

# [RE027] China-based APT Mustang Panda might still have continued their attack activities against organizations in Vietnam

blog.vincss.net/2022/05/re027-china-based-apt-mustang-panda-might-have-still-continued-their-attack-activities-against-organizations-in-Vietnam.html



## 1. Executive Summary

At VinCSS, through continuous cyber security monitoring, hunting malware samples and evaluating them to determine the potential risks, especially malware samples targeting Vietnam. Recently, during hunting on [VirusTotal's](#) platform and performing scan for specific byte patterns related to the **Mustang Panda (PlugX)**, we discovered a series of malware samples, suspected to be relevant to APT Mustang Panda, that was uploaded from Vietnam.

All of these samples share the same name as **“log.dll”** and have a rather low detection rate.

Files	Detections	Size	First seen	Last seen	Submitters
088C990A5A0338F448C7280981F8ED2480541883282C2135887E89456897721 log.dll	11 / 68	864.00 KB	2022-05-07 01:33:18	2022-05-07 01:33:18	1
84893F360AC388A68F89E4843A507896888852F7647C25143DEE8E58EC980C5D @4893f360ac388a68f89e4843a507896888852f7647c25143de8e58ec980c5d.sample	9 / 68	103.00 KB	2022-05-05 12:42:34	2022-05-05 17:58:50	2
3171285C48A463689379e80F538C48A5C988FE3280DE18CF80268B122576F4E log.dll.sc	13 / 67	377.50 KB	2022-04-25 14:04:36	2022-04-25 14:04:36	1
6848282C8E5C97C7E8A74412E1F88E8A3C88A888E340C6885188947C133A9 log.dll	13 / 69	52.00 KB	2022-04-12 02:36:42	2022-04-12 02:36:42	1
D428E84F46A62561CE188E827307C8E48188F8EE3EFD82ECC2198178C803A log.dll	10 / 55	576.50 KB	2022-03-26 13:16:05	2022-03-26 13:16:05	1

Based on the above information, we infer that there is a possibility that malware has been infected in certain orgs in Vietnam, so we decided to analyze these malware samples. During analysis, based on the detected indicators, we continue to investigate and set the scenario of

the attack campaign.

A general overview of the execution flow demonstrated as follow:



Our blog includes:

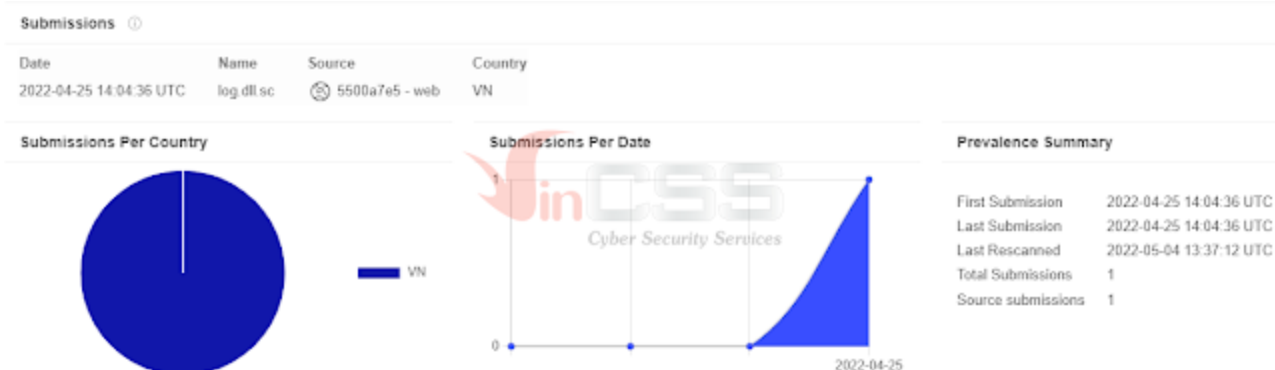
- Technical analysis of the **log.dll** file.
- Technical analysis of shellcode decrypted from **log.dat**.
- Analyze **PlugX DLL** as well as decrypt PlugX configuration information.

## 2. Analyze the log.dll

In the list of hunted samples above, we choose the one with hash:

[3171285c4a846368937968bf53bc48ae5c980fe32b0de10cf0226b9122576f4e](#)

This sample was submitted to VirusTotal from **Vietnam** on **2022-04-25 14:04:36 UTC**



The information from the Rich Header suggests that it is likely compiled with **Visual Studio 2012/2013**:

product-id (8)	build-id (4)
<a href="#">Implib1100</a>	Visual Studio 2012 - 11.0
<a href="#">Import</a>	Visual Studio
<a href="#">Utc1800 CPP</a>	Visual Studio 2013 - 12.0
<a href="#">Masm1200</a>	Visual Studio 2013 - 12.0
<a href="#">Utc1800 C</a>	Visual Studio 2013 - 12.0
<a href="#">Import (old)</a>	Visual Studio
<a href="#">Export1200</a>	Visual Studio 2013 - 12.0 RTM
<a href="#">Linker1200</a>	Visual Studio 2013 - 12.0 RTM

By checking the sections information, we can see that it is packed or the code is obfuscated:

Nr	Virtual offset	Virtual size	RAW Data offset	RAW size	Flags	Name	First bytes (hex)	First Ascii 20h bytes	sect. Stats
01	ep: 00001000	000577C6	00000400	00057800	60000020	.text	55 53 57 56 83 ...	USWV 0 □□1 D...	Strong Packed - 2.2743 % ZERO
02	im: 00059000	000046F4	00057C00	00004800	40000040	.rdata	20 D2 05 00 34 ...	□ 4 □ F □ T □ ...	Very not packed - 43.6306 % ZERO
03	0005E000	00002FA0	0005C400	00001200	C0000040	.data	4E E6 40 BB B1 ...	N @ □ D ...	Very not packed - 64.3012 % ZERO
04	00061000	00000ED4	0005D600	00001000	42000040	.reloc	00 10 00 00 0C ...	□ * □0□0 ...	Not packed - 16.6992 % ZERO

Sample has the original name **ljAt.dll**, and it exports two functions **LogFree** and **LogInit**:

5d7db86620f99cddb2002d8f7884761a2\_log\_dll.bin

Offset	Name	Value	Meaning
5BC90	Characteristics	0	
5BC94	TimeStamp	622DA6ED	Sunday, 13.03.2022 08:10:21 UTC
5BC98	MajorVersion	0	
5BC9A	MinorVersion	0	
5BC9C	Name	5D0CC	LjAt.dll
5BCA0	Base	1	
5BCA4	NumberOfFunctions	2	
5BCA8	NumberOfNames	2	
5BCAC	AddressOfFunctions	5D0B8	
5BCB0	AddressOfNames	5D0C0	
5BCB4	AddressOfNameOrdinals	5D0C8	

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
5BCB8	1	1000	5D0D5	LogFree	
5CBC	2	4E5E0	5D0DD	LogInit	

Load sample into IDA, analyze the code of the two functions above:

### LogFree function:

Looking at this function, it can be seen that its code has been completely obfuscated by Obfuscator-LLVM, using the Control Flow Flattening technique:

After further analysis, I found that this function has no special task.

### LogInit function:

This function will call the **LogInit\_0** function:



```

public LoginInit
proc near
; DATA XREF: .rdata:off_1005D0B840
jmp LoginInit_0 ; TAGS: ['Enum', 'FileWIN']
endp
23 calls, 1 strings
calls:
- call dword ptr[eax]
- call ds:CloseHandle ; call CloseHandle
- call ds:CreateFileA ; call CreateFileA to open file
- call ds:ReadFile ; call ReadFile to read file content
- call _strncmp ; call _strcmp to compare string
- 2 call dword ptr[eax] ; exec decrypted payload/shellcode
- call ds:CloseHandle ; call CloseHandle
- call ds>DeleteFileA ; call DeleteFileA
- call ds:CloseHandle ; call CloseHandle
- call ds>DeleteFileA ; call DeleteFileA
- 1 call f_read_content_of_log_dat_file_to_buf ; call f_read_content_of_log_dat_file
- call ds:GetModuleHandleA ; call GetModuleHandleA to retrieve kernel32.dll handle
- call ds:GetProcAddress ; retrieve api address
- call eax ; call API func
- call ds:ExpandEnvironmentStringsA ; call ExpandEnvironmentStringsA
- call ds:CreateFileA ; call CreateFileA for retrieving handle to create tmp file
- call _strlen ; call _strlen
- call ds:WriteFile ; call WriteFile to write content to file
- call ds:ExpandEnvironmentStringsA ; call ExpandEnvironmentStringsA
- call ds:CreateFileA ; call CreateFileA
- call _strlen ; call _strlen
- call ds:WriteFile ; call WriteFile
- call __security_check_cookie(x)
strings:
- kernel32

```

f\_read\_content\_of\_log\_dat\_file\_to\_buf's code is also completely obfuscated:

```

0910 {
0915 break;
0916 }
0917 LABEL_17:
0918 if ( control_var ≤ 0x28E893A )
0919 {
0920 goto LABEL_18;
0921 }
0922 }
0923 kernel32_handle = GetModuleHandleW(eszKernel32);
0924 v505 = dword_1005FEA8 + (dword_1005FEA8 - 1);
0925 log_dat_content_id = ~v505;
0926 v506 = ((dword_1005FEA8 + (dword_1005FEA8 - 1)) & 0x3E4D8491 | ~v505 & 0x0182796E
0927 v507 = (((~v506 & 0x278BCC0 | v506 & 0xF0B043F) * 0x103B00) & 0x4C90311E | (~v5
0928 v508 = ~v505 | (~v505 & 0xA877B9FE | (dword_1005FEA8 + (dword_1005FEA8 - 1)) & 0
0929 v509 = v507 & 0x71069B05 | ~v507 & 0x3C29642A;
0930 v510 = ((v508 | v507) & 0x378784D2 | ~v508 | v507) & 0xC828492D) * (~v509 & v5
0931 v511 = v510 & (v510 * 0x60C0D028) & ~v510 & 0x60D002B | ~v510 & 0x60D002B & v5
0932 v512 = ~v511 & 0x59FAD090 | v511 & 0x46854BDF;
0933 v513 = ~v512 & v512 & ~v512 | ~v512 * ~v512 & v512;
0934 v514 = ~v513;
0935 v515 = v513 & 0x1486804F & v510 & (v513 * 0xEB07780) | v514 & (v513 * 0xEB07780
0936 v516 = (~v515 & 0x269B2EBC | v515 & 0xD964C1B3) * 0xC0D244FD;
0937 v517 = (~v514 & 0xFFFFFFFF | v513 * 1) & v516 | v516 * ~v514 & 0xFFFFFFFF | v51
0938 v518 = v517 & (v517 * 0x2D40B4AF) & ~v517 & 0x2D40B4AF | ~v517 & 0x2D40B4AF & v5
0939 v519 = v506 & ((dword_1005FEA8 + (dword_1005FEA8 - 1)) * 0x11D587DC);
0940 v520 = (~v518 & 0xD25F4850 | v518 & 0x2D40B4AF) & 0x6A99A520 | (~v518 & 0xD25F4
0941 v521 = (v520 * 0x2218A560) & 0xA25EEF51 | (v520 * 0x1B29100E) & 0x30A110AE;
0942 v522 = (v521 * 0x0011800E) & 0xBF4F520F | (v521 * 0x2810AD50) & 0x7868ADF9;
0943 v523 = (~v519 & ~v505 & 0x11D587DC | ~v500 & 0x11D587DC * v519) & 0x70D57946 | (
0944 v524 = (v523 * 0x93FF314D) & 0xFFFFFFFF;
0945 v525 = (v523 * 0x4C00CE8A) & 0xFFFFFFFF | (v523 * 0x93FF314D) & 1;
0946 v526 = v525 & v524;
0947 v527 = v524 * v525;
0948 v528 = v522 * 0x8F4F520F;

```

The major task of this function as the following:

- Call the **GetModuleHandleW** function to retrieve the handle of **kernel32.dll**.
- Call the **GetProcAddress** function to get the addresses of the APIs: **VirtualAlloc**, **GetModuleFileNameA**, **CreateFileA**, **ReadFile**.
- Use the above APIs to retrieve the path to the **log.dat** file and read the contents of this file into the allocated memory.

```

call f_read_content_of_log.dat_file_to_buf ; call f_read_content_of_log.d
mov ecx, [ebp+documentad_challenge]
test eax, eax
mov edx, 11
mov [ecx], calls:
mov eax, 7A
cmovz eax, edx
cmp eax, 0E
jg loc_1002
call ds:GetModuleHandleW ; call GetModuleHandleW to retrieve handle of kernel32.dll
call ds:GetProcAddress ; retrieve VirtualAlloc addr
call ds:GetProcAddress ; retrieve GetModuleFileNameA
call ds:GetProcAddress ; retrieve CreateFileA addr
call ds:GetProcAddress ; retrieve ReadFile addr
call [esp+1FCh+GetModuleFileNameA] ; call GetModuleFileNameA to retrieve full path of module that load malware dll
call f_strstr ; Returns a pointer to the first occurrence of a search string in a string.
call oax ; call CreateFileA for open file but not retrieve file handle
call ds:CloseHandle ; call CloseHandle to release handle to log.dat file
call oax ; call ReadFile for reading log.dat content to allocated buffer
call eax ; call CreateFileA to retrieve handle to log.dat file
call ds:GetFileSize ; call GetFileSize to retrieve size of log.dat
call oax ; call VirtualAlloc to allocate buffer with buf's size equal size of log.dat
call ds:lstrcatA ; call lstrcatA to build full path to log.dat
call _security_check_cookie(x)

```

Decode the contents of **log.dat** into shellcode so that this shellcode is then executed by the call from the **LogInit\_0** function.

```

138003 LOGBYTE(v4929) = v4929 & BYTE1(v4929) | BYTE1(v4929) * v4929;
138004 BYTE1(v5065) = ~(_BYTE)v5065 & 0x1A;
138005 BYTE1(v4939) = ((v5065 & 0x79 | ~(_BYTE)v5065 & 0x30) * 0x36) & ((v5065 & 0
138006 LOGBYTE(v5356) = v5064;
138007 LOGBYTE(v5064) = ~BYTE1(v5064) | v5064;
138008 LOGBYTE(v5065) = (v5259 & 0x15 | BYTE1(v5065)) * (~BYTE1(v5064) & 0x45 | BYTE1
138009 BYTE1(v5065) = (~BYTE1(v4929) & 0x8C | BYTE1(v4929) & 0x71) * (~(_BYTE)v50
138010 BYTE1(v5065) = BYTE1(v5065) & ~(_BYTE)v5064 | ~(_BYTE)v5064 | (BYTE1(v506
138011 LOGBYTE(v5065) = ~BYTE1(v5065);
138012 LOG_INIT_CONTENTS[16x] = ((v5065 | v4929) & 0x35 | ~v5065 | v4929) & 0xA4) &
138013 v5065 * idx + 1;
138014 control_var_1 = 0x92E699EC;
138015 }
138016 }
138017 }
138018 }
138019 }
138020 }
138021 }
138022 }
138023 }
138024 }
138025 }

```

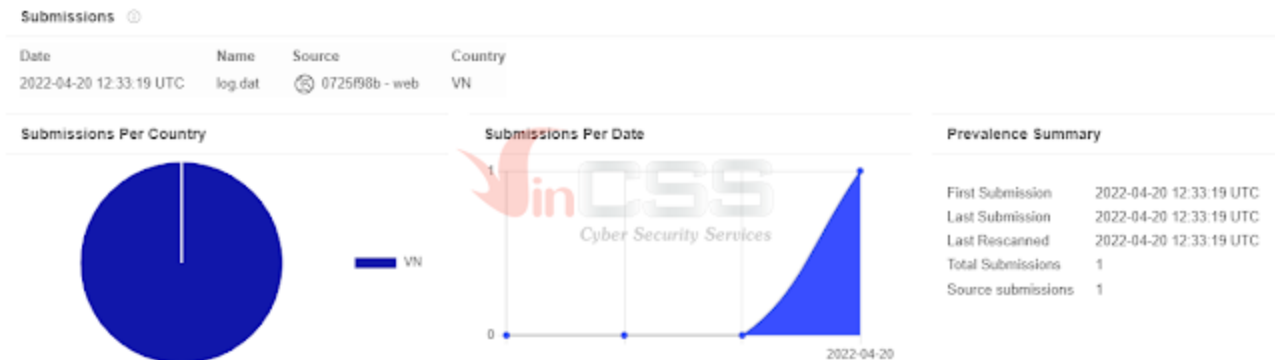
### 3. Shellcode analysis

Based on the information analyzed above, we know that the **log.dll** file will read the content from the **log.dat** file and decrypt it into shellcode for further execution. Relying on this indicator, we continue to hunt **log.dat** file on VirusTotal which restrict the scope of submission source from Vietnam.

The results are following:

Files	Detections	Size	First seen	Last seen	Submitters
3248DC1C05c629280f168120E22f681A7642485a28882c4715fE28f9c10E8B	0 / 57	194.66 KB	2022-05-07 01:32:51	2022-05-07 01:32:51	1
R248038EAA344754E27880F7608A2F260c633a608fE7718c2b4b7330FARCS	2 / 59	189.23 KB	2022-05-05 12:44:31	2022-05-05 12:44:31	1
BE8E278244371A51F88891EE246EF3475787132822C850A8C9F18017538C8	0 / 57	194.66 KB	2022-04-25 14:07:46	2022-04-25 14:07:46	1
2DE77804E28D9B843A826f194389c2605cfc17fd2fafde1b8eb2f819fc6c0c84	0 / 57	194.66 KB	2022-04-20 12:33:19	2022-04-20 12:33:19	1

With the above results, at the time of analysis, we selected the **log.dat** file (2de77804e28d9b843a826f194389c2605cfc17fd2fafde1b8eb2f819fc6c0c84) was submitted to VirusTotal on **2022-04-20 12:33:19 UTC** (5 days before the above **log.dll** file).



Debugging and dump the decrypted shellcode look like this:

```

log_dat_sc.bin
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
00000000 77 06 81 EE 00 00 00 00 80 C5 00 45 4D 66 83 EE 0..i...eA.KMef1
00000010 00 73 07 55 7C 03 C0 C2 70 5D 8D 12 55 66 83 C9 .s.U].AAp]..UefE
00000020 00 5D 7D 05 0D 00 00 00 00 E8 00 00 00 00 57 BF .j].....e...Wz
00000030 44 49 00 00 5F F9 58 50 50 48 58 58 57 66 BF 9D DI...aXPPHXKwz.
00000040 00 5F 83 E8 05 0B C0 FC 68 0C 15 00 00 0D 00 00 _fe..Agh.....
00000050 00 00 6A D5 83 C4 04 57 7C 06 81 FF BF 60 00 00 ..jOfA.W].yZ'.
00000060 5F 8B F6 F9 E8 0C 15 00 00 5E BE 68 CA EA 0A DC <80e...^MhEe.U
00000070 7E B4 B4 B4 B4 B4 4B 4B 4B 4B B4 B4 B4 B4 B4 B4 -''''KKKK''''''
00000080 B4 B4 B4 B4 B4 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B -''''KKKKKKKKKK
00000090 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B B4 B4 B4 B4 B4 B4 KKKKKKKK''''''''
000000A0 BE B4 B4 B4 B4 B5 75 85 85 85 85 85 85 85 85 85 N''''puppppppppp
000000B0 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
000000C0 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
000000D0 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
000000E0 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
000000F0 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000100 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000110 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000120 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000130 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000140 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000150 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000160 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000170 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
00000180 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 ppppppppppppppppp
  
```

I use two tools, FLOSS and sctdbg to get an overview of this shellcode. The results can be seen in the screenshots below:

```

FLOSS static Unicode strings
FLOSS decoded 2 strings
(EAA
&EAA
FLOSS extracted 8 stackstrings
VirtualProtect
VirtualAlloc
ExitThread
memcpy
ntdll
LoadLibraryA
VirtualFree
RtlDecompressBuffer
  
```



The image shows two windows. The left window is a command prompt running a debugger on a file named 'log\_dat\_sc.bin'. It displays the results of a memory dump and execution steps. The right window is the 'scDbg - libemu Shellcode Logger Launch Interface', which shows the shellcode file path and various options for logging and debugging.

```

C:\WINDOWS\SYSTEM32\cmd.exe
Loaded 30aa8 bytes from file C:\Users\ADMINI~1\Desktop\log_dat_sc.bin
Memory monitor enabled..
Initialization Complete..
Dump mode Active...
Max Steps: -1
Using base offset: 0x401000

430e80 GetProcAddress(LoadLibraryA)
430fb2 GetProcAddress(VirtualAlloc)
4310ca GetProcAddress(VirtualFree)
431145 GetProcAddress(VirtualProtect)
43124f GetProcAddress(ExitThread)
43128a LoadLibraryA(ntdll)
4313f3 GetProcAddress(RtlDecompressBuffer)
431436 GetProcAddress(memcpy)
4314dc VirtualAlloc(base=0, sz=2e552) = 600000
43154d VirtualAlloc(base=0, sz=4c000) = 62f000
431592 RtlDecompressBuffer(fmat=2, ubuf=62f000, sz=4c000)
0 emu_parse no memory found at 0x0

0 ???? No memory At Address step: 2859730
eax=e ecx=4c000 edx=62f000 ebx=0
esp=12fe04 ebp=12fff0 esi=0 edi=0

Stepcount 2859730
Primary memory: Reading 0x30aa8 bytes from 0x401000
Scanning for changes...
No changes found in primary memory, dump not created.
Dumping 2 runtime memory allocations..
Alloc 000000 (2e552 bytes) dumped successfully to disk 0000
Alloc 62f000 (4c000 bytes) dumped successfully to disk 0000
  
```

With the results obtained above, it can be seen that this shellcode will perform memory allocation and then call the **RtlDecompressBuffer** function to decompress the data with the compression format is **COMPRESSION\_FORMAT\_LZNT1**.

By using IDA to analyze this shellcode, its main task is to decompress a Dll into memory and call the exported function of this Dll to execute. The function that does this task is named **f\_load\_dll\_from\_memory**:

The image shows the IDA Pro interface with assembly code for the **f\_load\_dll\_from\_memory** function. The code includes instructions for pushing the shellcode size, setting up registers, and calling various system functions like **VirtualAlloc**, **RtlDecompressBuffer**, and **GetProcAddress** to load and execute a DLL.

```

.text:00431AE4 ; int __usercall sub_431AE4@eax<eax>(int a1@<eax>)
.text:00431AE4 sub_431AE4 proc near ; CODE XREF: sub_403575+101p
.text:00431AE4 push 30AA8h ; shellcode size
.text:00431AE9 push eax ; ptr_call_addr
.text:00431AEA rol si, 20h
.text:00431AEE stc
.text:00431AEF stc
.text:00431AF0 test ah, ah
.text:00431AF2 call f_load_dll_from_memory
.text:00431AF7 retn
.text:00431AF7 sub_431AE4 endp ; sp-analysis failed
.text:00431AF7

1 // positive sp value has been detected, the output may be wrong!
2 int __usercall sub_431AE4@eax<eax>(int a1@<eax>)
3 {
4   _DWORD v2; // [esp-10h] [ebp-10h]
5   int v3; // [esp-Ch] [ebp-Ch]
6   int v4; // [esp-8h] [ebp-8h]
7   int v5; // [esp-4h] [ebp-4h]
8
9   return f_load_dll_from_memory(a1, 0x30AA8, v2, v3, v4, v5);
10 }

calls:
- call [ebp+GetProcAddress]
- call [ebp+GetProcAddress]
- call [ebp+GetProcAddress]
- call [ebp+GetProcAddress]
- call [ebp+GetProcAddress]
- call [ebp+LoadLibraryA]
- call [ebp+GetProcAddress]
- call [ebp+GetProcAddress]
- call [ebp+VirtualAlloc]
- call [ebp+VirtualAlloc]
- call [ebp+RtlDecompressBuffer]
- call [ebp+VirtualAlloc]
- call [ebp+memcpy]
- call [ebp+LoadLibraryA]
- call [ebp+GetProcAddress]
- call [ebp+GetProcAddress]
- call [ebp+VirtualProtect]
- call ecx ; call to DllEntryPoint
- call [ebp+exported_func] ; call to PlugX exported function
- call [ebp+VirtualFree]
- call [ebp+VirtualFree]
  
```

The code in this function will first get the base address of **kernel32.dll** based on the pre-calculated hash value is **0x6A4ABC5B**. This hash value has also been mentioned by us in [this analysis](#).

```

kernel32_base_addr = 0;
GetProcAddress = 0;
pLdr = NtCurrentPeb()->Ldr;
for ( ldr_entry = pLdr->InMemoryOrderModuleList.Flink; ldr_entry; ldr_entry = ADJ(ldr_entry)->InMemoryOrderLinks.Flink )
{
    wszDllName = ADJ(ldr_entry)->BaseDllName.Buffer;
    dll_name_length = ADJ(ldr_entry)->BaseDllName.Length;
    calced_hash = 0;
    do
    {
        calced_hash = _ROR4_(calced_hash, 13);
        if ( *wszDllName < 'a' )
            calced_hash += *wszDllName; // calced_hash + letter
        else
            calced_hash = calced_hash + *wszDllName - 0x20; // calced_hash + upper_letter
        wszDllName = (wszDllName + 1);
        --dll_name_length;
    }
    while ( dll_name_length );
    if ( calced_hash == 0x6A4A3C5B ) // kernel32.dll's hash
    {
        kernel32_base_addr = ADJ(ldr_entry)->DllBase;
        break;
    }
}
if ( !kernel32_base_addr )
    return 1;

```

```

python .\brute_force_dll_name.py
Found dll kernel32.dll of 0x6a4abc5b
Found dll ntdll.dll of 0x3cfa685d

```

Next it will retrieve the address of **GetProcAddress**:

```

for ( i = 0; i < export_dir_va->NumberOfNames; ++i )
{
    szAPIName = kernel32_base_addr + pFuncsNamesAddr[i];
    if ( *szAPIName == 'G'
        && szAPIName[1] == 'e'
        && szAPIName[2] == 't'
        && szAPIName[3] == 'p'
        && szAPIName[4] == 'r'
        && szAPIName[5] == 'o'
        && szAPIName[6] == 'c'
        && szAPIName[7] == 'A'
        && szAPIName[8] == 'd'
        && szAPIName[9] == 'd' )
    {
        GetProcAddress = (kernel32_base_addr
            + *(kernel32_base_addr
                + 4 * *(kernel32_base_addr + 2 * i + export_dir_va->AddressOfNameOrdinals)
                + export_dir_va->AddressOfFunctions));
        break;
    }
}
if ( !GetProcAddress )
    return 2;

```

By using the stackstring technique, the shellcode constructs the names of the APIs and gets the addresses of the following API functions:



```

signature = *ptr_enc_compressed_dll_addr; // ptr_enc_compressed_dll_addr = 0x1592 (offset on disk)
// signature = 0xC7EA9B1C
xor_key = signature - 0x7979A9AA; // xor_key = 0x4E70F172
// dd 0B598E96Eh
// dd 0C7EA9B1Ch → signature
// dd 0004C000h → uncompressed_size
// dd 2E542h → compressed_size;
for ( j = 0; j < 0x10; ++j )
    config_info_buf[j] = xor_key ^ ptr_enc_compressed_dll_addr[j]; // xor_key = 0x72
if ( signature ≠ computed_signature )
    return 0xA;
dwSize = computed_compressed_size + 0x10; // dwSize = 0x2E552
compressed_buf = VirtualAlloc(0, computed_compressed_size + 0x10, MEM_COMMIT, PAGE_READWRITE);
if ( !compressed_buf )
    return 0xB;
xor_key = signature - 0x7979A9AA;
// fill compressed buffer
for ( k = 0; k < dwSize; ++k )
    *(&compressed_buf→decoded_buffer + k) = xor_key ^ ptr_enc_compressed_dll_addr[k];
// uncompressed_buf_size = 0x4C000
uncompressed_buf = VirtualAlloc(0, uncompressed_buf_size, MEM_COMMIT, PAGE_READWRITE);
if ( !uncompressed_buf )
    return 0xC;
final_uncompressed_size = 0;
// decompress dll payload to memory
if ( RtlDecompressBuffer(
    COMPRESSION_FORMAT_LZNT1,
    uncompressed_buf,
    uncompressed_buf_size, // 0x4C000
    &compressed_buf→compressed_buf, // 0x2E542
    compressed_buf→compressed_size, // 0x2E542
    &final_uncompressed_size ) )
{
    return 0xD;
}
if ( uncompressed_buf_size ≠ final_uncompressed_size )

```

Based on the above analysis results, it is easy to get the extracted Dll file (however, the file header information was destroyed):

```

decompressed_dll_4C000.dump
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
00000000 6C 41 76 62 42 48 6A 44 4C 75 4D 42 54 6B 57 57 1AvbBHjDLuMBTrkWW
00000010 45 78 5A 45 4F 6F 54 65 79 70 75 44 63 4B 4E 45 ExZEOoTeypuDcKNE
00000020 74 6C 73 50 61 48 48 78 69 5A 7A 4A 6E 4E 6E 74 t1sPaHxiZzJnNnt
00000030 69 49 46 4C 42 43 4F 59 50 58 54 00 E0 00 00 00 iIFLBCOYPXT.à...
00000040 78 43 52 55 6A 44 62 52 4E 4C 58 4A 76 73 47 79 xCRUjDbRNLXJvsGy
00000050 75 4F 77 76 55 59 55 76 76 46 58 5A 77 7A 42 55 uOwvUYUvVFXZwzBU
00000060 70 6F 4B 48 4D 75 50 46 45 45 67 45 73 67 71 61 poKHMuPFEEgEsgqa
00000070 56 69 75 4C 6E 6C 53 52 74 69 51 72 7A 63 4C 49 ViuLnlSRtiQrzcLI
00000080 69 7A 61 55 6E 5A 6A 78 79 45 51 62 6D 76 42 69 izaUnZjxyEQbmVBi
00000090 53 4F 67 72 75 55 64 46 4E 6C 78 78 50 6F 50 64 SOgruUdFNlxxPoPd
000000A0 75 72 75 68 61 69 67 6F 61 58 52 71 4E 59 63 6C uruhaigoeXRqNYcl
000000B0 75 4E 58 72 4C 44 42 69 48 49 65 67 56 43 75 48 uNXrLDBiHtegVCuH
000000C0 77 73 77 48 68 53 6B 45 72 4B 77 68 55 6C 52 78 wswHhSkErKwhUlRx
000000D0 4C 44 6B 46 42 64 59 79 4C 6E 79 72 50 52 71 54 LDkFBdYyLnyrPrqT
000000E0 53 6C 00 00 4C 01 03 00 30 83 1E 53 00 00 00 00 Sl...L...Of.S...
000000F0 00 00 00 00 E0 00 02 21 0B 01 0C 00 00 00 00 00 .....à..!.....
00000100 00 3C 00 00 00 00 00 00 B0 81 00 00 00 10 00 00 <.....>.....
00000110 00 10 00 00 00 00 00 10 00 10 00 00 00 02 00 00 .....>.....
00000120 05 00 01 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000130 00 E0 04 00 00 00 00 00 00 00 00 00 00 40 01 ..à.....@.
00000140 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00 .....
00000150 00 00 00 00 10 00 00 00 60 8F 04 00 45 00 00 00 .....E...
00000160 30 51 04 00 78 00 00 00 00 00 00 00 00 00 00 00 0'..x.....
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000180 00 A0 04 00 0C 33 00 00 00 00 00 00 00 00 00 00 .....3.....
00000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001A0 00 00 00 00 00 00 00 00 50 7A 00 00 40 00 00 00 .....Pz..@...
000001B0 00 00 00 00 00 00 00 00 00 90 04 00 30 01 00 00 .....0...
000001C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001D0 00 00 00 00 00 00 00 00 0E 74 65 78 74 00 00 00 .....text...
000001E0 A5 7F 04 00 00 10 00 00 00 80 04 00 00 04 00 00 Y.....€...
000001F0 00 00 00 00 00 00 00 00 00 00 00 00 60 00 00 60 .....
00000200 2E 69 64 61 74 61 00 00 D2 07 00 00 00 50 04 00 .idata..ô.....
00000210 00 08 00 00 00 84 04 00 00 00 00 00 00 00 00 00 .....@..@.reloc..
00000220 00 00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00 .....@..@.reloc..
00000230 0C 33 00 00 00 A0 04 00 00 34 00 00 00 8C 04 00 .3... ..4...@..

```

Fix the header information and check with PE-bear, this Dll has the original name is RFPmzNfQQFPXX and only exports one function named Main:

Offset	Name	Value	Meaning
4B360	Characteristics	0	
4B364	TimeDateStamp	612C95CD	Monday, 30.08.2021 08:24:45 UTC
4B368	MajorVersion	0	
4B36A	MinorVersion	0	
4B36C	Name	4BF92	BFPrzNfQQFPXX
4B370	Base	1	
4B374	NumberOfFunctions	1	
4B378	NumberofNames	1	
4B37C	AddressOfFunctions	4BF88	
4B380	AddressOfNames	4BF8C	
4B384	AddressOfNameOrdinals	4BF90	

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
4B388	1	8190	4BFAB	Main	

Back to the shellcode, after decompressing the Dll into memory, it will perform the task of a loader to map this Dll into a new memory region. Then, call to the exported function (here is the **Main** function) to perform the the main task of malware:

```

plugx_decrypted_dll = plugx_mapped_dll;
// 0070000 00 00 00 00 29 00 6C 02 A8 0A 03 00 92 15 6C 02 ....).l.~...'.l.
// 0070010 52 E5 02 00 69 00 6C 02 0C 15 00 00 00 00 00 00 RA..i.l.....
plugx_mapped_dll->signature = 0;
plugx_decrypted_dll->ptr_shellcode_base = ptr_call_addr; // 00402029 E8 00 00 00 00
plugx_decrypted_dll->shellcode_size = end_sc_offset;
plugx_decrypted_dll->ptr_encrypted_PlugX = ptr_enc_compressed_dll_addr; // 00403592 1C 9B ....
plugx_decrypted_dll->encrypted_PlugX_size = compressed_dll_size; // 0x2E552
plugx_decrypted_dll->config = config; // 0x0402069 (offset 0x69 on disk)
plugx_decrypted_dll->config_size = config_size; // 0x0150C
plugx_decrypted_dll->ptr_PlugX_entry_point = plugx_mapped_dll + payload_nt_headers->OptionalHeader.AddressOfEntryPoint;
VirtualProtect(lpAddress, payload_raw_size, PAGE_EXECUTE_READWRITE, &fOldProtect);
if ( !(plugx_decrypted_dll->ptr_PlugX_entry_point)(plugx_mapped_dll, 1, 0) )
return 0x15;
if ( ExportProc )
ExportProc(); // execute export function
if ( !VirtualFree(compressed_buf, 0, MEM_RELEASE) )
return 0x16;
if ( VirtualFree(uncompressed_buf, 0, MEM_RELEASE) )
return 0;
return 0x17;

```

**Note:** At the time of analyzing this shellcode, we have not yet confirmed it is a variant of the PlugX malware, but only raised doubts about the relationship. It was only when we analyzed the above extracted Dll, then we confirmed for sure that this was a variant of PlugX and renamed the fields in the struct for understandable reasons as screenshot above.

## 4. Analyze the extracted Dll

We will not go into detailed analysis of this Dll, but only provide the necessary information to prove that this is a PlugX variant as well as the process of decrypting the configuration information that the malware will be used.

### 4.1. How PlugX calls an API function

In this variant, information about API functions is stored in **xmmword**, then loaded into the **xmm0** (128-bit) register, the missing part of the function name will be loaded through the stack. The malicious code gets the handle of the Dll corresponding to these API functions,

then uses **GetProcAddress** function to retrieve the address of the specified API function to use later:

```
.text:10027A90 000      push    ebp
.text:10027A91 004      mov     ebp, esp
.text:10027A93 004      sub    esp, 84h
.text:10027A99 088      movdqa xmm0, xmmword_100078A0
.text:10027AA1 088      mov    eax, GetCurrentProcess_0
.text:10027AA6 088      push   ebx
.text:10027AA7 08C      push   esi
.text:10027AA8 090      xor    esi, esi
.text:10027AAA 090      mov    [ebp+lpName], ecx
.text:10027AAD 090      mov    [ebp+token_handle], esi
.text:10027AB0 090      mov    [ebp+var_60], 73h ; 's'
.text:10027AB6 090      push  edi
.text:10027AB7 094      mov    edi, ds:GetProcAddress
.text:10027ABD 094      movdqu xmmword ptr [ebp+ProcName], xmm0
.text:10027AC2 094      test   eax, eax
.text:10027AC4 094      jnz    short loc_10027AD7
.text:10027AC6 094      lea   eax, [ebp+ProcName]
.text:10027AC9 094      push  eax ; lpProcName
.text:10027ACA 098      call  f_retrieve_kernel32_handle
.text:10027ACA 098      push  eax ; hModule
.text:10027AD0 09C      call  edi ; GetProcAddress
.text:10027AD0 094      mov   GetCurrentProcess_0, eax
.text:10027AD2 094
.text:10027AD7 094      loc_10027AD7: ; CODE XREF: f_check_and_enable_privilege
                call  eax ; GetCurrentProcess_0
```

## 4.2. Create main thread to execute

The malware adjusts the **SeDebugPrivilege** and **SeTcbPrivilege** tokens of its own process in order to gain full access to system processes. Then it creates its main thread, which is named “**bootProc**”:

```

f_create_unnamed_event(0)→dll_base = dll_base;
f_create_unnamed_event(0)→dll_base = dll_base;
f_create_unnamed_event(0)→dll_base = dll_base;
*wszSeDebugPrivilege = 'e\0S';
*&wszSeDebugPrivilege[2] = 'e\0D';
*&wszSeDebugPrivilege[4] = 'u\0b';
*&wszSeDebugPrivilege[6] = 'P\0g';
*&wszSeDebugPrivilege[8] = 'i\0r';
*&wszSeDebugPrivilege[0xA] = 'i\0v';
*&wszSeDebugPrivilege[0xC] = 'e\0L';
*&wszSeDebugPrivilege[0xE] = 'e\0g';
wszSeDebugPrivilege[0x10] = 0;
*wszSeTcbPrivilege = 'e\0S';
*&wszSeTcbPrivilege[2] = 'c\0T';
*&wszSeTcbPrivilege[4] = 'P\0b';
*&wszSeTcbPrivilege[6] = 'i\0r';
*&wszSeTcbPrivilege[8] = 'i\0v';
*&wszSeTcbPrivilege[0xA] = 'e\0L';
*&wszSeTcbPrivilege[0xC] = 'e\0g';
v6 = 0;
f_check_and_enable_privilege(wszSeDebugPrivilege); // SeDebugPrivilege
f_check_and_enable_privilege(wszSeTcbPrivilege); // SeTcbPrivilege
strcpy(szbootProc, "bootProc");
critical_section = sub_10007E50(0);
return f_spawn_thread(critical_section, &p_thread_handle, szbootProc, f_main_thread_func 0);

```

### 4.3. Communicating with C2

The malware can communicate with C2 via TCP, HTTP or UDP protocols:

```

strcpy(szTCP_proto, "TCP");
strcpy(szHTTP_proto, "HTTP");
sz_protocol_info = L"";
strcpy(szUDP_proto, "UDP");
strcpy(szICMP_proto, "ICMP");
switch ( choose_proto_flag )
{
    case 2:
        sz_protocol_info = szTCP_proto;
        break;
    case 3:
        sz_protocol_info = szHTTP_proto;
        break;
    case 4:
        sz_protocol_info = szUDP_proto;
        break;
    case 5:
        sz_protocol_info = szICMP_proto;
        break;
    default:
        break;
}

```

```

// Protocol:[%s],
*szProto_Host_Proxy_format_str = _mm_load_si128(&xmmword_10007120);
strcpy(v15, "%s:%s\r\n");
port_num_hi = HIWORD(src→f_retrieve_ip_address);
port_num_lo = LOWORD(src→f_retrieve_ip_address);
v8 = a2[1];
// Host: [%s:%d], p
v13 = _mm_load_si128(&xmmword_10007240);
// roxy: [%d:%s:%d:
v14 = _mm_load_si128(&xmmword_10007180);
// Protocol:[%s], Host: [%s:%d], Proxy: [%d:%s:%d:%s:%s]\r\n
wprintfA(
    szProto_Host_Proxy_full_str,
    szProto_Host_Proxy_format_str,
    sz_protocol_info,
    a2 + 2,
    v8,
    port_num_lo,
    &src→field_4,
    port_num_hi,
    &src→event_handle_1,
    &src→field_84);
f_send_str_to_debugger(szProto_Host_Proxy_full_str);
switch ( choose_proto_flag )
{
    case 2:
        result = f_connect_c2_over_TCP(this, arg0, a2, src);
        break;
    case 3:
        result = f_connect_c2_over_HTTP(this, arg0, a2, src);
        break;
    case 4:
        result = f_connect_c2_over_UDP(this, arg0, a2, src);
        break;
    case 5:
        result = 0x32;
}

```

## 4.4. Implemented commands

The malware will receive commands from the attacker to execute the corresponding functions related to *Disk*, *Network*, *Process*, *Registry*, etc.

```

map_file_buf = f_mapping_file_and_retrun_buf();
if ( map_file_buf )
{
    strcpy(4sz_input_cmd[0], "Disk");
    (&map_file_buf)(0xFFFFFFFF, 0, 0x20120325, f_perform_disk_action_command);
}
f_perform_keylogger();
v15 = sub_100175f0();
if ( v15 )
{
    strcpy(4sz_input_cmd[0], "Netshod");
    (&v15)(0xFFFFFFFF, 5, 0x20120325, f_enumerate_network_resources, 4sz_input_cmd[0]);
}
v16 = sub_100174d0();
if ( v16 )
{
    strcpy(4sz_input_cmd[0], "Netstat");
    (&v16)(0xFFFFFFFF, 4, 0x20120325, f_retrieve_network_statistics, 4sz_input_cmd[0]);
}
v17 = sub_10018000();
if ( v17 )
{
    strcpy(4sz_input_cmd[0], "Option");
    (&v17)(0xFFFFFFFF, 6, 0x20120325, f_perform_option_sub_command, 4sz_input_cmd[0]);
}
v18 = sub_10019500();
if ( v18 )
{
    strcpy(4sz_input_cmd[0], "Perthop");
    (&v18)(0xFFFFFFFF, 7, 0x20120325, f_start_port_mapping, 4sz_input_cmd[0]);
}
v19 = sub_10019A10();
if ( v19 )
{
    strcpy(4sz_input_cmd[0], "Process");
    (&v19)(0xFFFFFFFF, 1, 0x20120324, f_perform_process_sub_command, 4sz_input_cmd[0]);
}

switch ( cmd_info->subcommand )
{
case 0x3000:
    result = f_enumerate_drives(a1, cmd_info);
    break;
case 0x3001:
    result = f_find_file(a1, cmd_info);
    break;
case 0x3002:
    result = f_find_file_recursively(a1, cmd_info);
    break;
case 0x3004:
    result = f_read_file(a1, cmd_info);
    break;
case 0x3007:
    result = f_write_file(a1, cmd_info);
    break;
case 0x300A:
    result = f_create_directory(a1, cmd_info);
    break;
case 0x300C:
    result = f_create_process_on_hidden_desktop(a1, cmd_info);
    break;
case 0x3000:
    result = f_file_action(a1, cmd_info); // file copy/rename/delete/move
    break;
case 0x300E:
    result = f_get_expanded_environment_string(a1, cmd_info);
    break;
default:
    result = 0xFFFFFFFF;
    break;
}
return result;
    
```

The entire list of commands as shown in the table below that the attacker can execute through this malware sample:

Command Group	Sub-command	Description
Disk	0x3000	Get information about the drives (type, free space)
0x3001	Find file	
0x3002	Find file recursively	
0x3004	Read data from the specified file	
0x3007	Write data to the specified file	
0x300A	Create a new directory	
0x300C	Create a new process on hidden desktop	



0x300D	File action (file copy/rename/delete/move)	
0x300E	Expand environment-variable strings	
Nethood	0xA000	Enumeration of network resources
Netstat	0xD000	Retrieve a table that contains a list of TCP endpoints
0xD001	Retrieve a table that contains a list of UDP endpoints	
0xD002	Set the state of a TCP connection	
Option	0x2000	Lock the workstation's display
0x2001	Force shut down the system	
0x2002	Restart the system	
0x2003	Shut down the system safety	
0x2005	Display message box	
PortMap	0xB000	Perform port mapping
Process	0x5000	Retrieve processes info
0x5001	Retrieve modules info	
0x5002	Terminate specified process	
RegEdit	0x9000	Enumerate registry

0x9001	Create registry	
0x9002	Delete registry	
0x9003	Copy registry	
0x9004	Enumerates the values of the specified open registry key	
0x9005	Sets the data and type of a specified value under a registry key	
0x9006	Deletes a named value from the specified registry key	
0x9007	Retrieves a registry value	
Service	0x6000	Retrieves the configuration parameters of the specified service
0x6001	Changes the configuration parameters of a service	
0x6002	Starts a service	
0x6003	Sends a control code to a service	
0x6004	Delete service	
Shell	0x7002	Create pipe and execute command line
SQL	0xC000	Get SQL data sources
0xC001	Lists SQL drivers	
0xC002	Executes SQL statement	

Telnet	0x7100	Start telnet server
Screen	0x4000	simulate working over the RDP Protocol
0x4100	Take screenshot	
KeyLog	0xE000	Perform key logger function, log keystrokes to file "%allusersprofile%\MSDN\6.0\USER.DAT"

#### 4.5. Decrypt PlugX configuration

As analyzed above, the malware will connect to the C2 address via HTTP, TCP or UDP protocols depending on the specified configuration. So where is this config stored? With the old malware samples that we have analyzed (1, 2, 3, 4), the PlugX configuration is usually stored in the **.data** section with the size of **0x724 (1828)** bytes.

```

.data:1001E000  _data          segment para pub
.data:1001E000          assume cs:_data
.data:1001E000          ;org 1001E000h
.data:1001E000  encoded_config_data  db 0D9h ; 0
.data:1001E001          db 31h ; 1
.data:1001E002          db 33h ; 3
.data:1001E003          db 34h ; 4
.data:1001E004          db 78h ; x
.data:1001E005          db 36h ; 6
.data:1001E006          db 5Eh ; ^
.data:1001E007          db 38h ; 8
.data:1001E008          db 5Ah ; Z
.data:1001E009          db 31h ; 1
.data:1001E00A          db 40h ; 0
.data:1001E00B          db 33h ; 3
.data:1001E00C          db 5Bh ; [
.data:1001E00D          db 35h ; 5
.data:1001E00E          db 45h ; E
.data:1001E00F          db 37h ; 7
.data:1001E010          db 57h ; W
.data:1001E011          db 39h ; 9
.data:1001E012          db 57h ; W
.data:1001E013          db 32h ; 2
.data:1001E014          db 47h ; G
.data:1001E015          db 34h ; 4
.data:1001E016          db 15h ;
.data:1001E017          db 36h ; 6
.data:1001E018          db 7Ah ; z
.data:1001E019          db 38h ; 8
.data:1001E01A          db 58h ; X
.data:1001E01B          db 31h ; 1
.data:1001E01C          db 5Eh ; ^
.data:1001E01D          db 33h ; 3

f_MemCpy(&MalConfig, &encoded_config_data, 0x724u);
result = f_memcmp(&MalConfig, "XXXXXXXX", 8u);
if ( result )
{
    // 123456789
    strcpy(xor_key, "123456789");
    xor_key_len = f_lstrlenA(xor_key);
    result = f_XorDecode(&MalConfig, 0x724, xor_key, xor_key_len);
}


```

Going back to the sample we are analyzing, we see that before the step of checking the parameters passed when the malware executes, it will call the function that performs the task of decrypting the configuration:

```

ptr_cmd_line = GetCommandLineW();
CommandLineToArgvW = ::CommandLineToArgvW;
strcpy(v46, "vW");
*v45 = _mm_load_si128(&xmmword_10007610);
if ( !::CommandLineToArgvW )
{
    shell32_handle = g_shell32_handle;
    strcpy(sz_shell32, "shell32");
    if ( !g_shell32_handle )
    {
        shell32_handle = LoadLibraryA(sz_shell32);
        g_shell32_handle = shell32_handle;
    }
    CommandLineToArgvW = GetProcAddress(shell32_handle, v45);
    ::CommandLineToArgvW = CommandLineToArgvW;
}
sz_arg_list = CommandLineToArgvW(ptr_cmd_line, &num_arguments);
sub_10007DC0(0);
f_decrypt_embedded_config_or_from_file_and_copy_to_mem();
if ( num_arguments == 1 )
    f_launch_process_or_create_service();
if ( num_arguments == 3 )
{
    lstrlenW = ::lstrlenW;
    arg_passed_1 = sz_arg_list[1];
    passed_arg1_info.buffer = 0;
    passed_arg1_info.buffer1 = 0;
}

```



Diving into this function, combined with additional debugging from shellcode, renaming the fields in the generated struct, we get the following information:

- PlugX's configuration is embedded in shellcode and starts at offset **0x69**.
- The size of the configuration is **0x0150C (5388)** bytes.
- Decryption key is **0xB4**.

```

plugx_mapped_dll->signature = 0;
plugx_decrypted_dll->ptr_shellcode_base = ptr_call_addr; // 00002029 EB 00 00 00 00
plugx_decrypted_dll->shellcode_size = end_sc_offset;
plugx_decrypted_dll->ptr_encrypted_PlugX = ptr_enc_compressed_dll_addr; // 00403092 1C 9B ....
plugx_decrypted_dll->encrypted_PlugX_size = compressed_dll_size; // 0x20532
plugx_decrypted_dll->PlugX_config = config; // 0x0002000 (offset 0x69 on disk)
plugx_decrypted_dll->PlugX_config_size = config_size; // 0x0150C
plugx_decrypted_dll->ptr_PlugX_entry_point = plugx_mapped_dll + payload_nt_headers->OptionalHeader.AddressOfEntryPoint;
VirtualProtect(lpaddress, payload_raw_size, PAGE_EXECUTE_READWRITE, &Fl0idProtect);
if ( ! (plugx_decrypted_dll->ptr_PlugX_entry_point)(plugx_mapped_dll, 1, 0) )
return 0x15;
if ( ExportProc )
ExportProc(); // execute export function

```

```

PlugX_mapped_dll_base = f_create_unnamed_event(&dll_base);
ptr_PlugX_config = PlugX_mapped_dll_base->PlugX_config;
signature = ptr_PlugX_config->signature; // 0xC6A8BE5E
if ( ptr_PlugX_config->signature == ptr_PlugX_config->compared_value )
goto setup_config_buffer;
if ( PlugX_mapped_dll_base->PlugX_config_size != 0x150C )
goto setup_config_buffer;
sub_10007D90(0);
xor_key = signature + 0x36; // 0xC6A8BE5E + 0xb88505d6 = 0x98EF14b4
// -> xor_key = 0xB4
i = 0;
while ( i < 0x150C )
{
ptr_decrypted_config = &decrypted_config[i++];
ptr_decrypted_config = xor_key ^ ptr_decrypted_config[ptr_PlugX_config - decrypted_config];
}
if ( ptr_PlugX_config->signature != signature_computed )
goto setup_config_buffer;
fmemset_ret(g_std_decrypt_data, 0, 0x150C);
result = 0;
else
{
f_decrypt_embedded_config_or_from_file_and_copy_to_mem
}

```

log\_file.cba

Address	Hex	Decoded text
00000000	77 04 81 EE 00 00 00 00 00 00 00 00 00 00 00 00	w.....EEMF1
00000010	80 73 07 85 7C 03 C8 C2 70 00 00 00 00 00 00 00	..D1;Ag1...DFE
00000020	44 19 03 00 00 00 00 00 00 00 00 00 00 00 00 00	-33.....M
00000030	80 5F 83 E8 05 08 C3 C8 48 0C 15 00 00 00 00 00	Di...XXXXXXXXXX
00000040	80 00 4A 18 83 C6 94 37 7C 04 81 FF 81 80 00 00	..fA..Ah.....
00000050	5F 25 7E 79 20 00 00 00 00 00 00 00 00 00 00 00	..66.....N&E..0
00000060	24 04 34 24 04 42 45 4D 4D 43 4D 4D 43 4D 43 4D	....XXXX.....
00000070	4B 40 48 4D 48 4D 48 4D 48 4D 48 4D 48 4D 48 4D	XXXXXXXXXXXX.....
00000080	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	N''''XXXXXXXXXXXX
00000090	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000000A0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000000B0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000000C0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000000D0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000000E0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000000F0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000100	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000110	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000120	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000130	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000140	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000150	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000160	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000170	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000180	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
00000190	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000001A0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000001B0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000001C0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000001D0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000001E0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX
000001F0	88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88	XXXXXXXXXXXXXXXXXX

With all the complete information as above, it is possible to recover the configuration information easily:

IP	Port
86.78.23.152	53
86.78.23.152	22
86.78.23.152	8080
86.78.23.152	23

PlugX\_decrypted\_config.bin

Address	Hex	Decoded text
00000330	01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01	port-num 01 sip addr_01
00000340	01 01 01 01 01 01 00 00 00 00 00 00 00 00 00 00	.....FF FF FF FF 05 00 00 00 00 00 00 00 00 00
00000350	FF FF FF FF FF 05 00 00 00 00 00 00 00 00 00 00 00	.....1.152 86.78.23.152
00000360	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....22 86.78.23.152
00000370	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....8080 86.78.23.152
00000380	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....23 86.78.23.152
00000390	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000003A0	01 38 28 22 11 22 11 22 11 22 11 22 11 22 11 22	.....
000003B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000003C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000003D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000003E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000003F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000400	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000410	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000420	00 05 00 10000 0000 0000 0000 0000 0000 0000 0000	.....
00000430	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000440	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000450	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000460	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000470	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000480	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000490	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000004A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000004B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000004C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000004D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000004E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000004F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000500	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000510	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000520	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000530	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000540	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000550	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000560	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000570	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

In addition to the list of C2 addresses above, there is additional information related to the directory created on the victim machine to contain malware files as well as the name of the service that can be created:

```
// "bdreinit.exe" -> (size: 13)
// crash handling component BDRReinit.exe
wsz_bdreinit_exe[0] = 'd\0b';
wsz_bdreinit_exe[1] = 'e\0r';
wsz_bdreinit_exe[2] = 'n\0i';
wsz_bdreinit_exe[3] = 't\01';
wsz_bdreinit_exe[4] = 'e\0.';
wsz_bdreinit_exe[5] = 'e\0x';
LOWORD(wsz_bdreinit_exe[6]) = 0;
```

00000970	00 00 00 00 00 25 00 50	00 72 00 6F 00 67 00 72	a	%	P	r	o	g	r
00000980	00 61 00 6D 00 46 00 69	00 6C 00 65 00 73 00 25	a	m	f	i	l	e	s%
00000990	00 5C 00 42 00 69 00 74	00 44 00 65 00 66 00 65	\	B	i	t	D	e	f
000009A0	00 6E 00 64 00 65 00 72	00 20 00 55 00 70 00 64	n	d	e	r	U	p	d
000009B0	00 61 00 74 00 65 00 00	00 00 00 00 00 00 00 00	a	t	e				
000009C0	00 00 00 00 00 42 00 69	00 74 00 44 00 65 00 66							
00000B80	00 65 00 6E 00 64 00 65	00 72 00 20 00 43 00 72	e	n	d	e	r	C	r
00000B90	00 61 00 73 00 68 00 20	00 48 00 61 00 6E 00 64	a	s	h	H	a	n	d
00000BA0	00 6C 00 65 00 72 00 00	00 00 00 00 00 00 00 00	l	e	r				

To make our life easier, I wrote a python script to automatically extract configuration information for this variant. The output after running the script is as follows:

```
$ python plugx_extract_config.py plugx_decrypted_config.bin
[+] Config file: plugx_decrypted_config.bin
[+] Config size: 5388 bytes
[+] Folder name: %ProgramFiles%\BitDefender Update
[+] Service name: BitDefender Crash Handler
[+] Proto info: HTTP://
[+] C2 servers:
    86.78.23.152:53
    86.78.23.152:22
    86.78.23.152:8080
    86.78.23.152:23
[+] Campaign ID: 1234
```

## 5. Conclusion

CrowdStrike researchers first published information on Mustang Panda in June 2018, after approximately one year of observing malicious activities that shared unique Tactics, Techniques, and Procedures (TTPs). However, according to research and collect from many different cybersecurity companies, this group of APTs has existed for more than a decade with different variants found around the world. Mustang Panda, believed to be a APT group based in China, is evaluated as one of the highly detrimental APT groups, applying sophisticated techniques to infect malware, aiming to gain as much long-term access as possible to conduct espionage and information theft.

In this blog we have analyzed the different steps the infamous PlugX RAT follows to start execution and avoid detection. Thereby, it can be seen that this APT group is still active and constantly looking for ways to improve their techniques. VinCSS will continue to search for additional samples and variants that may be associated with this PlugX variant that we analyzed in this article.

## 6. References

## 7. Indicators of Compromise

log.dll - db0c90da56ad338fa48c720d001f8ed240d545b032b2c2135b87eb9a56b07721

---

log.dll - 84893f36dac3bba6bf09ea04da5d7b9608b892f76a7c25143deebe50ecbbdc5d

---

log.dll - 3171285c4a846368937968bf53bc48ae5c980fe32b0de10cf0226b9122576f4e

---

log.dll - da28eb4f4a66c2561ce1b9e827cb7c0e4b10afe0ee3efd82e3cc2110178c9b7a

---

log.dat - 2de77804e2bd9b843a826f194389c2605cfc17fd2fafde1b8eb2f819fc6c0c84

Decrypted config:

[+] Folder name: %ProgramFiles%\BitDefender Update

[+] Service name: BitDefender Crash Handler

[+] Proto info: HTTP://

[+] C2 servers:

86.78.23.152:53

86.78.23.152:22

86.78.23.152:8080

86.78.23.152:23

[+] Campaign ID: 1234

---

---

log.dat - 0e9e270244371a51fbb0991ee246ef34775787132822d85da0c99f10b17539c0

Decrypted config:

[+] Folder name: %ProgramFiles%\BitDefender Update

[+] Service name: BitDefender Crash Handler

[+] Proto info: HTTP://

[+] C2 servers:

86.79.75.55:80

86.79.75.55:53

86.79.75.46:80

86.79.75.46:53

[+] Campaign ID: 1234

---

log.dat - 3268dc1cd5c629209df16b120e22f601a7642a85628b82c4715fe2b9fbc19eb0

Decrypted config:

[+] Folder name: %ProgramFiles%\Common Files\ARO 2012

[+] Service name: BitDefender Crash Handler

[+] Proto info: HTTP://

[+] C2 servers:

86.78.23.152:23

86.78.23.152:22

86.78.23.152:8080

86.78.23.152:53

[+] Campaign ID: 1234

---



---

log.dat - 02a9b3beaa34a75a4e2788e0f7038aaf2b9c633a6bdbfe771882b4b7330fa0c5  
(THOR PlugX)

Decrypted config:

[+] Folder name: %ProgramFiles%\BitDefender Handler

[+] Service name: BitDefender Update Handler

[+] Proto info: HTTP://

[+] C2 servers:

www.locvnpt.com:443

www.locvnpt.com:8080

www.locvnpt.com:80

www.locvnpt.com:53

[+] Campaign ID: 1234

*Click [here](#) for Vietnamese version.*

**Dang Dinh Phuong – Threat Hunter**

**Tran Trung Kien (aka m4n0w4r) - Malware Analysis Expert**

**R&D Center - VinCSS (a member of Vingroup)**