x7280
...
```

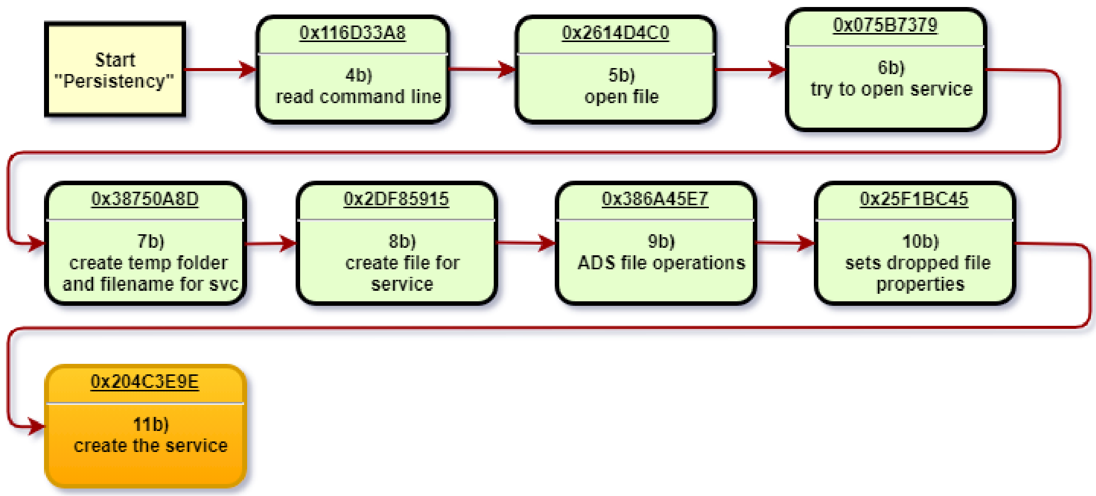
This list was not complete (because I skipped on purpose some failing calls and probably some calls were not correctly intercepted), but it gave me an overall picture of what was going on. Thanks FireEye!

Mapping

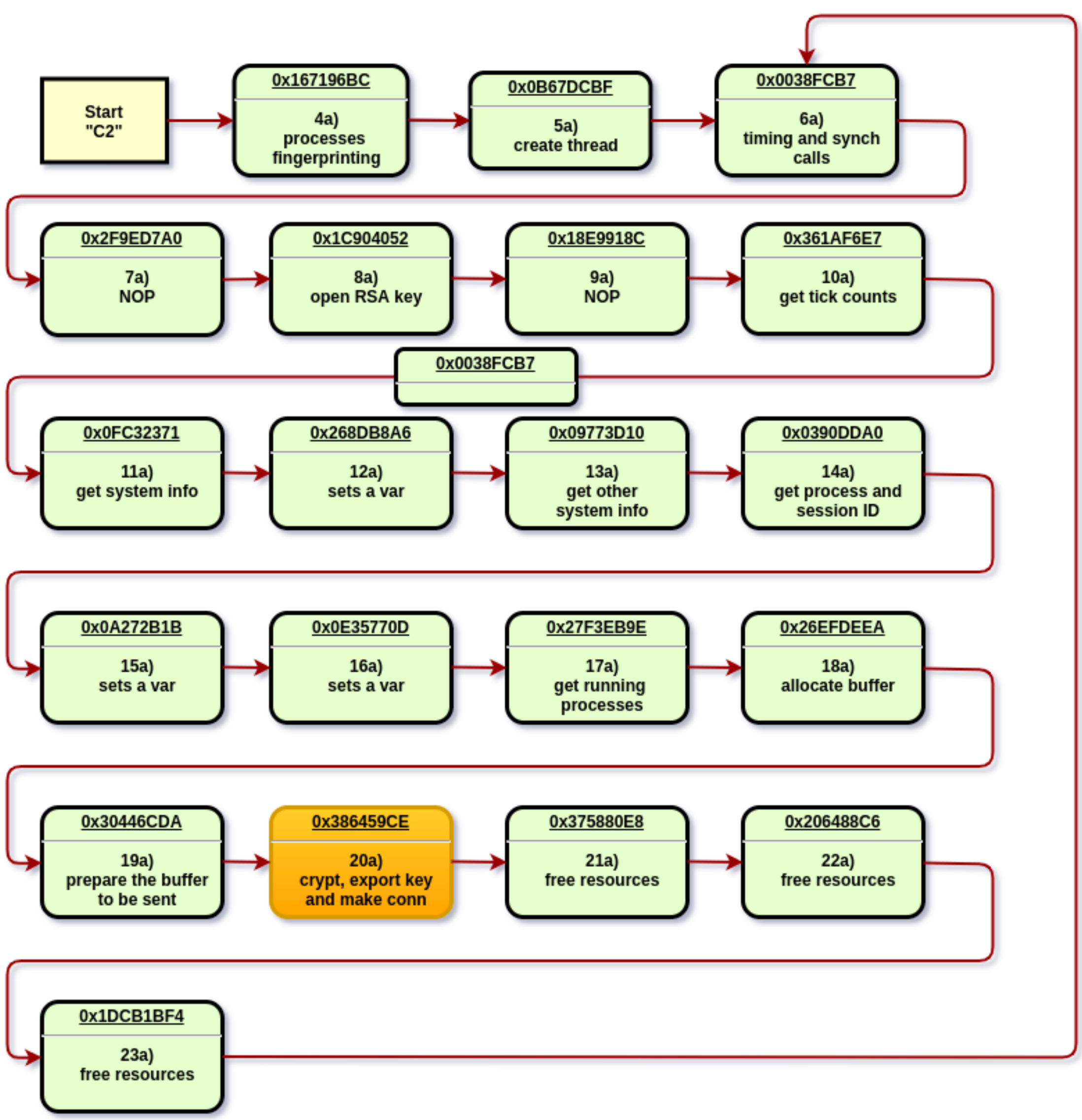
With the help of **Speakeasy** output and a combination of dynamic and static analysis (done with **x64gdb** and **Ghidra**), I was able to reconstruct the main flows of the Malware. Consider that these flows are not complete, they are high level snapshot of what is going on for some (not all) the "states". I'm sure something is missing. This is the "main" flow



Then we have the "Persistency" flow (the yellow boxes are the interesting ones):

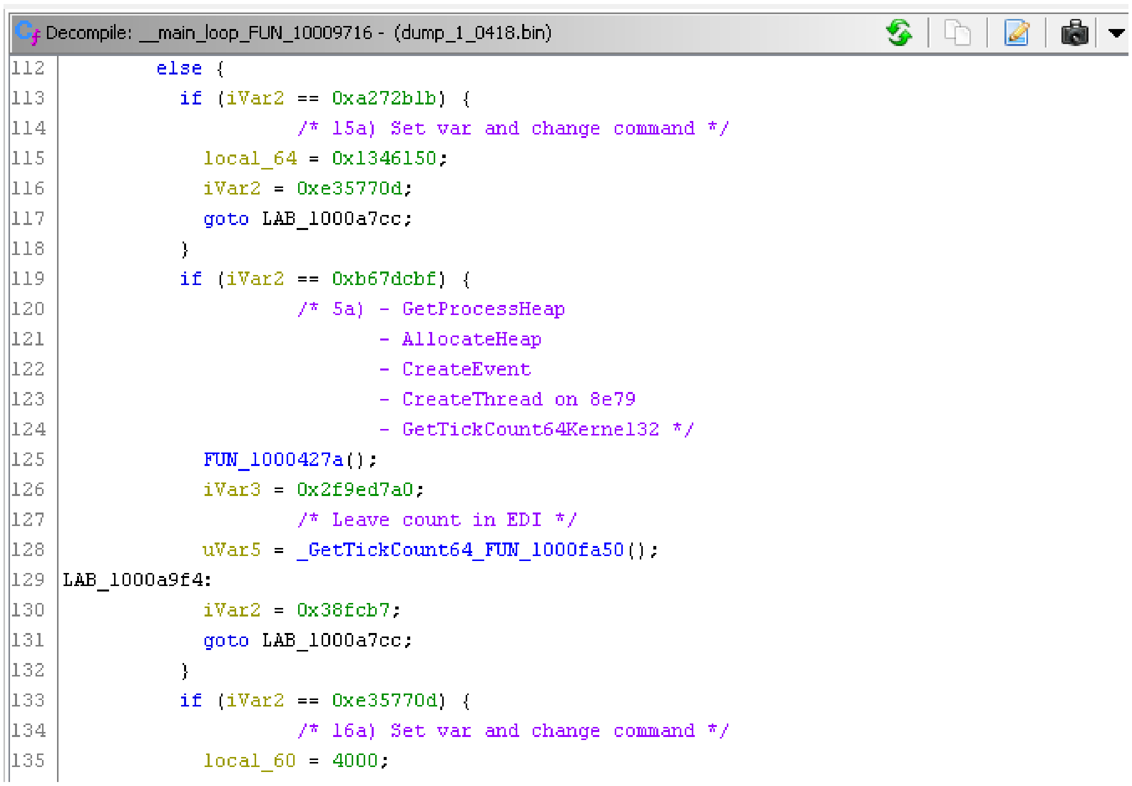


And the initial "C2" communication flow:



Not all the states were explored. I focused on persistence and initial C2. The great thing of this approach is that you can now alter the execution flow, by setting the `ECX` value you want to explore or execute.

I added a lot of details in the Ghidra file, by renaming the API calls and inserting comments. Every number reported in the graphs (ex **19a**) are in the comments, so you can easily track the code section.

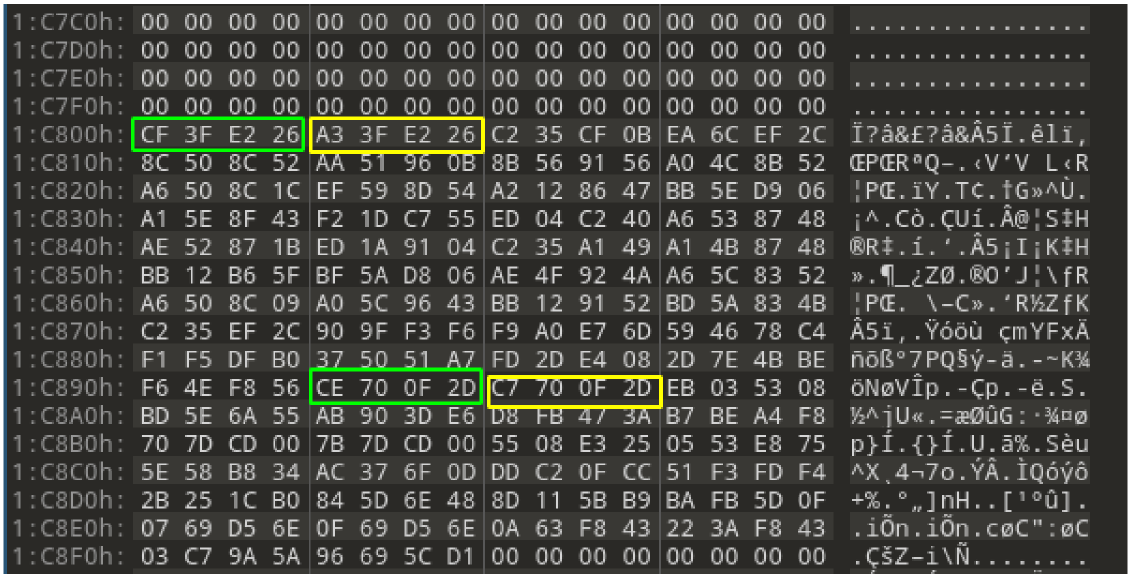


I renamed the functions with this standard:

- a single underscore in front of API calls
- a double underscore in front of internal function calls

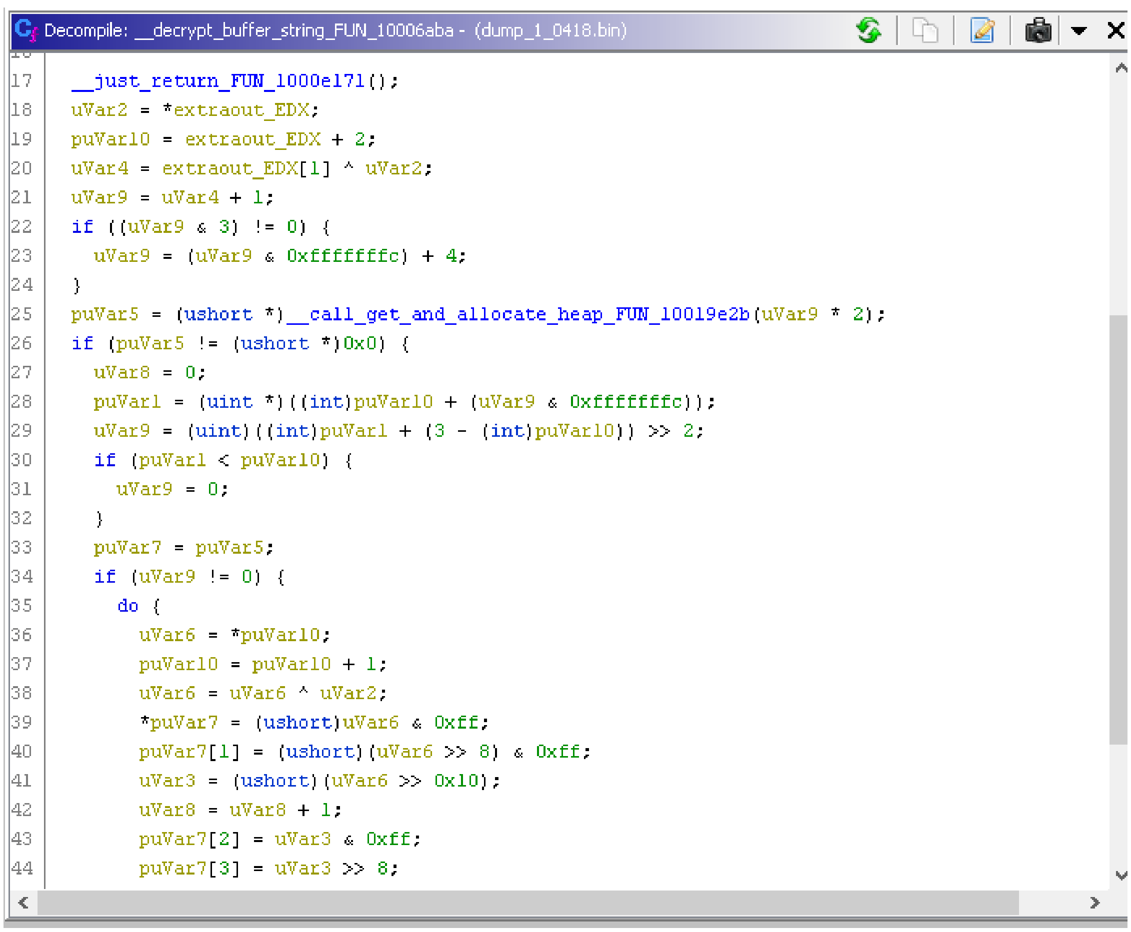
Interesting findings: encrypted strings

All the strings are encrypted in a BLOB, located, in this particular dumped sample, at `0x1C800`



```
1:C7C0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
1:C7D0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
1:C7E0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
1:C7F0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
1:C800h: CF 3F E2 26 A3 3F E2 26 C2 35 CF 0B EA 6C EF 2C  Ï?â&£?â&Â5Ï.êlï,
1:C810h: 8C 50 8C 52 AA 51 96 0B 8B 56 91 56 A0 4C 8B 52  ŒPŒRªQ-.‹V'V L‹R
1:C820h: A6 50 8C 1C EF 59 8D 54 A2 12 86 47 BB 5E D9 06  ¦PŒ.ïY.T¢.†G»^Ù.
1:C830h: A1 5E 8F 43 F2 1D C7 55 ED 04 C2 40 A6 53 87 48  ¡^.Cò.ÇUí.Â@¦S‡H
1:C840h: AE 52 87 1B ED 1A 91 04 C2 35 A1 49 A1 4B 87 48  ®R‡.í.'.Â5¡I¡K‡H
1:C850h: BB 12 B6 5F BF 5A D8 06 AE 4F 92 4A A6 5C 83 52  ».¶_¿ZØ.®O'J¦\ƒR
1:C860h: A6 50 8C 09 A0 5C 96 43 BB 12 91 52 BD 5A 83 4B  ¦PŒ. \-C».'R½ZƒK
1:C870h: C2 35 EF 2C 90 9F F3 F6 F9 A0 E7 6D 59 46 78 C4  Â5ï,.Ÿóöù çmYFxÄ
1:C880h: F1 F5 DF B0 37 50 51 A7 FD 2D E4 08 2D 7E 4B BE  ñõß°7PQ§ý-ä.-~K¾
1:C890h: F6 4E F8 56 CE 70 0F 2D C7 70 0F 2D EB 03 53 08  öNøVÎp.-Çp.-ë.S.
1:C8A0h: BD 5E 6A 55 AB 90 3D E6 D8 FB 47 3A B7 BE A4 F8  ½^jU«.=æØûG:·¾¤ø
1:C8B0h: 70 7D CD 00 7B 7D CD 00 55 08 E3 25 05 53 E8 75  p}Í.{}Í.U.ã%.Sèu
1:C8C0h: 5E 58 B8 34 AC 37 6F 0D DD C2 0F CC 51 F3 FD F4  ^X¸4¬7o.ÝÂ.ÌQóýô
1:C8D0h: 2B 25 1C B0 84 5D 6E 48 8D 11 5B B9 BA FB 5D 0F  +%.°„]nH..[¹ºû].
1:C8E0h: 07 69 D5 6E 0F 69 D5 6E 0A 63 F8 43 22 3A F8 43  .iÕn.iÕn.cøC":øC
1:C8F0h: 03 C7 9A 5A 96 69 5C D1 00 00 00 00 00 00 00 00  .ÇšZ-i\Ñ........
```

The **green** box is the XOR key and the **yellow** one is the length of the string. The function used to perform the decryption is the `__decrypt_buffer_string_FUN_10006aba` and `__decrypt_headers_footer_FUN_100033f4`



Decompile: __decrypt_buffer_string_FUN_10006aba - (dump_1_0418.bin)

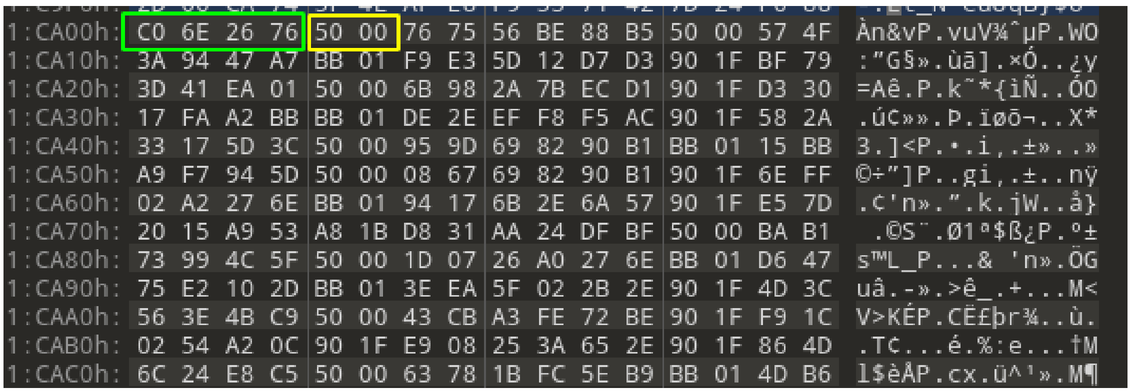
```
  __just_return_FUN_1000e171();
  uVar2 = *extraout_EDX;
  puVar10 = extraout_EDX + 2;
  uVar4 = extraout_EDX[1] ^ uVar2;
  uVar9 = uVar4 + 1;
  if ((uVar9 & 3) != 0) {
    uVar9 = (uVar9 & 0xfffffffc) + 4;
  }
  puVar5 = (ushort *)__call_get_and_allocate_heap_FUN_10019e2b(uVar9 * 2);
  if (puVar5 != (ushort *)0x0) {
    uVar8 = 0;
    puVar1 = (uint *)((int)puVar10 + (uVar9 & 0xfffffffc));
    uVar9 = (uint)((int)puVar1 + (3 - (int)puVar10)) >> 2;
    if (puVar1 < puVar10) {
      uVar9 = 0;
    }
    puVar7 = puVar5;
    if (uVar9 != 0) {
      do {
        uVar6 = *puVar10;
        puVar10 = puVar10 + 1;
        uVar6 = uVar6 ^ uVar2;
        *puVar7 = (ushort)uVar6 & 0xff;
        puVar7[1] = (ushort)(uVar6 >> 8) & 0xff;
        uVar3 = (ushort)(uVar6 >> 0x10);
        uVar8 = uVar8 + 1;
        puVar7[2] = uVar3 & 0xff;
        puVar7[3] = uVar3 >> 8;
```

Every single string is decrypted and then removed from memory after usage. This is true even for C format strings. So you will not find anything in memory if you try to inspect the mapped sections at runtime.

As said before, I added a specific section in the **Speakeasy** script to dump those strings.

Interesting findings: list of C2 servers

IP of C2 are dumped form the same BLOB (in this case at `0x1CA00`) just after the decryption in step `20a` .



As stated in Fortinet Analysis, this list is made of IP (green box) and port (yellow box). You can decode the whole list if you pass this part of the binary in the following python code:

```
import sys
import struct

b = bytearray(sys.stdin.buffer.read())

for x in range(0,len(b),8):
    print('%u.%u.%u.%u:%u' %
(b[x+3],b[x+2],b[x+1],b[x],struct.unpack('<H',bytes(b[x+4:x+6]))[0]))
```

You can find the full list extracted in **IoC** section.

Interesting findings: persistence

This particular sample obtain persistency by installing a System Service. This campaign deployed different versions of the DLL using also different techniques: `Run` Registry Key is one of them.

The section installing the service is the **20a** (state `0x204C3E9E`). The high level steps are the following:

- decrypt the format string `%s.%s`
- generates random chars to build the service name (which results in something like `xzyw.qwe`)
- get one random "Service Description" from the existing ones, and use it as description of the new service

Interesting findings: encrypted communications with C2

In section **8a** (state `0x1C904052`) we can spot out the load of a RSA public key

```
CryptImportKey
pbData (0x74 bytes)

0316B0C0  06 02 00 00 00 A4 00 00 52 53 41 31 00 03 00 00  .....¤..RSA1....
0316B0D0  01 00 01 00 15 78 AE D2 E5 38 03 34 E9 7F F3 96  .....x®Òå8.4é.ó.
0316B0E0  88 F2 20 78 38 BA 9B 63 9C DE 64 E3 EA 73 79 3C  .ò x8º.c.Þdãêsy<
0316B0F0  3F 71 1E 44 D2 E1 89 40 5B 94 8D C1 F8 CF 7F D9  ?q.DÒá.@[..ÁøÏ.Ù
0316B100  8E 3A 47 21 94 27 3D CC 8A 57 42 18 C0 CE 48 C1  .:G!.'=Ì.WB.ÀÎHÁ
0316B110  35 A0 A5 D9 56 81 99 A0 68 7F F9 2E B9 FA 7C 8F  5 ¥ÙV.. h.ù.¹ú|.
0316B120  CF 27 03 A3 E8 DD 11 FB A4 11 95 39 34 B2 52 7C  Ï'.£èÝ.û¤..94²R|
0316B130  B2 7C 7D E6 74 00 65 00 3B 6C 34 38 40 D0 00 00  ²|}æt.e.;l48@Ð..
0316B140  C0 00 14 03 28 88 16 03 63 00 6F 00 64 00 65 00  À...(...c.o.d.e.
0316B150  45 00 78 00 00 00 00 00 00 00 00 00 00 00 00 00  E.x.............
0316B160  00 00 00 00 00 00 00 00 27 07 00 00 00 00 00 00  ........'.......
```

After this we have a call to `CryptGenKey` with algo `CALG_AES_128` . So it looks that the sample is going to use a symmetric key to encrypt communication.

In section **20a** (state `0x386459ce`) we see how the communication is encrypted:

- CryptGenKey
- CryptEncrypt of the buffer to send, with the previous key
- CryptExportKey encrypted with the RSA public key
- the exported and encrypted symmetric key is then prepended to the buffer sent via HTTP

Wrap up

The analysis is far to be complete, there are a lot of unexplored part of the sample. At the end my goal was to build a procedure to make the analysis easier, even for different or future samples, where it would be faster to understand the overall picture.

Appendix: IoC

C2 IP list:

118.38.110.192:80
181.136.190.86:80
167.71.148.58:443
211.215.18.93:8080
1.234.65.61:80
209.236.123.42:8080
187.162.250.23:443
172.245.248.239:8080
60.93.23.51:80
177.144.130.105:443
93.148.247.169:80
177.144.130.105:8080
110.39.162.2:443
87.106.46.107:8080
83.169.21.32:7080
191.223.36.170:80
95.76.153.115:80
110.39.160.38:443
45.16.226.117:443
46.43.2.95:8080
201.75.62.86:80
190.114.254.163:8080
12.162.84.2:8080
46.101.58.37:8080
197.232.36.108:80
185.94.252.27:443
70.32.84.74:8080
202.79.24.136:443
2.80.112.146:80
202.134.4.210:7080
105.209.235.113:8080
187.162.248.237:80
190.64.88.186:443
111.67.12.221:8080
5.196.35.138:7080
50.28.51.143:8080
181.30.61.163:443
103.236.179.162:80
81.215.230.173:443
190.251.216.100:80
51.255.165.160:8080
149.202.72.142:7080
192.175.111.212:7080
178.250.54.208:8080
24.232.228.233:80
190.45.24.210:80
45.184.103.73:80
177.85.167.10:80
212.71.237.140:8080
181.120.29.49:80
170.81.48.2:80
68.183.170.114:8080
35.143.99.174:80
217.13.106.14:8080
168.121.4.238:80

172.104.169.32:8080
111.67.12.222:8080
62.84.75.50:80
77.78.196.173:443
177.23.7.151:80
213.52.74.198:80
12.163.208.58:80
1.226.84.243:8080
113.163.216.135:80
188.225.32.231:7080
191.182.6.118:80
81.213.175.132:80
104.131.41.185:8080
152.169.22.67:80
185.183.16.47:80
192.232.229.54:7080
186.146.13.184:443
178.211.45.66:8080
122.201.23.45:443
70.32.115.157:8080
190.24.243.186:80
51.15.7.145:80
46.105.114.137:8080
81.214.253.80:443
192.232.229.53:4143
59.148.253.194:8080
191.241.233.198:80
181.61.182.143:80
190.195.129.227:8090
68.183.190.199:8080
138.97.60.140:8080
138.97.60.141:7080
137.74.106.111:7080
85.214.26.7:8080
71.58.233.254:80
94.176.234.118:443
188.135.15.49:80
80.15.100.37:80
82.76.111.249:443
155.186.9.160:80
189.2.177.210:443