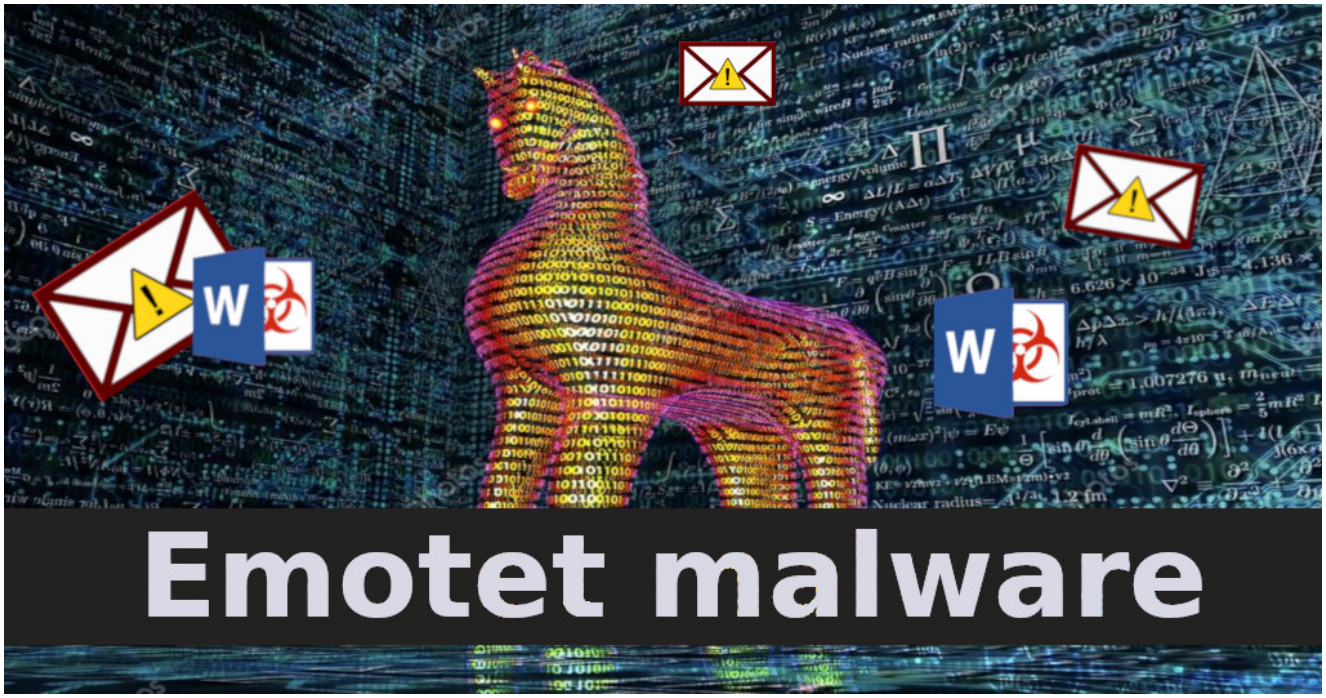
[Emotet malware analysis. Part 2.]

persianov.net/emotet-malware-analysis-part-2

April 7, 2019



=== Apr 7, 2019 ===

This is the Part 2 of my Emotet analysis. It covers phase 3 of the attack, specifically the PE file which is being dropped by infected websites, used in Phishing/Spam campaigns. Emotet is an advanced modular Trojan, predominantly used as Malware Distribution Platform, main goal being systems infection with other types of malware.

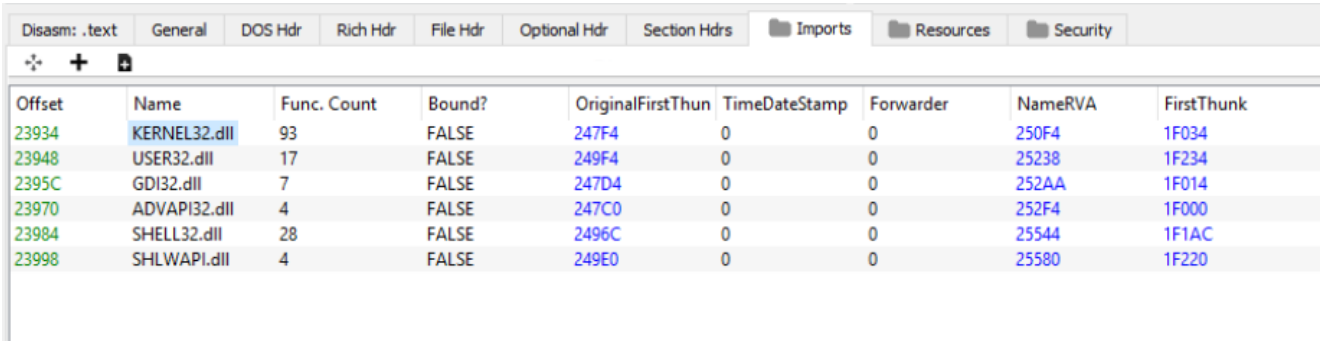
More information about phases 1 and 2: HERE

General information.

Phase 3 of this version of Emotet attack is characterized by the new version of executable. Malware authors spent some time to make it harder to analyze, by implementing multiple Anti-Debugging techniques, loading Windows DLLs dynamically, encrypting imported functions names, several unpacking stages, and so on.

File name	Checksum	Hosted at
DFDWiz.exe	cebb919d8d04f224b78181a4d3f0b10a315ae2f2	hxxp://biederman.net/leslie/lL/

Based on IAT information, there are several Windows DLLs this binary is loading: `kernel32.dll`, `user32.dll`, `gdi32.dll`, `advapi32.dll`, `shell32.dll` and `shlwapi.dll`.



Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
23934	KERNEL32.dll	93	FALSE	247F4	0	0	250F4	1F034
23948	USER32.dll	17	FALSE	249F4	0	0	25238	1F234
2395C	GDI32.dll	7	FALSE	247D4	0	0	252AA	1F014
23970	ADVAPI32.dll	4	FALSE	247C0	0	0	252F4	1F000
23984	SHELL32.dll	28	FALSE	2496C	0	0	25544	1F1AC
23998	SHLWAPI.dll	4	FALSE	249E0	0	0	25580	1F220

At the first glance, the binary doesn't import any "red flag" functions, usually used by packers.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size
.text	400	1DE00	1000	1DC68
.rdata	1E200	6600	1F000	658C
.data	24800	5C00	26000	5B14
.rsrc	2A400	1C00	2C000	1BF0

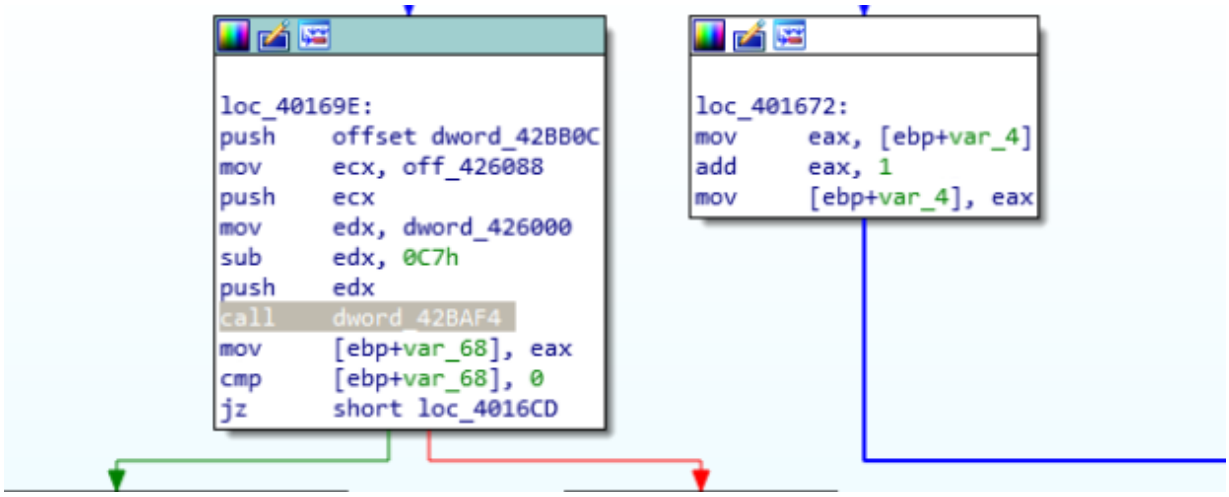
Let's load this sample in IDA and check for any details that can help in upcoming dynamic analysis. From the multitude of imported functions, listed in IAT, there are only few used (visible at least). One of them is `VirtualAlloc`, which points to a possible custom packer.

```
sub_401470 proc near

var_235= byte ptr -235h
var_1C3= byte ptr -1C3h
var_18= dword ptr -18h
var_14= dword ptr -14h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 3D0h
mov     [ebp+var_4], 40h
mov     [ebp+var_C], 0
mov     eax, dword_42BA84
mov     [ebp+var_18], eax
mov     [ebp+var_8], 0FFFFFFFFh
mov     [ebp+var_1C3], 74h
mov     [ebp+var_235], 3Eh
mov     ecx, ds:VirtualAlloc
mov     dword_42BAF0, ecx
push    [ebp+var_4]
mov     eax, 3001h
dec     eax
push    eax
mov     eax, [ebp+var_18]
push    eax
push    [ebp+var_C]
mov     edx, dword_42BAF0
push    offset loc_4014CE
push    edx
retn
```

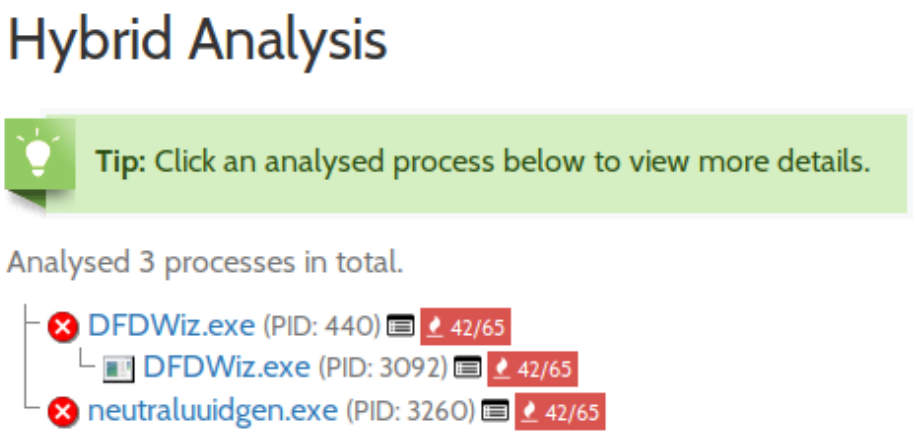
Some other clues showing that this binary is packed are `call`s to pointers to data segment:



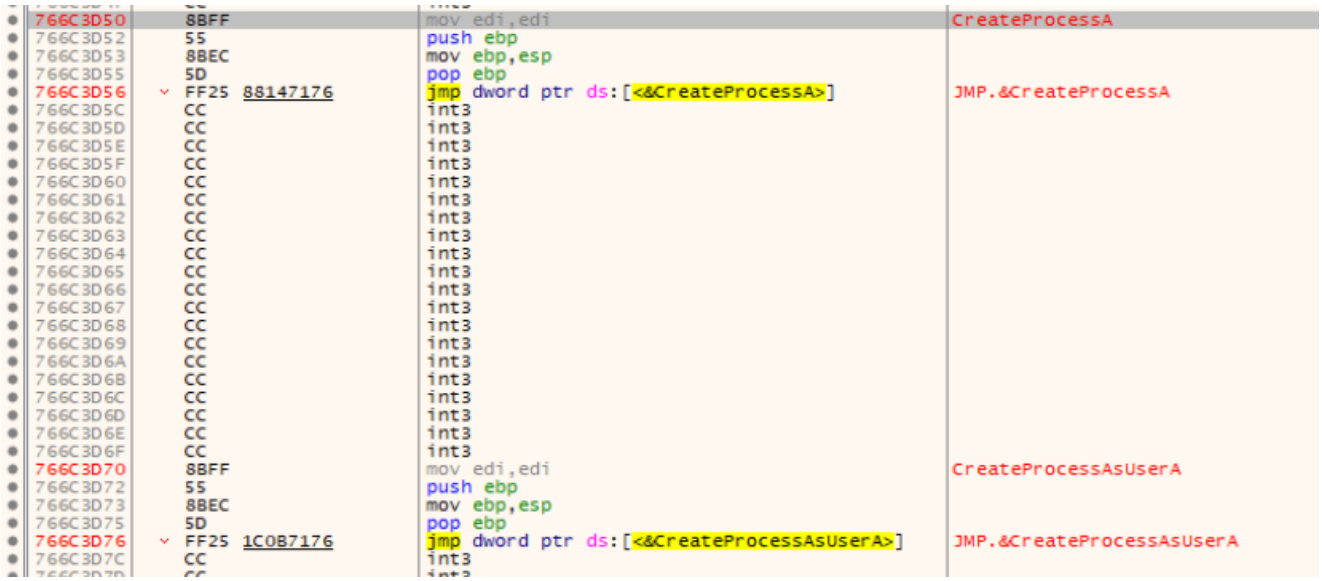
Dynamic analysis. Unpacking.

For the dynamic analysis of this sample, I'm going to use `x32dbg`. Based on the report from Hybrid Analysis, looks like this sample creates 2 new processes, once executed: copy of itself and a second process with a different name. This is the packed PE file, which is embedded

into the first sample.



Most probably, parent process will call one of the `CreateProcess*` Windows API functions. Since 2018, Microsoft moved some functionality from **kernel32.dll** and **advapi32.dll** to new low-level binary, called `kernelbase.dll`. If we take a look at `CreateProcessA` and `CreateProcessAsUserA` in kernel32.dll, the only thing we can see are several `mov` and `push` instructions, followed by a jump, to **kernelbase.dll** equivalent function.



Following the thread to kernelbase.dll, we see that `CreateProcessA` function contains a bunch of another push instructions followed by a `call` to `CreateProcessInternalA`. Same happens for `CreateProcessAsUserA`.

```
777C5EE0    8BFF          mov edi,edi                          CreateProcessA
777C5EE2    55            push ebp
777C5EE3    8BEC          mov ebp,esp
777C5EE5    6A 00         push 0
777C5EE7    FF75 2C       push dword ptr ss:[ebp+2C]
777C5EEA    FF75 28       push dword ptr ss:[ebp+28]
777C5EED    FF75 24       push dword ptr ss:[ebp+24]
777C5EF0    FF75 20       push dword ptr ss:[ebp+20]
777C5EF3    FF75 1C       push dword ptr ss:[ebp+1C]
777C5EF6    FF75 18       push dword ptr ss:[ebp+18]
777C5EF9    FF75 14       push dword ptr ss:[ebp+14]
777C5EFC    FF75 10       push dword ptr ss:[ebp+10]
777C5EFF    FF75 0C       push dword ptr ss:[ebp+C]
777C5F02    FF75 08       push dword ptr ss:[ebp+8]
777C5F05    6A 00         push 0
777C5F07    E8 94000000   call <kernelbase.CreateProcessInternalA>
777C5F0C    5D            pop ebp
777C5F0D    C2 2800       ret 28
```

To keep this short, the overall call chain looks like this:

[kernel32.dll] `CreateProcessA` -> [kernelbase.dll] `CreateProcessA` -> [kernelbase.dll] `CreateProcessInternalA` -> [kernelbase.dll] `CreateProcessInternalW`

It means that for any `CreateProcess*` function call, we'll get `CreateProcessInternalW` called right before process creation. If we set a breakpoint at the beginning of this function, we possibly could find the **unpacked** binary, which is going to be injected into the new process. Once we hit the breakpoint, there are 4 memory regions with **ERW** (Execute-Read-Write) flags set. 3/4 are PE files, based on the header.

```
00318000  000E5000  Reserved (00200000)                          PRV             -RW--
00400000  0001A000                                             PRV  ERW--      ERW--
00430000  000C5000  \Device\HarddiskVolume2\Windows\System32\lc  MAP  -R---      -R---
00500000  00035000  Reserved                                     PRV             -RW--
00535000  0000B000                                             PRV  -RW-G      -RW--
00540000  00015000                                             PRV  ERW--      ERW--
00560000  00014000                                             PRV  ERW--      ERW--
00580000  0001A000                                             PRV  ERW--      ERW--
005C0000  00006000                                             PRV  -RW--      -RW--
005C6000  0000A000  Reserved (005C0000)                        PRV             -RW--
```

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Lo

```
Address   Hex                                              ASCII
005B0000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
005B0010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
005B0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
005B0030  00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00  ............À...
005B0040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..°..´.Í!¸.LÍ!Th
005B0050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
005B0060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
005B0070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
```

Once all of them are dumped and properly aligned, we can proceed with the second phase of analysis.

Dynamic analysis. Dumped binaries.

All 3 exported binaries look the same, even if they have different checksums. IAT table is empty, which means that malware loads dependencies in runtime. There are no API function names in binary's strings, which implies that all API function names are encrypted as well as library names.

The execution starts with 3 function calls.

```
public start
start proc near
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
call    sub_40D340
call    sub_40DF80
call    sub_401030
test    eax, eax
jz      short loc_40F5FE
```

First 2 functions contain encrypted names of the APIs this sample is going to load dynamically. These functions call another one, once all encrypted values are loaded to stack. The `sub_401550` looks to be responsible for decryption.

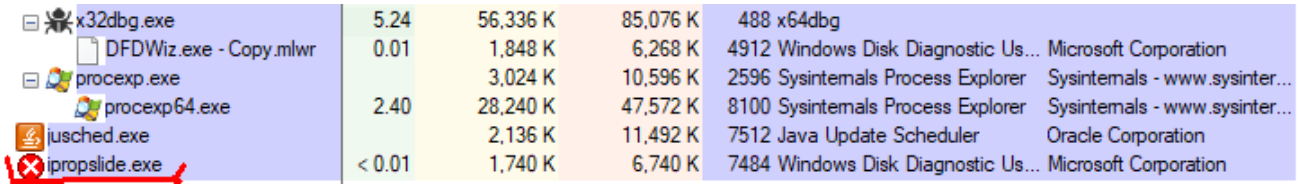
```
mov     [ebp+var_54], 9A171EFFh
mov     [ebp+var_50], 7A6259BFh
mov     [ebp+var_4C], 5988B28Eh
mov     [ebp+var_48], 80CA441Fh
mov     [ebp+var_44], 73AAB973h
mov     [ebp+var_40], 0AD728D54h
mov     [ebp+var_3C], 55B2FEBAh
mov     [ebp+var_38], 0EE9D51ABh
mov     [ebp+var_34], 6FA72180h
mov     [ebp+var_30], 0C59B6F09h
mov     [ebp+var_2C], 2915C1DEh
mov     [ebp+var_28], 0DF94CDB4h
mov     [ebp+var_24], 6E525CBBh
mov     [ebp+var_20], 31C1921Ah
mov     [ebp+var_1C], 8F22AA15h
mov     [ebp+var_18], 0B64C044h
mov     [ebp+var_14], 62AEF5A2h
mov     [ebp+var_10], 4DE207Eh
mov     [ebp+var_C], 50099F52h
mov     [ebp+var_8], 7BFACF9Dh
mov     [ebp+var_4], 0B69B27D5h
call    sub_401550
test    eax, eax
jz      short loc_40DF78
```

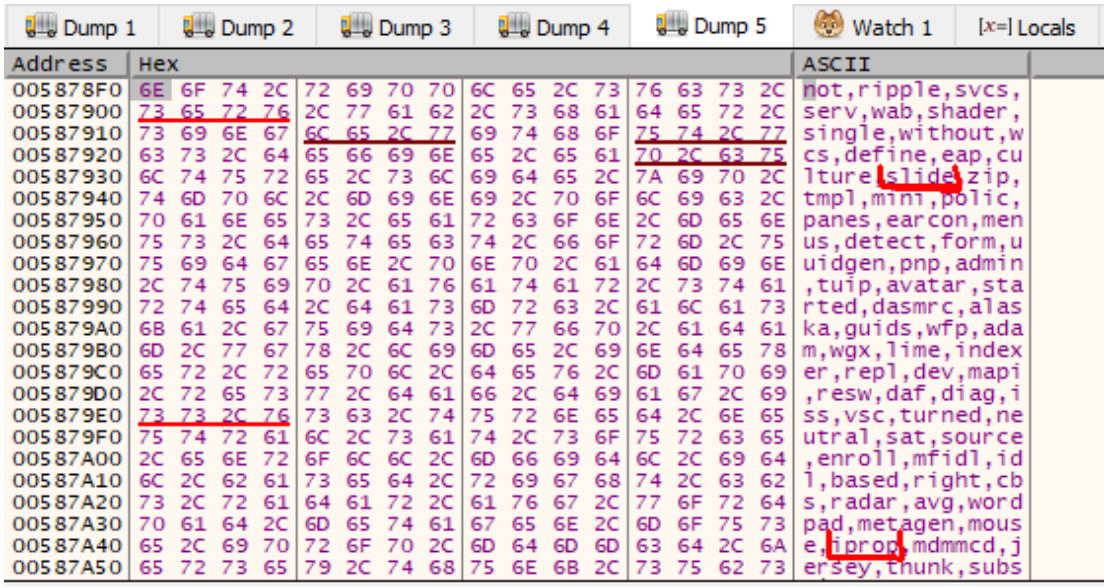
So far, I was able to detect 4 DLLs loaded dynamically by this sample: `kernel32.dll`, `user32.dll`, `ntdll.dll`, `shell32.dll`.

In order to run just one copy of it, this sample checks if a specific MUTEX exists and creates it, if missing. MUTEX name is: `PEMF24`.

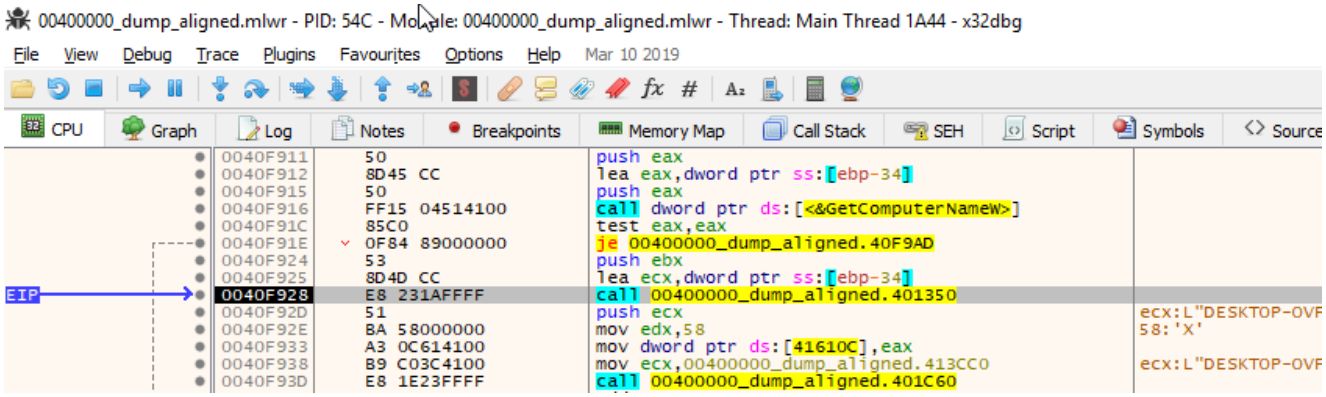
```
0040107E   50              push eax                                 eax:L"PEMF24"
0040107F   FF15 DC594100   call dword ptr ds:[<&_snwprintf>]
00401085   83C4 10         add esp,10
00401088   56              push esi                                 esi:"X@W"
00401089   57              push edi
0040108A   FF15 C0524100   call dword ptr ds:[<&GetProcessHeap>]
00401090   50              push eax                                 eax:L"PEMF24"
00401091   FF15 C4514100   call dword ptr ds:[<&HeapFree>]
00401097   8D85 E8FEFFFF   lea eax,dword ptr ss:[ebp-118]
0040109D   50              push eax                                 eax:L"PEMF24"
0040109E   6A 01           push 1
004010A0   57              push edi
EIP 004010A1   FF15 904F4100   call dword ptr ds:[<&CreateMutexW>]
004010A7   8BD8            mov ebx,eax                              eax:L"PEMF24"
004010A9   85DB            test ebx,ebx
004010AB   0F84 82000000   je 00400000_dump_aligned.401133
004010B1   FF15 80544100   call dword ptr ds:[<&GetLastError>]
004010B7   3D B7000000     cmp eax,B7                               eax:L"PEMF24"
004010BC   75 66           jne 00400000_dump_aligned.401124
004010BE   68 269FDA64     push 64DA9F26
004010C3   8D50 1D         lea edx,dword ptr ds:[eax+1D]
004010C6   B9 20204100     mov ecx,00400000_dump_aligned.412020
004010CB   E8 300C0000     call 00400000_dump_aligned.401D00
004010D0   83C4 04         add esp,4
004010D3   8BF0            mov esi,eax                              esi:"X@W", eax:L"PEMF24"
004010D5   8D85 68FFFFFF   lea eax,dword ptr ss:[ebp-98]
004010DB   FF75 FC         push dword ptr ss:[ebp-4]
```

Once MUTEX is checked/created, malware looks for Windows directory to copy itself there, as well as into **%APPDATA%** folder. This time, the new binary is named differently and this name is generated in runtime by concatenating 2 strings (in my case it was `ipropslide.exe`). All possible strings are stored in memory at some point.





Once the new process is created, it starts looking for host information like **Computer Name** and **Volume info** and C2 communication begins.



After some patching during debugging and several failures =)) I was able to get some details about the C2 communication part. Sample tries to connect to 3 IP addresses (round robin?)

IP Address	Destination Port	Protocol	User Agent

IP Address	Destination Port	Protocol	User Agent
45.36.20[.]17	8443	HTTP	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
103.39.131[.]88	80	HTTP	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
86.239.117[.]57	8090	HTTP	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)

Conclusion.

Malware authors did a good effort in packing this malware and introducing different layers of defence. It wasn't too difficult to bypass those layers, however taking into consideration how often a new Emotet version is released and the changes to the binary, the analysis becomes time consuming in the long term. Sending HTTP traffic to non-standard destination ports, like `8090`, is not the best way to keep a low profile in a compromised network. Most businesses have to treat this type of traffic as suspicious nowadays and maintain a clean asset inventory.