oR10n Labs

By oR10n

2020-05-24

Reverse Engineering the Mustang Panda PlugX Loader

Hello everyone! In this series, we will be diving into the inner workings of a new-ish variant of PlugX malware gaining traction around the Asia Pacific region for the past few months.

Introduction

PlugX is a fully featured remote access trojan (RAT) with various capabilities such as file upload/download, file operations, registry operations, process operations, keystroke logging, capturing screenshots or videos, and initiating remote shell on compromised systems.

Based on the analysis reports released by two security companies – Anomali and Avira, this new variant is primarily used by a suspected China-based APT group being referred to as "Mustang Panda", to target organizations primarily located in the Asia Pacific region.

For this post, we will reverse engineer the loader component of the new variant to understand how it loads, decrypts, and executes the encrypted payload in memory. Then, we will create a quick-and-dirty python script to automate the decryption process so we won't need to run the loader every time we want to do a deeper analysis on a payload or perform bulk analysis. Lastly, I will show you one of the ways to hunt for new encrypted payloads uploaded in VirusTotal.

But before we get our hands dirty, let us first take a look on how PlugX is initially delivered and executed on a system.

PlugX Delivery and Execution

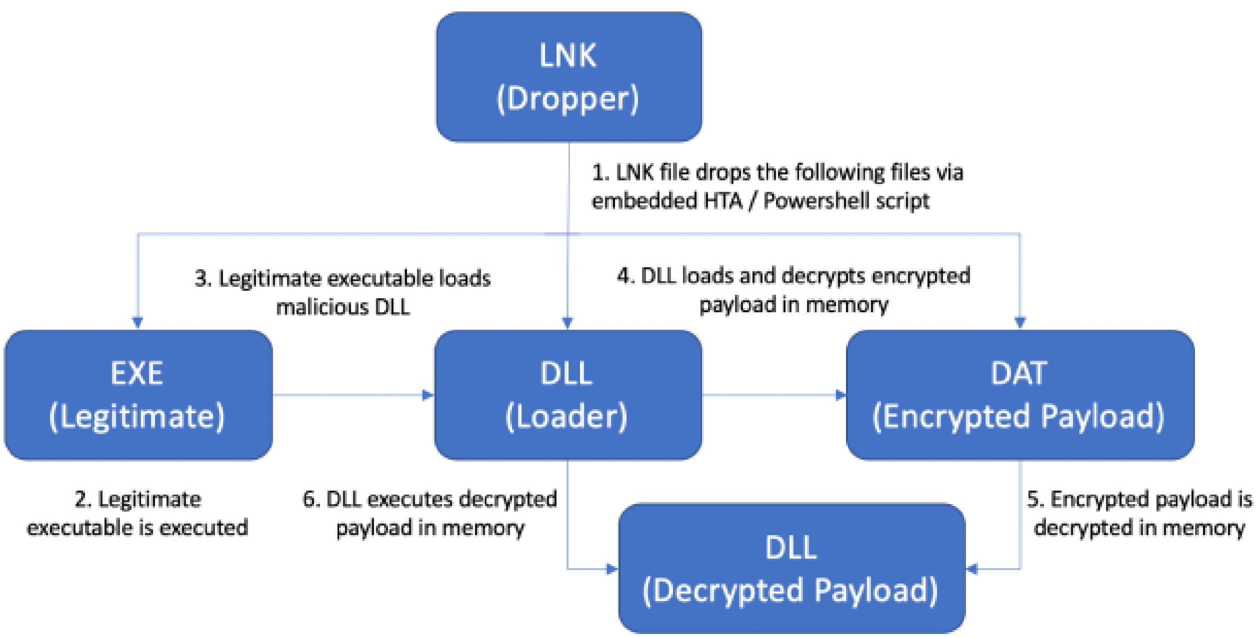
As you might have known from previous analysis reports, PlugX is primarily made up of three main components:

1. A legitimate executable used for loading and executing a malicious DLL
2. A malicious DLL used for decrypting, loading, and executing an encrypted payload
3. An encrypted payload containing the main RAT functionalities

On the earlier variants of PlugX, these three components are typically delivered via phishing emails containing an attached self-extracting RAR (SFX) archive, acting as a dropper for these components. However for this variant, this RAR archive was replaced by a malicious LNK file as seen on the Anomali analysis report.

Note: Logrhythm released an article on April 2018, detailing the evolution and variants of PlugX over the years.

A general overview of the delivery and execution flow looks like this:



Reverse Engineering the Loader

For our analysis, we will take a deeper look at a sample discovered by Avira in the wild. These are the details of the PlugX components for our analysis:

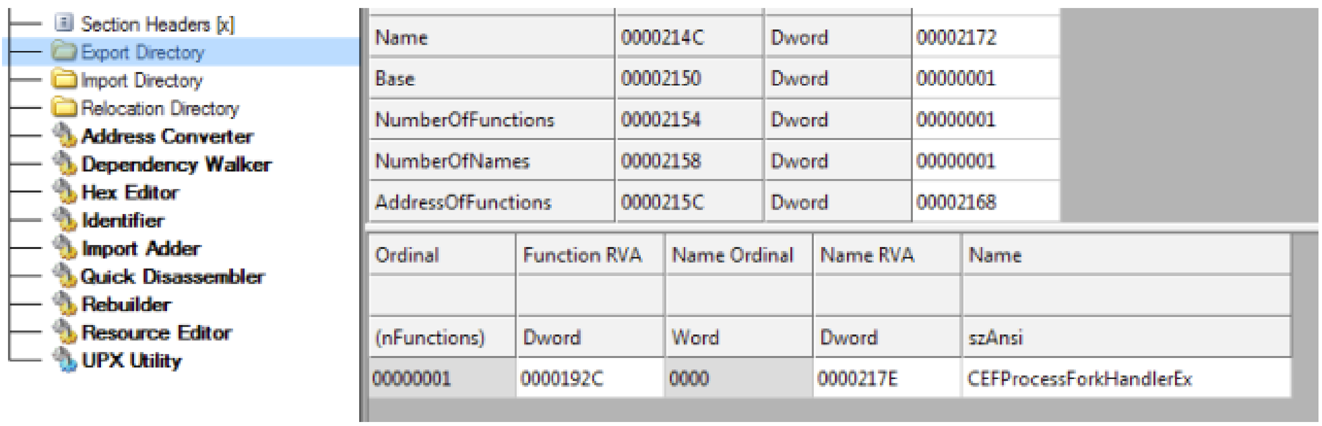
Component	Filename	MD5

Component	Filename	MD5
Legit exe	AdobeInstall.exe	c70d8dce46b4551133ecc58aed84bf0e
Loader	hex.dll	eafaba7898e149895b36ee488e3d579c
Payload	adobeupdate.dat	58bdf783da4c627d2f13612a09a9b5a8

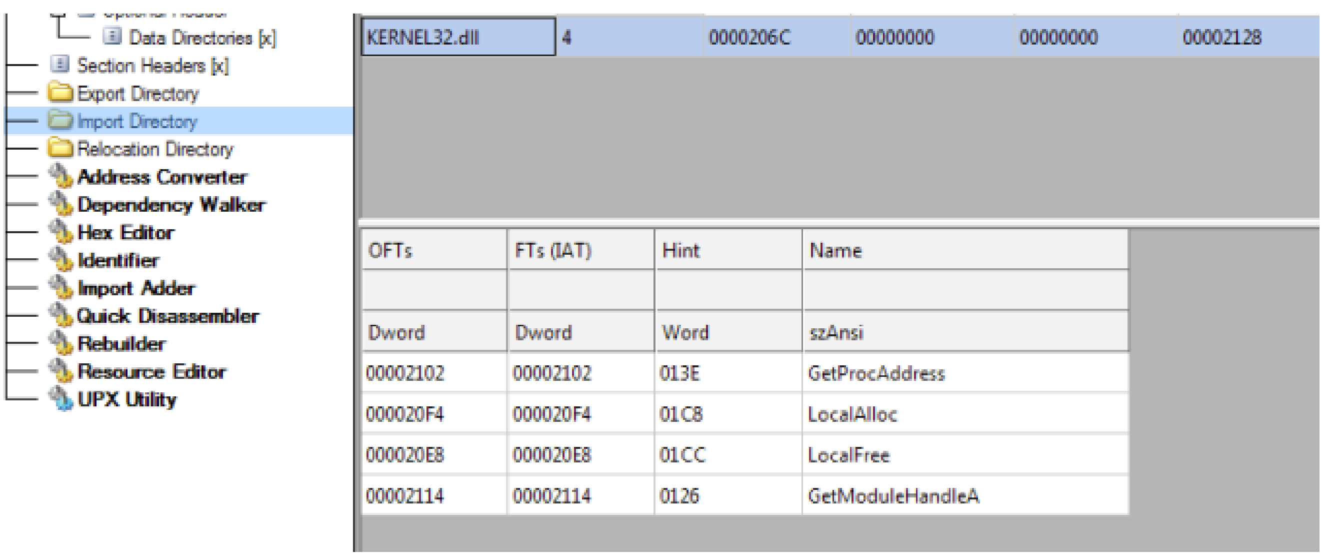
Let’s dive in!

As a first step in reverse engineering, it’s a good practice to perform static analysis first to gain a general overview of the sample that can serve as a guide through out the process.

Checking the sample on CFF explorer shows us that it has only 1 Export function named **CEFProcessForHandlerEx**.



It also has a very few Import functions which suggests that this sample dynamically loads Win32 API functions at runtime via **GetModuleHandleA** and **GetProcAddress**.



Additionally, we can also run a string utility to check out the strings on the sample. I normally use FIreEye's FLOSS tool which is a string utility on steroids. Aside from displaying static ASCII/UNICODE strings, it can also display stack strings and automatically decode strings that are encoded with simple and well known algorithms.

Here are some of the interesting strings FLOSS found on the sample:

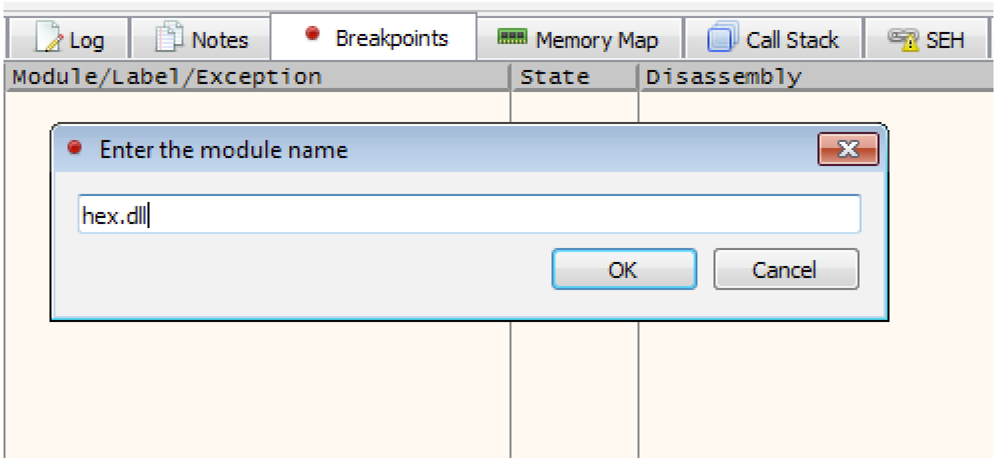
```
LocalFree
LocalAlloc
GetProcAddress
GetModuleHandleA
VirtualProtect
CloseHandle
CreateFileA
ReadFile
\adobeupdate.dat
GetModuleFileNameA
kernel32
GetFileSize
lstrcatA
strlen
```

Based from these strings, we can come up with a hypothesis on the general flow and functionality of the sample:

- Dynamically loads Win32 API functions from **kernel32** at runtime via **GetModuleHandleA** and **GetProcAddress**
- Obtains the full path of the running process via **GetModuleFileNameA**
- Performs string operations via **strlen** and **lstrcatA**
- Allocates a memory buffer via **LocalAlloc**
- Reads a file named **adobeupdate.dat** via **CreateFileA**, **GetFileSize**, **ReadFile**, and **CloseHandle**
- Marks an allocated memory buffer as executable using **VirtualProtect**

Next, we can load the sample on a disassembler like IDA and a debugger like x32dbg. For debugging, you can open AdobeInstall.exe and set a DLL breakpoint on hex.dll in order to debug it.

Note: AdobeInstall.exe loads hex.dll from the same directory and PlugX have taken advantage of this to load the malicious DLL as a form of anti-detection/anti-analysis technique.



Since there's only one export function in the DLL, it is fairly safe to assume that the sample only has one purpose – to load, decrypt, and execute the encrypted payload.

We can easily follow the export function in IDA and dertermine that the main function of the DLL lies in **sub_10001354**.

The first few lines of disassembly will show you "**\adobeupdate.dat**", "**kernel32**", and "**GetModuleFileNameA**" being initiated as stack strings. Usage of stack strings is a common anti-analysis and anti-detection technique employed by malware. This is typically used to prevent certain strings from showing up on basic string utilities.

```
0000000010001364 mov     [ebp+adobeupdate.dat], '\'
0000000010001368 mov     [ebp+var_37], 'a'
000000001000136C mov     [ebp+var_36], 'd'
0000000010001370 mov     [ebp+var_35], 'o'
0000000010001374 mov     [ebp+var_34], 'b'
0000000010001378 mov     [ebp+var_33], 'e'
000000001000137C mov     [ebp+var_32], 'u'
0000000010001380 mov     [ebp+var_31], 'p'
0000000010001384 mov     [ebp+var_30], 'd'
0000000010001388 mov     [ebp+var_2F], 'a'
000000001000138C mov     [ebp+var_2E], 't'
0000000010001390 mov     [ebp+var_2D], 'e'
0000000010001394 mov     [ebp+var_2C], '.'
0000000010001398 mov     [ebp+var_2B], 'd'
000000001000139C mov     [ebp+var_2A], 'a'
00000000100013A0 mov     [ebp+var_29], 't'
00000000100013A4 mov     [ebp+var_28], 0
00000000100013A8 mov     ax, [ebp+var_18]
00000000100013AC or      ax, 0DF6h
00000000100013B0 mov     [ebp+var_18], ax
00000000100013B4 mov     cx, [ebp+var_18]
00000000100013B8 imul    cx, 6591h
00000000100013BD mov     [ebp+var_18], cx
00000000100013C1 mov     [ebp+var_fullpath], 0
00000000100013C8 mov     ecx, 40h
00000000100013CD xor     eax, eax
00000000100013CF lea     edi, [ebp+var_168]
00000000100013D5 rep stosd
00000000100013D7 stosw
00000000100013D9 stosb
00000000100013DA mov     dx, [ebp+var_18]
00000000100013DE xor     dx, 8125h
00000000100013E3 mov     [ebp+var_18], dx
00000000100013E7 mov     ax, [ebp+var_18]
00000000100013EB and     ax, 5E61h
00000000100013EF mov     [ebp+var_18], ax
00000000100013F3 mov     [ebp+str_kernel32], 'k'
00000000100013F7 mov     [ebp+var_23], 'e'
00000000100013FB mov     [ebp+var_22], 'r'
00000000100013FF mov     [ebp+var_21], 'n'
0000000010001403 mov     [ebp+var_20], 'e'
0000000010001407 mov     [ebp+var_1F], 'l'
000000001000140B mov     [ebp+var_1E], '3'
000000001000140F mov     [ebp+var_1D], '2'
0000000010001413 mov     [ebp+var_1C], 0
```

After initiating the stack strings, the address of **GetModuleFileNameA** is dynamically resolved via **GetModuleHandleA** and **GetProcAddress**. Upon resolving its address in **kernel32**, **GetModuleFileNameA** is called.

```
00000000100014F2 lea     ecx, [ebp+str_GetModuleFileNameA] ; ecx = GetModuleFileNameA
00000000100014F5 push    ecx
00000000100014F6 lea     edx, [ebp+str_kernel32] ; edx = kernel32
00000000100014F9 push    edx
00000000100014FA call    ds:GetModuleHandleA ; Handle to kernel32 is obtained
0000000010001500 push    eax
0000000010001501 call    ds:GetProcAddress ; Address of GetModuleFileNameA in kernel32 is resolved
0000000010001507 mov     [ebp+var_GetModuleFileNameA], eax ; var_GetModuleFileNameA gets address of GetModuleFileNameA in kernel32
000000001000150D mov     ax, [ebp+var_18]
0000000010001511 xor     ax, 6E31h
0000000010001515 mov     [ebp+var_18], ax
0000000010001519 mov     cx, [ebp+var_18]
000000001000151D xor     cx, 9C92h
0000000010001522 mov     [ebp+var_18], cx
0000000010001526 push    104h
000000001000152B lea     edx, [ebp+var_fullpath]
0000000010001531 push    edx
0000000010001532 push    0
0000000010001534 call    [ebp+var_GetModuleFileNameA] ; call to GetModuleFileNameA
```

Running through it in **x32dbg** shows that **GetModuleFileNameA** returned the full path of the binary for the running process – which is "**C:\Users\user\Desktop\AdobeInstall.exe**".

```
        1000152B   lea edx,dword ptr ss:[ebp-16C]
        10001531   push edx
        10001532   push 0
        10001534   call dword ptr ss:[ebp-170]        call GetModuleFileNameA
EIP --> 1000153A   mov ax,word ptr ss:[ebp-18]
        1000153E   sub ax,4D3C
        10001542   mov word ptr ss:[ebp-18],ax

edx=0050017C
dword ptr [ebp-16C]=[0049F91C "C:\\Users\\user\\Desktop\\AdobeInstall.exe"]=555C3A43
```

Next, the full path and the character "\" is passed as a parameter in a function **sub_10001000**. This function splits the full path using "\" as delimiter and returns the address of the filename – which is "**\AdobeInstall.exe**".

```
        10001553   push 5C                             5c = '\'
        10001555   lea edx,dword ptr ss:[ebp-16C]
        1000155B   push edx
        1000155C   call hex.10001000                   function to get the filename portion from the full path
        10001561   add esp,8
EIP --> 10001564   mov dword ptr ss:[ebp-4],eax
        10001567   mov ax,word ptr ss:[ebp-18]
        1000156B   sar ax,1

dword ptr [ebp-4]=[0049FA84]=7EFDE000
eax=0049F931 "\\AdobeInstall.exe"

.text:10001564 hex.dll:$1564 #1564
```

A few lines after, the first character of "**\AdobeInstall.exe**" is replaced by **0x00**, thereby splitting the full path into two different strings in memory "**C:\Users\user\Desktop**" and "**AdobeInstall.exe**".

Next, the address of **lstrcatA** is also resolved dynamically using the same **GetModuleHandleA** and **GetProcAddress** technique mentioned earlier. **lstrcatA** is used to form the full path of the encrypted payload by concatenating "**C:\Users\user\Desktop**" and "**\adobeupdate.dat**".

```
00000000100015C8 lea     edx, [ebp+str_lstrcatA] ; edx = lstrcatA
00000000100015CB push    edx
00000000100015CC lea     eax, [ebp+str_kernel32] ; eax = kernel32
00000000100015CF push    eax
00000000100015D0 call    ds:GetModuleHandleA ; Handle to kernel32 is obtained
00000000100015D6 push    eax
00000000100015D7 call    ds:GetProcAddress ; Address of lstrcatA in kernel32 is resolved
00000000100015DD mov     [ebp+var_lstrcatA], eax
00000000100015E3 mov     cx, [ebp+var_18]
00000000100015E7 imul    cx, 853Eh
00000000100015EC mov     [ebp+var_18], cx
00000000100015F0 lea     edx, [ebp+adobeupdate.dat] ; edx = "\adobeupdate.dat"
00000000100015F3 push    edx
00000000100015F4 lea     eax, [ebp+var_fullpath] ; eax = "C:\Users\user\Desktop"
00000000100015FA push    eax
00000000100015FB call    [ebp+var_lstrcatA] ; call to lstrcatA
```

```
100015F0   lea edx,dword ptr ss:[ebp-38]     edx = \\adobeupdate.dat
100015F3   push edx
100015F4   lea eax,dword ptr ss:[ebp-16C]    eax = <path_of_running_process>
100015FA   push eax                          eax:"C:\\Users\\user\\Desktop\\adobeupdate.dat"
100015FB   call dword ptr ss:[ebp-184]       call lstrcartA
10001601   mov cx,word ptr ss:[ebp-18]
10001605   or cx,F565
```

Now that the full path of the encrypted payload is formed, a call to a function **sub_10001084** is made in order to read the file contents of the encrypted payload, get the file size, and load the contents into a buffer in memory.

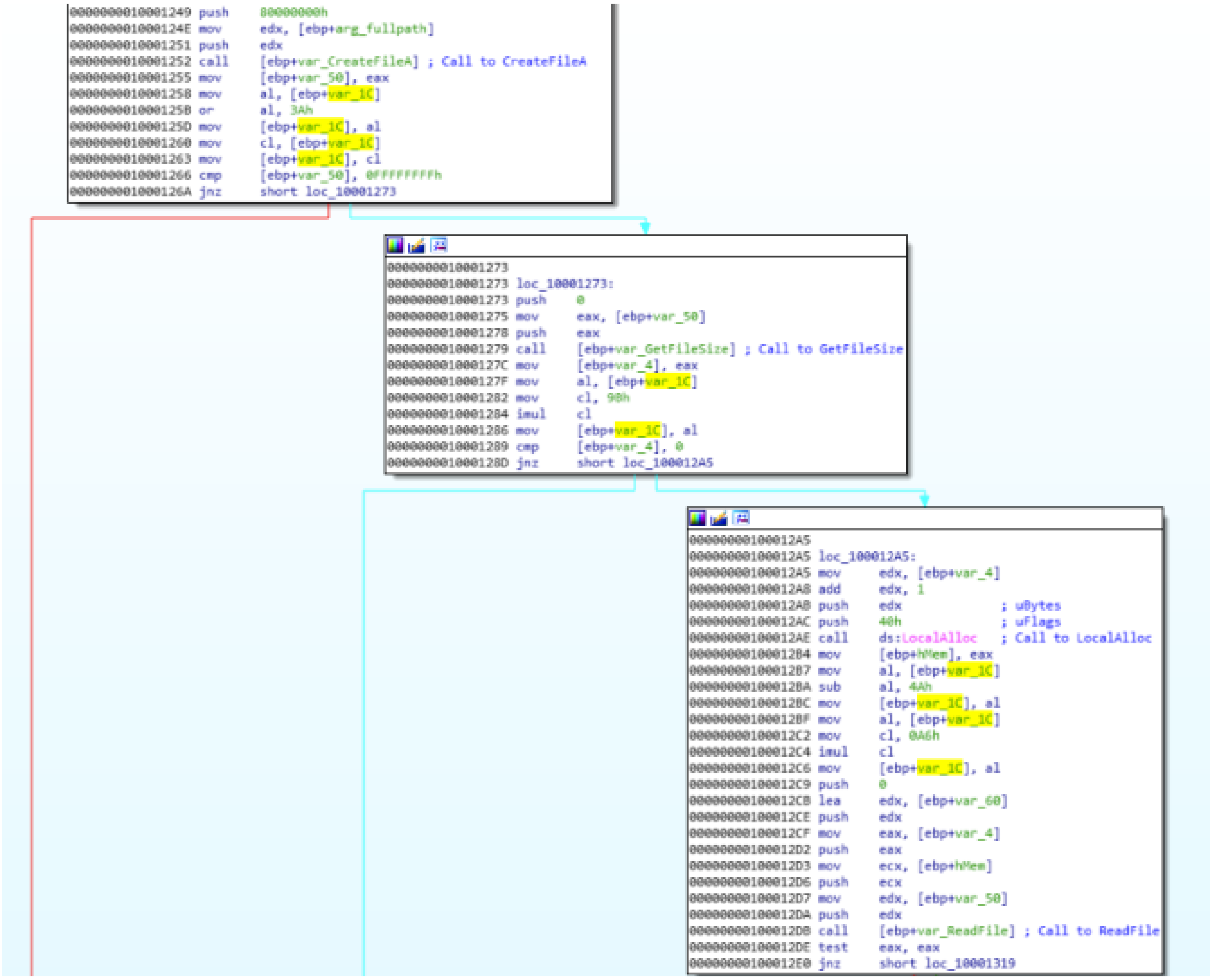
The following arguments are pushed into the stack before the function call is made:

```
0000000010001627 lea     ecx, [ebp+var_filesize] ; var_filesize will contain the file size of the encrypted payload after function call
000000001000162D push    ecx
000000001000162E lea     edx, [ebp+var_buffer] ; var_buffer will contain the address of the buffer containing the contents of the encrypted payload after function call
0000000010001634 push    edx
0000000010001635 lea     eax, [ebp+var_fullpath] ; var_fullpath contains the full path of the encrypted payload
000000001000163B push    eax
000000001000163C call    func_read_file
```

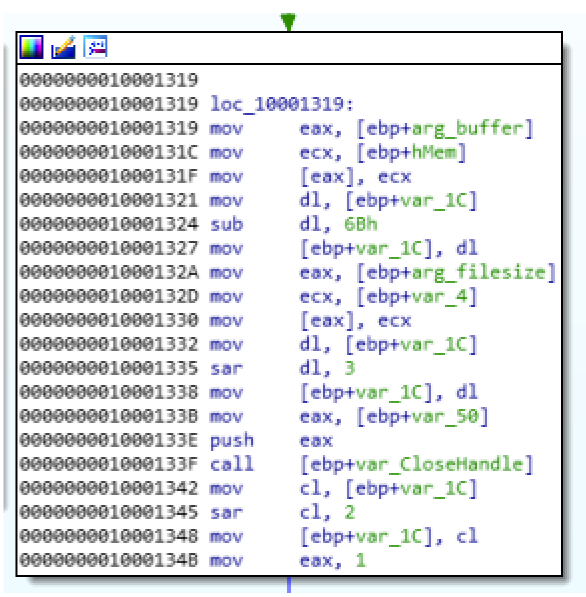
Looking closely at the disassembly of the function, we can see that same as before, the address of **CreateFileA**, **GetFileSize**, **ReadFile**, and **CloseHandle** are resolved dynamically using the same **GetModuleHandleA** + **GetProcAddress** technique.

After resolving the addresses of the functions, a call to each one is made in the following order:

- **CreateFileA** to open the encrypted payload
- **GetFileSize** to obtain the file size of the encrypted payload
- **LocalAlloc** to allocate a buffer in memory
- **ReadFile** to read the contents of the encrypted payload and place it in the buffer



After making these function calls, the address of the buffer containing the contents of the encrypted payload and the file size are stored in the arguments pushed to the stack earlier. Then finally, a call to **CloseHandle** is made and EIP returns to the main function.



Just a few lines of disassembly after, we can see some instructions assigning the address of the buffer to a new variable and that variable being passed as a parameter to **strlen**.

```
000000001000166E
000000001000166E loc_1000166E:           ; ecx = address of buffer
000000001000166E mov     ecx, [ebp+var_buffer]
0000000010001674 mov     [ebp+var_key], ecx ; address of buffer stored in a new variable
000000001000167A mov     dx, [ebp+var_18]
000000001000167E add     dx, 0FE1Fh
0000000010001683 mov     [ebp+var_18], dx
0000000010001687 mov     eax, [ebp+var_key] ; eax = address of buffer
000000001000168D push    eax
000000001000168E call    strlen          ; call to strlen which will return the string length and store it in eax
0000000010001693 add     esp, 4
0000000010001696 mov     [ebp+var_keysize], eax ; string length is stored in a new variable
```

These instructions essentially obtain the encryption key from the encrypted payload and determine its length via **strlen**. As you might remember, a string is an array of characters terminated with a NULL byte. So passing the address of the encrypted buffer to **strlen** will give us the length of the first string it sees.

```
0049FFE8 52 75 46 63 59 7A 51 4E 50 4A 00 1F 2F AE 63 59 RuFcYzQNPJ../®cY
0049FFF8 7A 51 15 02 0F 07 FE AA E2 9A 43 40 4E 50 B5 81 zQ....þªâ.C@NPµ.
004A0008 BC 85 63 19 7A 51 4E 50 4A 52 75 46 63 59 7A 51 ¼.c.zQNPJRuFcYzQ
004A0018 4E 50 4A 52 75 46 63 59 7A 51 4E 50 4A 52 75 46 NPJRuFcYzQNPJRuF
004A0028 63 59 7A 51 4E 50 4A AA 75 46 63 57 65 EB 40 50 cYzQNPJªuFcWeë@P
004A0038 FE 5B B8 67 DB 58 36 9C 6F 04 22 3B 06 66 13 2B þ[¸gÛX6.o.";.f.+
004A0048 15 36 3C 31 27 72 16 27 0D 37 15 25 6E 32 2F 72 .6<1'r.'.7.%n2/r
004A0058 07 33 0D 79 13 3F 6E 14 05 01 55 2B 0C 3D 1F 7F .3.y.?n...U+.=..
004A0068 43 5D 40 76 75 46 63 59 7A 51 4E AD CC 30 51 FF C]@vuFcYzQN.Ì0Qÿ
004A0078 84 55 0D E8 A9 5C 3D EB 92 4A 14 A6 CC BC 39 F1 .U.è©\=ë.J.¦Ì¼9ñ
```

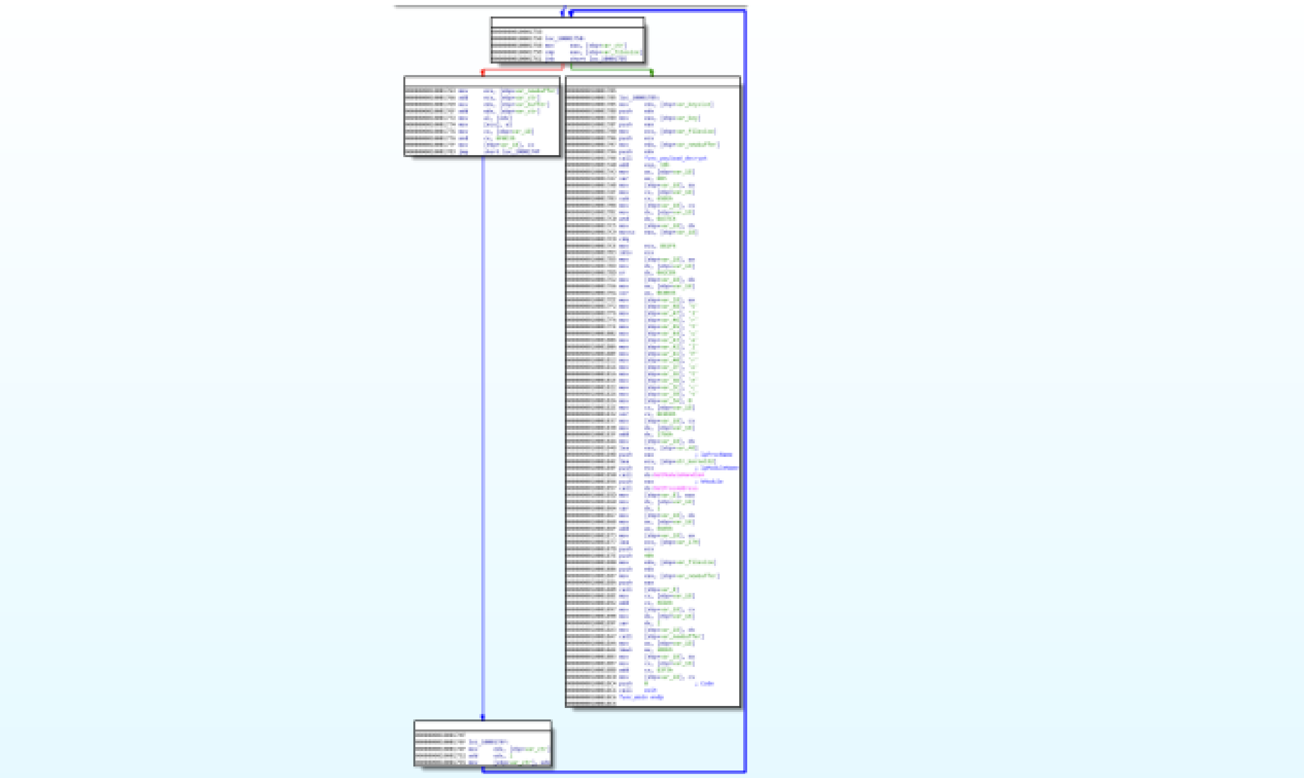
Later on, we will see how this string is used as a key for the decryption routine.

A few lines after, we can see a **sub** operation being performed on the file size using the determined key size to compute the file size of the payload without the key and the NULL byte.

```
00000000100016B2 mov     eax, [ebp+var_filesize]
00000000100016B8 sub     eax, [ebp+var_keysize]
00000000100016BB sub     eax, 1
00000000100016BE mov     [ebp+var_filesize], eax
```

Moving down further on the main function, we can immediately see that there is a loop before it proceeds to the final set of instructions.

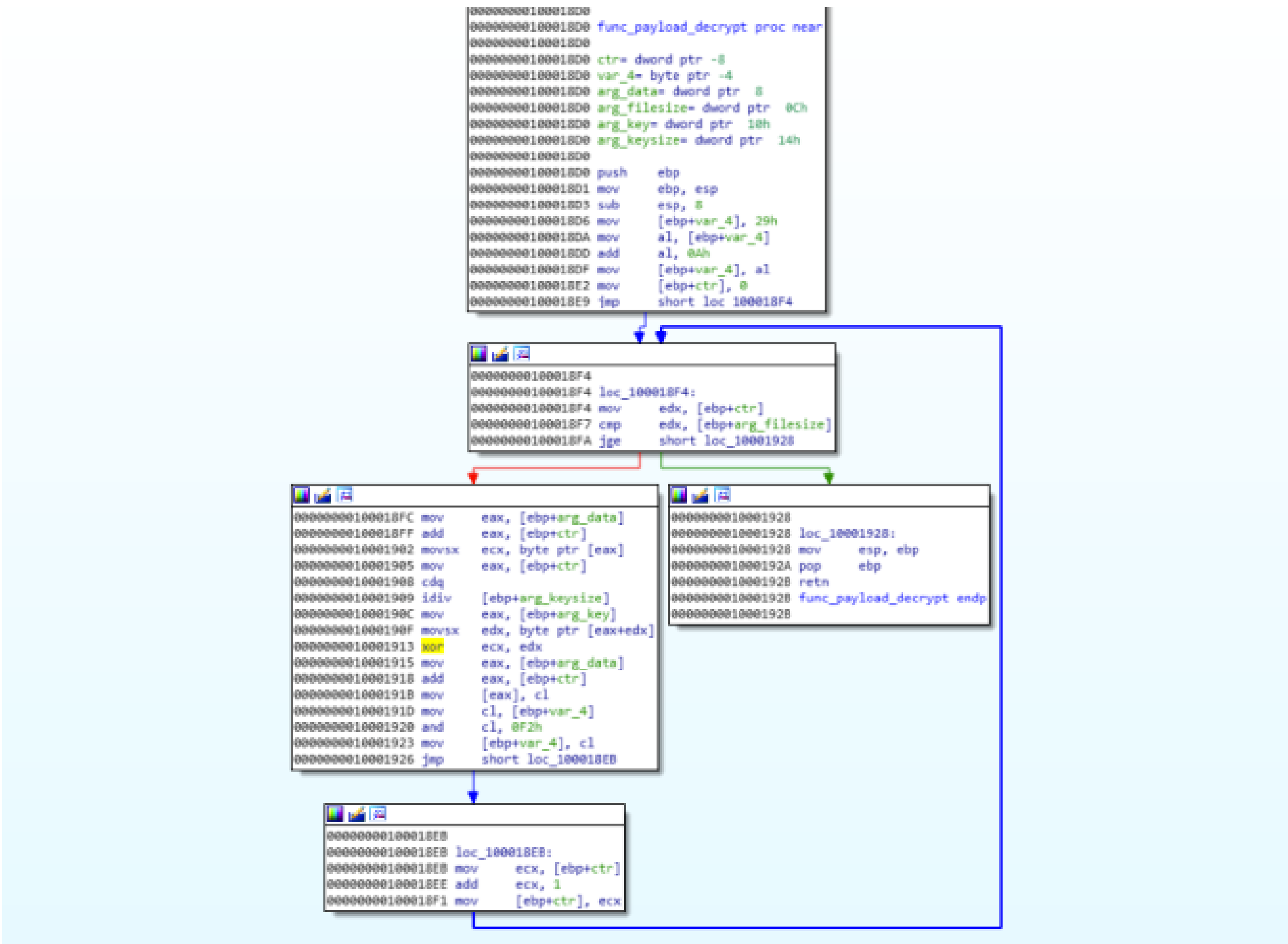
What this loop does is basically read the remaining bytes after the encryption key from the original buffer and copy it to a new buffer.



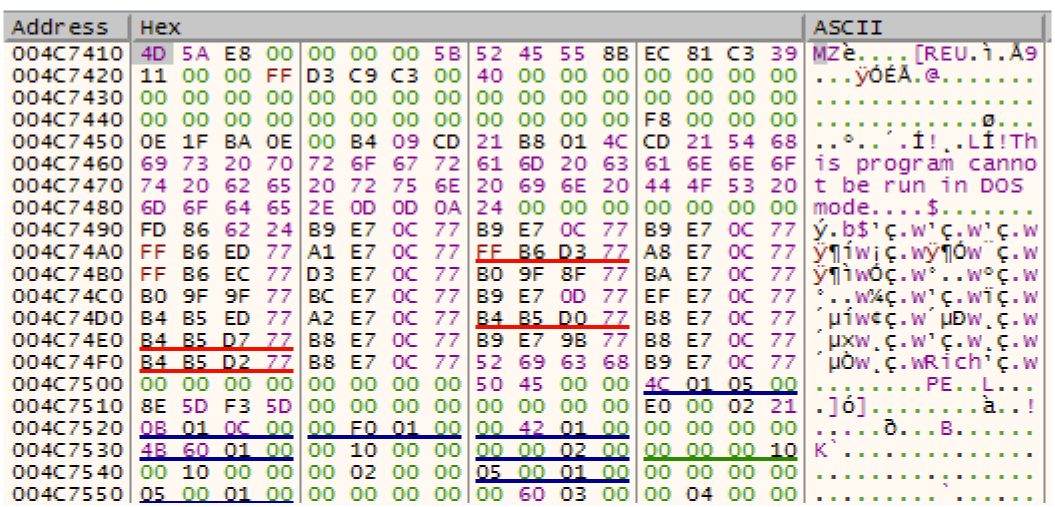
Right after the loop, the key size, key, file size, and new buffer is pushed to the stack and a call to a function **sub_100018D0** is made. This is the function that contains the algorithm to decrypt the encrypted payload.

```
0000000010001785 loc_10001785:
0000000010001785 mov     edx, [ebp+var_keysize]
0000000010001788 push    edx
0000000010001789 mov     eax, [ebp+var_key]
000000001000178F push    eax
0000000010001790 mov     ecx, [ebp+var_filesize]
0000000010001796 push    ecx
0000000010001797 mov     edx, [ebp+var_newbuffer]
000000001000179A push    edx
000000001000179B call    func_payload_decrypt
```

Looking closely at the disassembly of the function, we can immediately determine that the algorithm performs XOR using a multi-byte key.



Running this on a debugger, shows that the decrypted payload is a PE file.



Going back to the main function after the call to the decryption function, we can see the address of **VirtualProtect** being resolved using the same **GetModuleHandleA** + **GetProcAddress** technique.

Upon resolving the address, we can see a call to **VirtualProtect** to change the access protection of the buffer containing the decrypted payload to **0x40** (**PAGE_EXECUTE_READWRITE**).

Lastly, we can see a call to the address of the buffer to execute the decrypted payload.

```
0000000010001877 lea     ecx, [ebp+var_174]
000000001000187D push    ecx
000000001000187E push    40h             ; PAGE_EXECUTE_READWRITE
0000000010001880 mov     edx, [ebp+var_filesize] ; edx = file size
0000000010001886 push    edx
0000000010001887 mov     eax, [ebp+var_newbuffer] ; eax = address of buffer containing the decrypted payload
000000001000188A push    eax
000000001000188B call    [ebp+var_VirtualProtect] ; Call to VirtualProtect
000000001000188E mov     cx, [ebp+var_18]
0000000010001892 add     cx, 4E8Ah
0000000010001897 mov     [ebp+var_18], cx
000000001000189B mov     dx, [ebp+var_18]
000000001000189F sar     dx, 2
00000000100018A3 mov     [ebp+var_18], dx
00000000100018A7 call    [ebp+var_newbuffer] ; Execute the payload
```

..and there you have it folks, PlugX is now loaded to memory and executed on the system.

Automating the payload decryption

To make our lives easier, I created this quick-and-dirty python script to automatically decrypt payloads for this variant:

Hunting for encrypted payloads in VirusTotal

I'm also sharing this VT hunting YARA rule that I came up with to hunt for encrypted payloads associated with this variant. The rule is based on the filenames mentioned on the Avira report.

You may get some false positives on this one, but together with the python script above, this can be an effective approach to hunt for encrypted payloads that may otherwise go unnoticed on VT.

That's it guys! I really hope you learned something new today and as always, thank you for reading my blog!

PS: Stay tuned for the next post on this series where we'll reverse engineer some interesting parts of the PlugX payload itself. Cheers!

Tags:MustangPanda, PlugX, Reverse Engineering