

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

1. Executive Summary

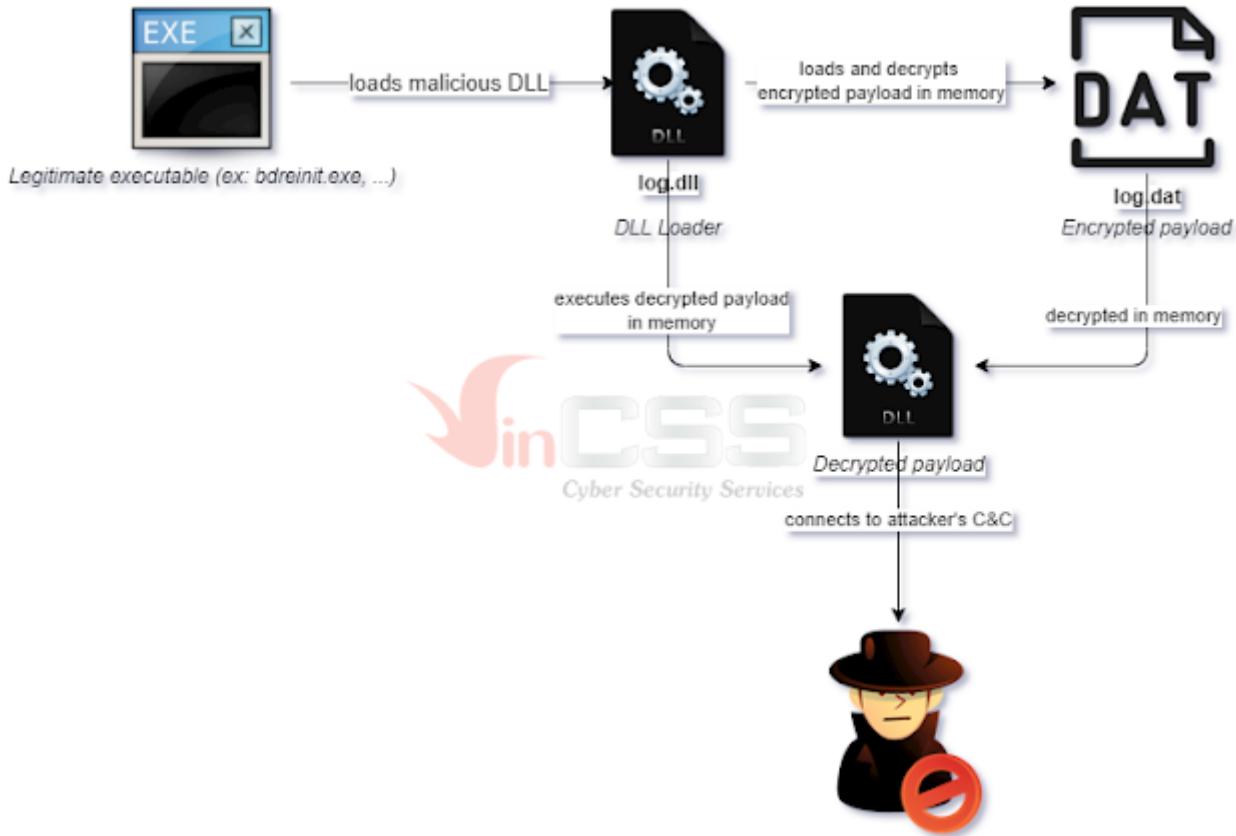
At VinCSS, through continuous cyber security monitoring, hunting malware samples and evaluate them to determine the potential risks, especially malware samples targeting Vietnam. Recently, during hunting on [VirusTotal's](#) platform, performing scan for specific byte patterns related to the **Mustang Panda (PlugX)**, we discovered a series of malware samples that we suspect have a relationship with this group were uploaded from Vietnam.

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

FILES 5 / 5		90 days				
		Detections	Size	First seen	Last seen	Submitters
<input type="checkbox"/>	log.dll pedl	11 / 68	664.00 KB	2022-05-07 01:33:18	2022-05-07 01:33:18	1
<input type="checkbox"/>	84893f360ac38846f88e#840450789688852f7647c25143deee9#ec880c50 pedl	9 / 68	103.00 KB	2022-05-05 12:42:34	2022-05-05 17:58:50	2
<input type="checkbox"/>	3171285c48463689579e8bf530c48ae5c988fe32800e10cf802e88122576f4e pedl	13 / 67	377.50 KB	2022-04-25 14:04:36	2022-04-25 14:04:36	1
<input type="checkbox"/>	6448282c91d5297c8474121f788c843c884e884e340c68851889417c133a9 pedl	13 / 68	52.00 KB	2022-04-12 02:56:42	2022-04-12 02:56:42	1
<input type="checkbox"/>	d428e4f46ac2561ce188e827c8708e4018bf8ee3ef082e5cc2118178c987a pedl	10 / 55	576.00 KB	2022-03-26 13:16:05	2022-03-26 13:16:05	1

Based on the above information, we believe that there is a possibility that malware has been installed in a few orgs in Vietnam, so we decided to analyze these samples. During the analysis, based on the detected indicators, we continue to hunt for the missing data to add a more complete picture for the analysis.

A general overview of the execution flow looks like this:



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)
Implib1100	Visual Studio 2012 - 11.0
Import	Visual Studio
Utc1800_CPP	Visual Studio 2013 - 12.0
Masim1200	Visual Studio 2013 - 12.0
Utc1800_C	Visual Studio 2013 - 12.0
Import (old)	Visual Studio
Export1200	Visual Studio 2013 - 12.0 RTM
Linker1200	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 ...	Strong Packed - 2.2743 % ZERO
02	im	00059000	000046F4	00057C00	00004800	40000040	.rdata	20 02 05 00 34 ...	Very not packed - 43.6306 % ZERO
03		0005E000	00002FA0	0005C400	00001200	C0000040	.data	4E E6 40 BB B1 ...	Very not packed - 64.3012 % ZERO
04		00061000	00000ED4	0005D600	00001000	42000040	.refloc	00 10 00 00 0C ...	Not packed - 16.6992 % ZERO

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

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	IjAt.dll		
5BCA0	Base	1			
5BCA4	NumberOfFunctions	2			
5BCA8	NumberOfNames	2			
5BCAC	AddressOfFunctions	5D0B8			
5BCB0	AddressOfNames	5D0C0			
5BCB4	AddressOfNameOrdinals	5D0C8			
Exported Functions [2 entries]					
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
5BCB8	1	1000	5D0D5	LogFree	
5BCBC	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-LVVM**, using the **Control Flow Flattening** technique:

```

591 LOGITE(v142) = (v107 & v109 | (v109 ^ 1) & (v109 ^ 1) & (v109 ^ 1) & (v107 ^ 1 ^ v
592 LOGITE(v107) = v108 & v142 ^ BYTEL(v109) ^ 1 ^ 1 & (v102 | BYTEL(v109) ^ 1 ^ 1 |
593 BYTEL(v107) = (BYTEL(v107) & BYTEL(v108) | BYTEL(v108) ^ BYTEL(v107)) ^ 1) & v108
594 BYTEL(v107) = (BYTEL(v107) ^ 1) & BYTEL(v107) & (BYTEL(v107) ^ 1) | BYTEL(v107) ^ 1
595 LOGITE(v107) = BYTEL(v107) & (BYTEL(v107) ^ 1) ^ 1;
596 result = (v108 ^ BYTEL(v107) | (BYTEL(v107) | v140) ^ 1) & ((v107 & (v107 ^ 1) & (v
597 v109 = BYTEL(v107) ^ v107 | (v107 | BYTEL(v107)) ^ 1);
598 tmp = 0x8129720;
599 if ( result = v108 ) & 1 )
600 {
601     tmp1 = 0x47F2940;
602 }
603 v106 = (v107 & 1) = 0;
604 control_var = 0x0C10C1EA;
605 if ( v106 )
606 {
607     tmp2 = tmp1;
608 }
609 if ( !result & 1 )
610 {
611     tmp2 = tmp1;
612 }
613 do
614 {
615     LABEL_7:
616     if ( control_var <= 0x5738270E )
617     {
618         goto LABEL_11;
619     }
620     LABEL_8:
621     while ( control_var == 0x5738270E )
622     {
623         control_var = tmp2;
624         if ( tmp2 <= 0x5738270E )
625         {

```

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

- **LogInit** function:

This function will call the **LogInit_0** function:

```

.text:1004E5E0 ; Exported entry 2. LogInit
.text:1004E5E0
.text:1004E5E0
.text:1004E5E0 ; Attributes: thunk
.text:1004E5E0
.text:1004E5E0 ; void __stdcall LogInit()
.text:1004E5E0             public LogInit
.text:1004E5E0 LogInit      proc near
.text:1004E5E0                 ; DATA XREF: .rdata:off_1005D08B+0
.text:1004E5E0                 jmp    LogInit_0 ; TAGS: ['Enum', 'FileWIN']
.text:1004E5E0 LogInit      endp
.text:1004E5E0

```

```

1 // attributes: thunk
2 void __stdcall LogInit()
3 {
4     LogInit_0();
5 }

```

Similar to the above, the code at the **LogInit_0** function has also been completely obfuscated, it takes a long time for IDA to decompile the code of this function:

```

4967 v200 = v199 * v199 * 0x72D90420;
4968 v201 = ~v199 & (v199 ^ 0x48826200F) | (v199 * 0x77090620) & v199;
4969 v202 = (~((v199 * 0x77090620) & v200) & 0xF52380E | (v199 * 0x77090620) & v200 & v201) | v202 & 0x89026107 | ~v202 & 0x89026107 | v201;
4970 v203 = v202 & (v202 ^ 0x89026107) & ~v202 & 0x89026107 | ~v202 & 0x89026107 | v200;
4971 v204 = v203;
4972 v205 = ~v203 & 0xA13D981E7;
4973 v206 = (~v203 & 0xA13D981E7 | v203 & 0xF50E2833) ^ 0xF50E2833;
4974 read_content_status = (v206 & ((v205 | v204 & 0x5EC2FE10) ^ 0x5EC2FE10) | (v205 | v204) & 0x5EC2FE10) ^ 0x5EC2FE10;
4975 control_var = 0x409000680;
4976 v207 = read_content_status;
4977 v208 = dword_10000FET4 < 0xA;
4978 do
4979 {
4980     LABEL_19:
4981     while ( control_var <= (int)0xC30CF813 )
4982     {
4983         if ( control_var > (int)0xA34D33B6 )
4984         {
4985             if ( control_var == 0xA3H63387 )
4986             {
4987                 control_var = 0x7089932;
4988                 goto LABEL_3;
4989             }
4990             (~decrypted_shellcode)();
4991             control_var = 0x8908665F; // exec decrypted shell
4992         }
4993         else
4994         {
4995             if ( control_var == 0x8908665F )
4996             {
4997                 (~decrypted_shellcode()); // exec decrypted shell
4998             v2485 = dword_10000FET8 & (dword_10000FET8 - 1);
4999             v2489 = ~v2489;
5000             v2481 = v2489 & ((dword_10000FET8 & (dword_10000FET8 - 1)) ^ 0x20430972);
5001             v2492 = ~v2491 & ~v2489 & 0x20430972 | ~v2489 & 0x20430972 * v2491 & 0x

```

The primary task of the **LogInit_0** function is to call the function

f_read_content_of_log_dat_file_to_buf for reading the content of **log.dat** file and execute the decrypted shellcode:

```

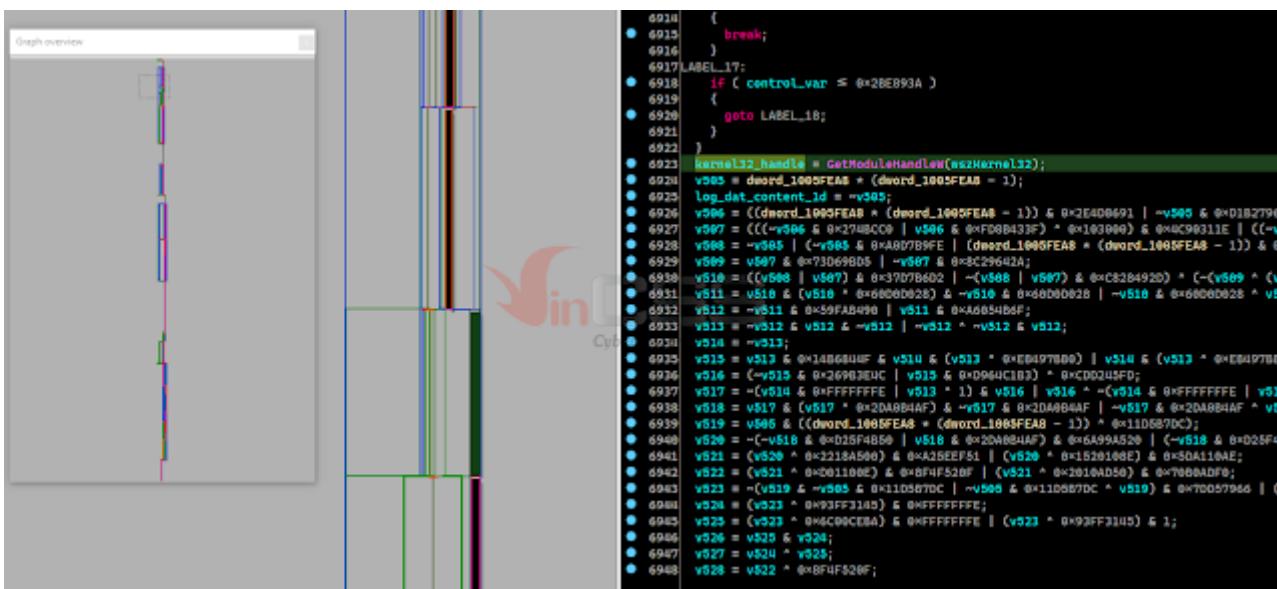
public LogInit
proc near
    ; DATA XREF: .rdata:off_1005D0B8+0
jmp  LogInit_0 ; TAGS: ['Enum', 'FileWIN']
endp
23 calls, 1 strings

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 _strncpy ; call _strncpy to compare string
- 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 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:



```

6918 {
6919     break;
6920 }
6921 LABEL_17:
6922     if ( control_var <= 0x2BE893A )
6923         goto LABEL_18;
6924     }
6925 kernel32_handle = GetModuleHandleW(mszKernel32);
6926 v505 = dword_1005FEAB * (dword_1005FEAB - 1);
6927 v506 = ((v505 & 0x274BC09 | v506 & 0xFD88433F) ^ 0x103000) & 0x4C90311E | ((~v506
6928 v507 = (~v506 & 0x8A07B9FE | (dword_1005FEAB * (dword_1005FEAB - 1)) & 0x
6929 v508 = v507 & 0x73069BD5 | ~v507 & 0x8C29642A);
6930 v509 = ((v508 | v507) & 0x3707B602) | (~v508 | v507) & 0xC8284920) * (~v509 & (v50
6931 v510 = v510 * (v510 * 0x60000028) & ~v510 & 0x60000028 | ~v510 & 0x60000028 * v51
6932 v511 = ~v511 & 0x50FAB49 | v511 & 0x4A6854BF;
6933 v512 = ~v512 & v512 & ~v512 | ~v512 & ~v512 & v512;
6934 v513 = ~v513;
6935 v513 = v513 & 0x14B68UHF & v514 & (v513 * 0x8E8497B80) | v514 & (v513 * 0x8E8497B80
6936 v516 = (~v515 & 0x269B3ENC | v515 & 0xD964C1B3) * 0xCD0245FD;
6937 v517 = (~v514 & 0xFFFFFFF | v513 * 1) & v516 | v516 ^ ~v514 & 0xFFFFFFFF | v513
6938 v518 = v517 & (v517 * 0x20A8B4AF) & ~v517 & 0x2D40B4AF | ~v517 & 0x2D40B4AF ^ v51
6939 v519 = v519 & (((dword_1005FEAB * (dword_1005FEAB - 1)) ^ 0x11D587DC);
6940 v520 = (~v518 & 0x0D25F4858) | v518 & 0x2D40B4AF) & 0x6A994520 | (~v518 & 0x0D25F48
6941 v521 = (v520 ^ 0x2218A500) & 0x4A25EEF51 | (v520 * 0x1520106E) & 0x5D4119AE;
6942 v522 = (v521 ^ 0x0D1108E) & 0x84F4F520F | (v521 ^ 0x2B10AD50) & 0x70B0AD50;
6943 v523 = (~v519 & 0x110587DC) | ~v505 & 0x11D587DC ^ v519) & 0x7D057966 | (~v519
6944 v524 = (v523 ^ 0x93FF3145) & 0xFFFFFFFF;
6945 v525 = (v523 ^ 0x46C90CEBA) & 0xFFFFFFFF | (v523 * 0x93FF3145) & 1;
6946 v526 = v525 & v524;
6947 v527 = v524 * v525;
6948 v528 = v522 * 0x8BF520F;

```

The major task of this function as the following:

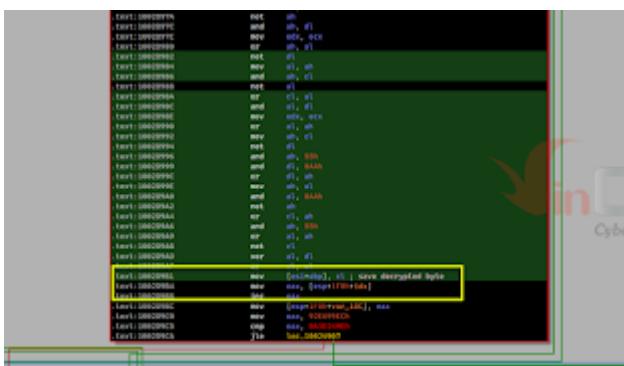
- Call the **GetModuleHandleW** function to retrieve the handle of **kernel32.dll**.
- Call the **GetProcAddress** function to get the addresses of the APIs: **VirtualAlloc**, **GetModuleFileNameA**, **CreateFileA**, **ReadFile**.
- Use the above APIs to retrieve the path to the **log.dat** file and 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 11662 control_var = 0x7A7LAZRA4;
mov  ecx, [ebp+documented_challenge] 11663 if ( read_content_status )
test eax, ea_15_calls, 0 strings
mov  edx, 11J
mov  [ecx], calls:
mov  eax, 7A7
cmovez eax, edx
cmp  eax, 8E1
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 eax ; call CreateFileA for open file but not retrieve file handle
call ds:CloseHandle ; call CloseHandle to release handle to log.dat file
call eax ; 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 eax ; 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.



```

13803    LOGYTE(v4939) = v4930 & BYTEL(v4939) | BYTEL(v4939) * v4939;
13804    BYTEL(v5065) = ~BYTE(v5064) & 0x1;
13805    BYTEL(v5065) = ((v5065 & 0x79) * (~BYTE(v5065 & 0x80) * 0x80)) & ((v5065 & 0x80) * 0x80);
13806    LOGYTE(v3530) = v5064;
13807    BYTEL(v5064) = v3530 & 0x815 | BYTEL(v5065)) ^ (~BYTEL(v5064) & 0x815) * BYTES;
13808    BYTEL(v5065) = (~BYTEL(v4939) & 0x81E) * 0x71) * (~(~BYTE)v5064);
13809    BYTEL(v5065) = BYTEL(v5064) ^ (~BYTEL(v5064) | BYTEL(v5064));
13810    LOGYTE(v4935) = ~BYTEL(v4935);
13811    log_dat_content[idx] = ((v5065 | v4939) & 0x55 | ~(v5065 | v4939) & 0x55) *
13812    v5060 = idx + 1;
13813    control_var_1 = 0x92E699EC;
13814    }
13815    else if ( control_var_1 == 0x92E699EC )
13816    {
13817        v4341 = ~(dword_1005FE80 + (dword_1005FE80 - 1));
13818        v5310 = dword_1005FE80 + (dword_1005FE80 - 1);
13819        v4342 = v4341 & 0x3A21E92 | (dword_1005FE80 * (dword_1005FE80 - 1)) & 0xC;
13820        v4343 = (~v4342 & 0x70F0FA02 | v4342 & 0x8FB0565D) * 0x158F7001) & 0x1D89F;
13821        v4344 = v4343 * 0x17FB2B9C) & (v4343 * 0xE27693E);
13822        v4345 = ~v4344;
13823        v4346 = (((v4341 & 0x7550720 | (dword_1005FE80 * (dword_1005FE80 - 1)) & 0x10000000) * 0x10000000) & 0x10000000) & (~v4346 | v4345) & 0x1FD7CE0E) & (~v4346 | v4345);
13824        v4347 = (v4346 | v4345) & 0x002031F1 | ~(v4346 | v4345) & 0x1FD7CE0E) & (~v4346 | v4345);
13825

```

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 4 / 4	90 days	00	X	Submitters
	Detectors	Size	Last seen	
306A0C1CD66A292060F7A9120E22F691A7641485A28682C4715FE289F9C19EB	0 / 57	194.66 KB	2022-05-07 01:32:51	1
8243836EA4A44754AE2788E0F7838A3F2890C334A6808FE7718828487330FAB5	2 / 59	189.23 KB	2022-05-05 12:44:31	1
BE6E278044371651FF88891EE246EF34775787132822D850A0C36718817530CB	0 / 57	194.66 KB	2022-04-25 14:07:46	1
2DE77804E2BD9B843A826f194389c2605fcf17fd2fafde1b8eb2f819fc6c0c84	0 / 57	194.66 KB	2022-04-20 12:33:19	1

With the above results, at the time of analysis, we selected the **log.dat** file

([2de77804e2bd9b843a826f194389c2605fcf17fd2fafde1b8eb2f819fc6c0c84](https://www.virustotal.com/gui/file/2de77804e2bd9b843a826f194389c2605fcf17fd2fafde1b8eb2f819fc6c0c84)) 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 F7 06 81 EE 00 00 00 80 C5 00 45 4D 66 83 EE H.i....EAA,EMEfi
00000010 00 73 07 55 7C 03 C0 C2 70 5D 8D 12 55 66 83 C9 s.UI.AAp)...UEfE
00000020 00 5D 7D 05 0D 00 00 00 E8 00 00 00 00 57 BF .]}.....è....Wt
00000030 44 49 00 00 5F F9 58 50 50 48 58 58 57 66 BF 9D DI...gXPPHXXWez.
00000040 00 5F 83 E8 05 0B C0 FC 68 0C 15 00 00 00 00 00 .._f...Auh...
00000050 00 00 6A D5 83 C4 04 57 7C 06 81 FF BF 60 00 00 ..jÖfÅ.W|...qz...
00000060 5F 0B F6 F9 E8 0C 15 00 00 5E BE 68 CA EA 0A DC <@é...^hEé.U
00000070 7E B4 B4 B4 B4 4B 4B 4B B4 B4 B4 B4 B4 B4 B4 =----KKKK-----
00000080 B4 B4 B4 B4 B4 4B -----KKKKKKKKKKK
00000090 4B KKKKKKKKKK-----
000000A0 BE B4 B4 B4 B5 75 PC PC BC BC BC BC BC BC BC N'----aaaaaaaarrrrrr
000000B0 B5 brrrrrrrrrrrrrrrrrr
000000C0 B5 brrrrrrrrrrrrrrrrrr
000000D0 B5 brrrrrrrrrrrrrrrrrr
000000E0 B5 brrrrrrrrrrrrrrrrrr
000000F0 B5 brrrrrrrrrrrrrrrrrr
00000100 B5 brrrrrrrrrrrrrrrrrr
00000110 B5 brrrrrrrrrrrrrrrrrr
00000120 B5 brrrrrrrrrrrrrrrrrr
00000130 B5 brrrrrrrrrrrrrrrrrr
00000140 B5 brrrrrrrrrrrrrrrrrr
00000150 B5 brrrrrrrrrrrrrrrrrr
00000160 B5 brrrrrrrrrrrrrrrrrr
00000170 B5 brrrrrrrrrrrrrrrrrr
00000180 B5 brrrrrrrrrrrrrrrrrr
```

decrypted shellcode

I use two tools, **FLOSS** and **scdbg** 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
```

scDbg - libemu Shellcode Logger Launch Interface

Shellcode file: C:\Users\Administrator\Desktop\log_dst_sc.bin

Options:

- Report Mode
- Scan for API table
- Unlimited steps
- FindSc
- Start Offset 0x0
- Create Dump
- Use Interactive Hooks
- Debug Shell
- No RW Display
- Monitor DLL Read/Write

Process Command Line: []

Open: []

Manual Arguments: []

Launch

Assembly code (left pane):

```

430e80 GetProcAddress(LoadLibraryA)
430fd2 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(fmt=2,ubuf=62f000, sz=4c
    emu_parse no memory found at 0x0

0 ???? No memory At Address      step: 2859730
eax=00000000  ecx=4c000  edx=62f000  ebx=0
esp=12fe04  ebp=12ffff0  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 600000 (2e552 bytes) dumped successfully to disk
00000000  Alloc 62f000 (4c000 bytes) dumped successfully to disk

```

Memory dump (right pane):

Address	Value	Content
000000	77 06 81 EE 00 00 00 00 80 C5 00 45 4D 66 83 EE	w.....EHF..
000010	00 73 07 55 7C 03 C0 C2 70 5D 8D 12 55 66 83 C9	.s.U...p].Ur..
000020	00 5D 7D 05 0D 00 00 00 E0 00 00 00 00 57 BF	.].....,W.
000030	44 49 00 00 5F F9 58 50 48 58 57 66 BF 9D	DI..._XPPPHXXWF..
000040	00 5F 83 E8 05 0B CO FC 68 0C 15 00 00 0D 00 00h.....
000050	00 00 6A D5 83 C4 04 57 7C 06 81 FF BF 60 00 00	..j...W!....
000060	5E BB F6 F9 E8 0C 15 00 00 5E BE 68 CR EA 0A DCh.....
000070	7E B4 B4 B4 B4 4B 4B 4B B4 B4 B4 B4 B4 B4H33K.....
000080	B4 B4 B4 B4 4B 4B 4B 4B B4 B4 B4 B4 B4 B4H00000000000K.....
000090	4B 4B 4B 4B 4B 4B 4B 4B B4 B4 B4 B4 B4 B4	HRRRRRRRRR.....
0000A0	BE B4 B4 B4 B5
0000B0	B5
0000C0	B5
0000D0	B5
0000E0	B5
0000F0	B5
000100	B5
000110	B5
000120	B5
000130	B5

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**:

Assembly code (left pane):

```

.text:00431AE4 ; int __usercall sub_431AE4@<eax>(int a1@<eax>
.text:00431AE4 sub_431AE4    proc near ; CODE XREF: sub_403575+18p
.text:00431AE4        push 30AA0h ; shellcode size
.text:00431AE9        push eax ; ptr_call_addr
.text:00431AEA        rel si, 20h
.text:00431AEE        stc
.text:00431AEE        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

```

Comments (right pane):

```

// positive sp value has been detected, the output may be wrong!
int __usercall sub_431AE4@<eax>(int a1@<eax>
{
    _DWORD *v2; // [esp-10h] [ebp-10h]
    int v3; // [esp-Ch] [ebp-Ch]
    int v4; // [esp-Bh] [ebp-Bh]
    int v5; // [esp-4h] [ebp-4h]

    return f_load_dll_from_memory(a1, 0x30AA8, v2, v3, v4, v5);
    18];
}

21 calls, 0 strings

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 = 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 == 0x6a4abc5b ) // 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:

```

text:00310E99 mov    [ebp+var_1],eax
text:00310E9A mov    [ebp+var_2],eax
text:00310E9B mov    [ebp+var_3],eax
text:00310E9C add    eax,1
text:00310E9D mov    [ebp+var_4],eax
text:00310E9E add    eax,1
text:00310E9F mov    [ebp+var_5],eax
text:00310EA0 mov    [ebp+var_6],eax
text:00310EA1 mov    [ebp+var_7],eax
text:00310EA2 mov    [ebp+var_8],eax
text:00310EA3 mov    [ebp+var_9],eax
text:00310EA4 mov    [ebp+var_10],eax
text:00310EA5 mov    [ebp+var_11],eax
text:00310EA6 mov    [ebp+var_12],eax
text:00310EA7 add    eax,1
text:00310EA8 mov    [ebp+var_13],eax
text:00310EA9 mov    [ebp+var_14],eax
text:00310EA0 mov    [ebp+var_15],eax
text:00310EA1 mov    [ebp+var_16],eax
text:00310EA2 mov    [ebp+var_17],eax
text:00310EA3 mov    [ebp+var_18],eax
text:00310EA4 mov    [ebp+var_19],eax
text:00310EA5 mov    [ebp+var_20],eax
text:00310EA6 mov    [ebp+var_21],eax
text:00310EA7 mov    [ebp+var_22],eax
text:00310EA8 mov    [ebp+var_23],eax
text:00310EA9 mov    [ebp+var_24],eax
text:00310EA0 mov    [ebp+var_25],eax
text:00310EA1 mov    [ebp+var_26],eax
text:00310EA2 mov    [ebp+var_27],eax
text:00310EA3 mov    [ebp+var_28],eax
text:00310EA4 mov    [ebp+var_29],eax
text:00310EA5 mov    [ebp+var_30],eax
text:00310EA6 mov    [ebp+var_31],eax
text:00310EA7 mov    [ebp+var_32],eax
text:00310EA8 mov    [ebp+var_33],eax
text:00310EA9 mov    [ebp+var_34],eax
text:00310EA0 mov    [ebp+var_35],eax
text:00310EA1 mov    [ebp+var_36],eax
text:00310EA2 mov    [ebp+var_37],eax
text:00310EA3 mov    [ebp+var_38],eax
text:00310EA4 mov    [ebp+var_39],eax
text:00310EA5 mov    [ebp+var_40],eax
text:00310EA6 mov    [ebp+var_41],eax
text:00310EA7 mov    [ebp+var_42],eax
text:00310EA8 mov    [ebp+var_43],eax
text:00310EA9 mov    [ebp+var_44],eax
text:00310EA0 mov    [ebp+var_45],eax
text:00310EA1 mov    [ebp+var_46],eax
text:00310EA2 mov    [ebp+var_47],eax
text:00310EA3 mov    [ebp+var_48],eax
text:00310EA4 mov    [ebp+var_49],eax
text:00310EA5 mov    [ebp+var_50],eax
text:00310EA6 mov    [ebp+var_51],eax
text:00310EA7 mov    [ebp+var_52],eax
text:00310EA8 mov    [ebp+var_53],eax
text:00310EA9 mov    [ebp+var_54],eax
text:00310EA0 mov    [ebp+var_55],eax
text:00310EA1 mov    [ebp+var_56],eax
text:00310EA2 mov    [ebp+var_57],eax
text:00310EA3 mov    [ebp+var_58],eax
text:00310EA4 mov    [ebp+var_59],eax
text:00310EA5 mov    [ebp+var_60],eax
text:00310EA6 mov    [ebp+var_61],eax
text:00310EA7 mov    [ebp+var_62],eax
text:00310EA8 mov    [ebp+var_63],eax
text:00310EA9 mov    [ebp+var_64],eax
text:00310EA0 mov    [ebp+var_65],eax
text:00310EA1 mov    [ebp+var_66],eax
text:00310EA2 mov    [ebp+var_67],eax
text:00310EA3 mov    [ebp+var_68],eax
text:00310EA4 mov    [ebp+var_69],eax
text:00310EA5 mov    [ebp+var_70],eax
text:00310EA6 mov    [ebp+var_71],eax
text:00310EA7 mov    [ebp+var_72],eax
text:00310EA8 mov    [ebp+var_73],eax
text:00310EA9 mov    [ebp+var_74],eax
text:00310EA0 mov    [ebp+var_75],eax
text:00310EA1 mov    [ebp+var_76],eax
text:00310EA2 mov    [ebp+var_77],eax
text:00310EA3 mov    [ebp+var_78],eax
text:00310EA4 mov    [ebp+var_79],eax
text:00310EA5 mov    [ebp+var_80],eax
text:00310EA6 mov    [ebp+var_81],eax
text:00310EA7 mov    [ebp+var_82],eax
text:00310EA8 mov    [ebp+var_83],eax
text:00310EA9 mov    [ebp+var_84],eax
text:00310EA0 mov    [ebp+var_85],eax
text:00310EA1 mov    [ebp+var_86],eax
text:00310EA2 mov    [ebp+var_87],eax
text:00310EA3 mov    [ebp+var_88],eax
text:00310EA4 mov    [ebp+var_89],eax
text:00310EA5 mov    [ebp+var_90],eax
text:00310EA6 mov    [ebp+var_91],eax
text:00310EA7 mov    [ebp+var_92],eax
text:00310EA8 mov    [ebp+var_93],eax
text:00310EA9 mov    [ebp+var_94],eax
text:00310EA0 mov    [ebp+var_95],eax
text:00310EA1 mov    [ebp+var_96],eax
text:00310EA2 mov    [ebp+var_97],eax
text:00310EA3 mov    [ebp+var_98],eax
text:00310EA4 mov    [ebp+var_99],eax
text:00310EA5 mov    [ebp+var_100],eax
text:00310EA6 mov    [ebp+var_101],eax
text:00310EA7 mov    [ebp+var_102],eax
text:00310EA8 mov    [ebp+var_103],eax
text:00310EA9 mov    [ebp+var_104],eax
text:00310EA0 mov    [ebp+var_105],eax
text:00310EA1 mov    [ebp+var_106],eax
text:00310EA2 mov    [ebp+var_107],eax
text:00310EA3 mov    [ebp+var_108],eax
text:00310EA4 mov    [ebp+var_109],eax
text:00310EA5 mov    [ebp+var_110],eax
text:00310EA6 mov    [ebp+var_111],eax
text:00310EA7 mov    [ebp+var_112],eax
text:00310EA8 mov    [ebp+var_113],eax
text:00310EA9 mov    [ebp+var_114],eax
text:00310EA0 mov    [ebp+var_115],eax
text:00310EA1 mov    [ebp+var_116],eax
text:00310EA2 mov    [ebp+var_117],eax
text:00310EA3 mov    [ebp+var_118],eax
text:00310EA4 mov    [ebp+var_119],eax
text:00310EA5 mov    [ebp+var_120],eax
text:00310EA6 mov    [ebp+var_121],eax
text:00310EA7 mov    [ebp+var_122],eax
text:00310EA8 mov    [ebp+var_123],eax
text:00310EA9 mov    [ebp+var_124],eax
text:00310EA0 mov    [ebp+var_125],eax
text:00310EA1 mov    [ebp+var_126],eax
text:00310EA2 mov    [ebp+var_127],eax
text:00310EA3 mov    [ebp+var_128],eax
text:00310EA4 mov    [ebp+var_129],eax
text:00310EA5 mov    [ebp+var_130],eax
text:00310EA6 mov    [ebp+var_131],eax
text:00310EA7 mov    [ebp+var_132],eax
text:00310EA8 mov    [ebp+var_133],eax
text:00310EA9 mov    [ebp+var_134],eax
text:00310EA0 mov    [ebp+var_135],eax
text:00310EA1 mov    [ebp+var_136],eax
text:00310EA2 mov    [ebp+var_137],eax
text:00310EA3 mov    [ebp+var_138],eax
text:00310EA4 mov    [ebp+var_139],eax
text:00310EA5 mov    [ebp+var_140],eax
text:00310EA6 mov    [ebp+var_141],eax
text:00310EA7 mov    [ebp+var_142],eax
text:00310EA8 mov    [ebp+var_143],eax
text:00310EA9 mov    [ebp+var_144],eax
text:00310EA0 mov    [ebp+var_145],eax
text:00310EA1 mov    [ebp+var_146],eax
text:00310EA2 mov    [ebp+var_147],eax
text:00310EA3 mov    [ebp+var_148],eax
text:00310EA4 mov    [ebp+var_149],eax
text:00310EA5 mov    [ebp+var_150],eax
text:00310EA6 mov    [ebp+var_151],eax
text:00310EA7 mov    [ebp+var_152],eax

```

LoadLibraryA
VirtualAlloc
VirtualFree
VirtualProtect
ExitThread
RtlDecompressBuffer
memcpy

Next, the shellcode performs a memory allocation (**compressed_buf**) of size **0x2E552**, then reads data from offset **0x1592** (on disk) and executes an xor loop with a key is **0x72** to fill data into the **compressed_buf**. In fact, the size of **compressed_buf** is **0x2E542**, but its first 16 bytes are used to store information about **signature**, **uncompressed_size**, **compressed_size**, so **0x10** is added.

Shellcode continues to allocate memory (**uncompressed_buf**) of size **0x4C000** and calls the **RtlDecompressBuffer** function to decompress the data at the **compressed_buf** into **uncompressed_buf** with the compression format is **COMPRESSION_FORMAT_LZNT1**.

```

signature = *ptr_enc_compressed_dll_addr;                                // ptr_enc_compressed_dll_addr = 0x1592 (offset on disk)
// signature = 0xC7EA9B1C
// xor_key = 0xE70F172

xor_key = signature - 0x7979A9AA;
// 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;
dmSize = computed_compressed_size + 0x10;                                // dmSize = 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 < dmSize; ++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,
        &compressed_buf->compressed_buf,
        compressed_buf->compressed_size,                                         // 0x2E542
        &final_uncompressed_size ) )
{
    return 0xD;
}
if ( uncompressed_buf_size ≠ final_uncompressed_size )

```

1

2

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 1AvbBHjDLuMBTkWW
00000010 45 78 5A 45 4F 6F 54 65 79 70 75 44 63 4B 4E 45 ExZEOoTeypuDcKNE
00000020 74 6C 73 50 61 40 40 78 69 5A 7A 4A 6E 4E 6E 74 tlsPeHHxiZzJnNnt
00000030 69 49 46 4C 42 43 4F 59 50 58 54 00 00 00 00 00 iIFLBCOYPTXT.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 poKHMuPFEEqEsgqa
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 izauUnZjxyEQbmvBi
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 6E 61 58 52 71 4E 59 63 6C uruhaiqoaXRgnYcl
000000B0 75 4E 58 72 4C 44 42 69 48 49 65 67 56 43 75 48 uNXrLDBiHiegVCuH
000000C0 77 73 77 48 68 53 68 45 72 4B 77 68 55 6C 52 78 wswHhSkfrKwhUlRx
000000D0 4C 44 6B 46 42 64 59 79 4C 6E 79 72 50 52 71 54 LDkFBdYyLnryPRqT
000000E0 53 6C 00 00 4C 01 03 00 30 83 1E 53 00 00 00 00 S1..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 00 10 00 10 00 00 00 02 00 .....
00000120 05 00 01 00 00 ..00..05..00..00..00..00..00..00..00 .....
00000130 00 E0 04 00 00 00 00 00 00 00 00 00 00 00 00 01 ..à.....@.
00000140 00 00 10 00 00 10 00 00 00 00 00 10 00 00 10 00 .....
00000150 00 00 00 00 10 00 00 00 60 8F 04 00 45 00 00 00 .....`..E...
00000160 30 91 04 00 78 00 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 00 .....
00000180 00 A0 04 00 0C 33 00 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 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 00 .....
000001D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..text...
000001E0 A5 7F 04 00 00 10 00 00 00 80 04 00 00 04 00 00 ¥.....€...
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 90 04 00 .idata..@...
00000210 00 08 00 00 00 84 04 00 00 00 00 00 00 00 00 00 .....
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 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**:

The screenshot shows the PE-bear interface with the following details:

- Disasm:** .text, General, DOS Hdr, File Hdr, Optional Hdr, Section Hdrs, Experts
- Header:** Characteristics: 0, TimeStamp: 612C95CD (Monday, 30.08.2021 08:24:45 UTC), MajorVersion: 0, MinorVersion: 0, Name: RFPmzNfQQFPXX.
- Exports:** One entry: Main at address 48F90.
- Exported Functions:** [1 entry]: Main at address 48F90.

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 main task of malware:

```

plugx_decrypted_dll = plugx_mapped_dll;
// 00700000 00 00 00 00 29 00 6C 02 A8 0A 03 00 92 15 6C 02 ....).l."...'.l.
// 00700010 52 E5 02 00 69 00 6C 02 0C 15 00 00 00 00 00 00 R...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 98 ....
plugx_decrypted_dll->encrypted_PlugX_size = compressed_dll_size; // 0x2E552
plugx_decrypted_dll->config = config; // 0x002069 (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 use.

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:10027AC4
.text:10027AC6 094      lea     eax, [ebp+ProcName]
.text:10027AC9 094      push    eax
.text:10027ACA 098      call    f_retrieve_kernel32_handle
.text:10027ACA
.text:10027ACF 098      push    eax
.text:10027AD0 09C      call    edi ; GetProcAddress
.text:10027AD0
.text:10027AD2 094      mov     eax, GetCurrentProcess_0, eax
.text:10027AD2
.text:10027AD7
.text:10027AD7 loc_10027AD7: ; CODE XREF: f_check_and_enable_privilege
.text:10027AD7 094      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:[%4s],
//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: [%d], P
v13 = _mm_load_si128(&xmmword_10007240);
// roxy: [%d:%s:%d:
v14 = _mm_load_si128(&xmmword_10007180);
// Protocol:[%4s], Host: [%s:%d], Proxy: [%d:%s:%d:%s]\r\n
wsprintfA(
    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_retrn_buf();
if ( map_file_buf )
{
    strcpy(cmd_input_cmd[0], "Disk");
    f_map_file_buf(0xFFFFFFF, 0, 0x20120320, f_perform_disk_action_command);
}
f_perform_keylogger();
v15 = sub_100175F0();
if ( v15 )
{
    strcpy(cmd_input_cmd[0], "Nethood");
    (*v16)(0xFFFFFFF, 0, 0x20120213, f_enumerate_network_resources, &sz_input_cmd[0]);
}
v16 = sub_100174D0();
if ( v16 )
{
    strcpy(cmd_input_cmd[0], "Netstat");
    (*v16)(0xFFFFFFF, 0, 0x20120215, f_retrieve_network_statistics, &sz_input_cmd[0]);
}
v17 = sub_10018000();
if ( v17 )
{
    strcpy(cmd_input_cmd[0], "Option");
    (*v17)(0xFFFFFFF, 0, 0x2012012B, f_perform_option_sub_command, &sz_input_cmd[0]);
}
v18 = sub_10019000();
if ( v18 )
{
    strcpy(cmd_input_cmd[0], "PortMap");
    (*v18)(0xFFFFFFF, 0, 0x2012032B, f_start_port_mapping, &sz_input_cmd[0]);
}
v19 = sub_10019A10();
if ( v19 )
{
    strcpy(cmd_input_cmd[0], "Process");
    (*v19)(0xFFFFFFF, 0, 0x20120204, f_perform_process_sub_command, &sz_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 0x3060:
        result = f_file_action(a1, cmd_info); // file copy/ rename/ delete/ move
        break;
    case 0x30E1:
        result = f_get_expanded_environment_string(a1, cmd_info);
        break;
    default:
        result = 0xFFFFFFF;
        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.

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

old PlugX sample

```
.data:1001E000 _data          segment para pul
.data:1001E000 assume cs:_data
.data:1001E000 ;org 1001E000h
.data:1001E000 encoded_config_data db 0D9h ; Ü
.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 ; @
.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
```

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;
}

```

decrypt PlugX config



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**.

```

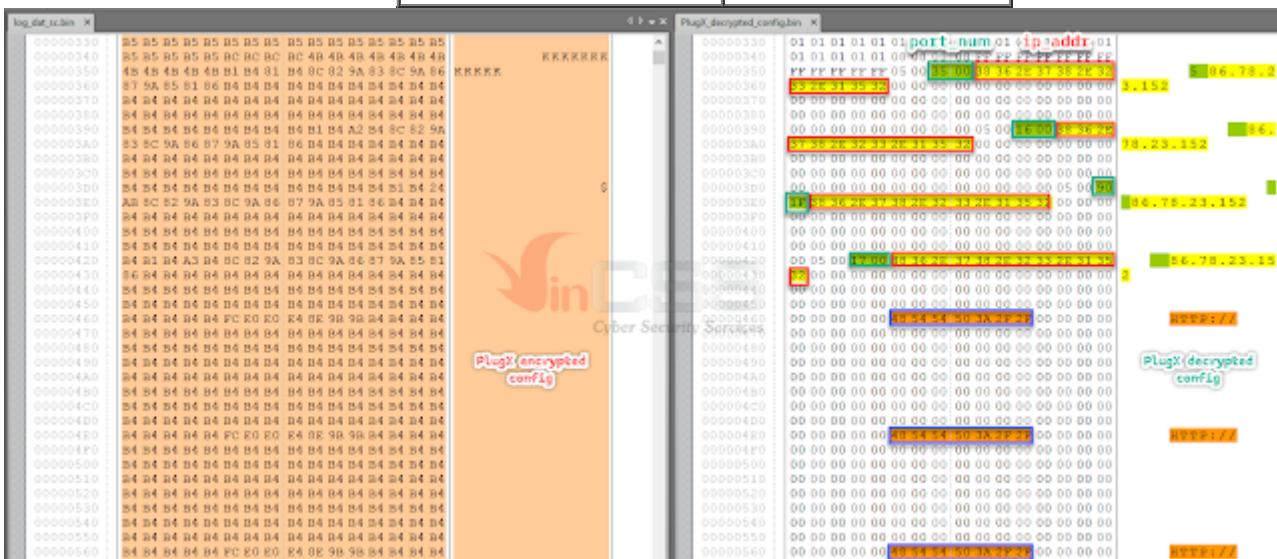
    plugx_mapped_dll->signature = 0;
    plugx_decrypt.dll->ptr_shellcode_base = ptr_call_addr; // 00400209 EB 00 00 00 00
    plugx_decrypt.dll->shellcode_size = end_sc_offset;
    plugx_decrypt.dll->ptr_encrypted_PlugX = ptr_enc_compressed_dll_addr; // 00403592 1C 98 ....
    plugx_decrypt.dll->encrypted_PlugX_size = compressed_dll_size; // 0x2E552
    plugx_decrypt.dll->PlugX_config = config; // 0x00070609 (offset 0x69 on disk)
    plugx_decrypt.dll->PlugX_config_size = config_size; // 0x8150C
    plugx_decrypt.dll->ptr_PlugX_entry_point = plugx_mapped_dll + payload_low_size + payload_nt_headers + optionalHeader.AddressOfEntry+0int;
    VirtualProtect(lpAddress, payload_low_size, PAGE_EXECUTE_READWRITE, &fOldProtect);
    #if (!plugx_decrypt.dll->ptr_PlugX_entry_point)(plugx_mapped_dll, 1, 0 )
        return 0x1B;
    #endif
    if ( ExportProc )
        ExportProc();
    else
        // execute export function

```

The code handles the decryption and execution of the PlugX shellcode. It reads the configuration from memory, decrypts it using a key (0x84), and then executes the decrypted code.

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



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 BDReinit.exe
wsz_bdreinit_exe[0] = 'd\0b';
wsz_bdreinit_exe[1] = 'e\0r';
wsz_bdreinit_exe[2] = 'n\0i';
wsz_bdreinit_exe[3] = 't\0i';
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	t 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 e
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 42 00 69	00 74 00 44 00 65 00 66	B i t D e f
000009D0	00 65 00 6B 00 64 00 65	00 72 00 20 00 43 00 72	e n d e r C r
000009E0	00 61 00 73 00 68 00 20	00 48 00 61 00 6E 00 64	a s h H a n d
000009F0	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 motivated APT groups, applying sophisticated techniques to infect and install malware, aims 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 and upgrade 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 - 604b202cbe5e97c7c8a74a12e1f08e843c08ae08be34dc60b8518b9417c133a9
log.dll - da28eb4f4a66c2561ce1b9e827cb7c0e4b10afe0ee3efd82e3cc2110178c9b7a
log.dat - 2de77804e2bd9b843a826f194389c2605fcf17fd2fafde1b8eb2f819fc6c0c84

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)