

# [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.

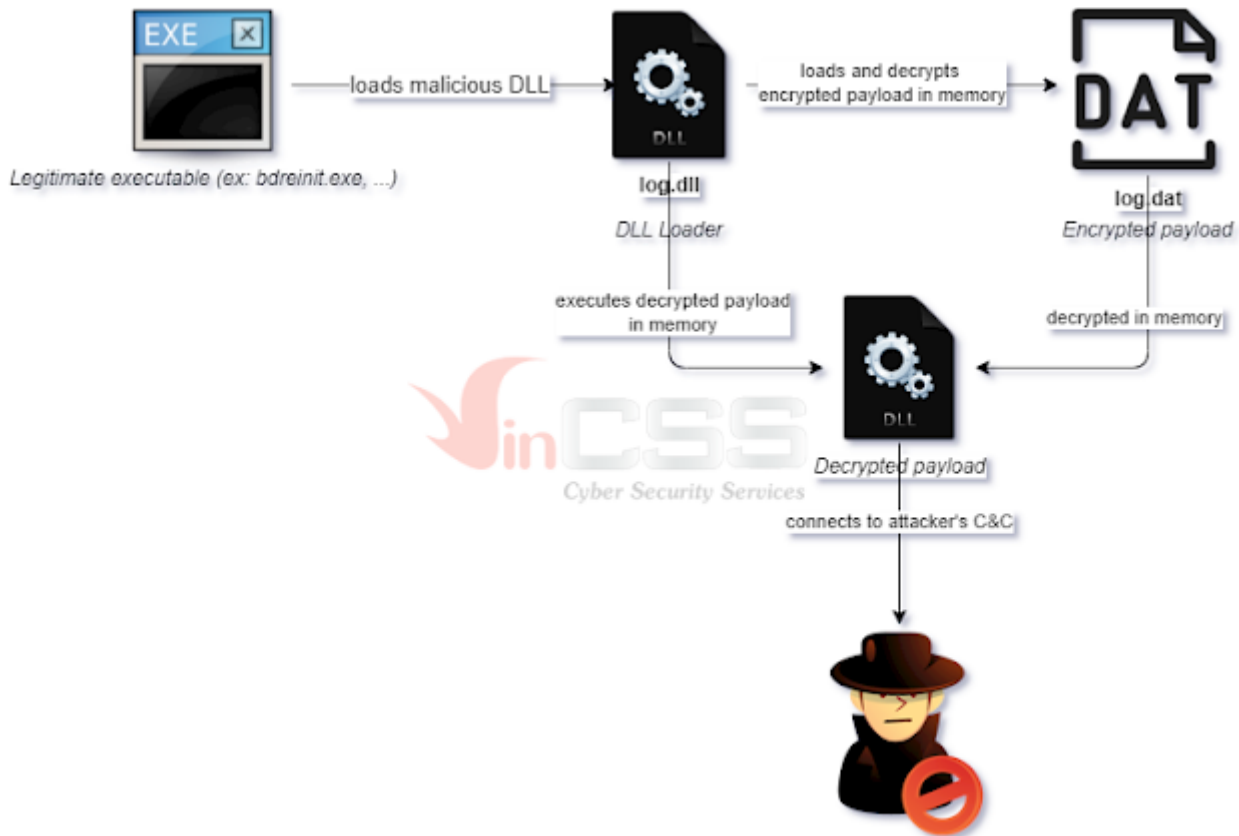
All of these samples share the same name as “log.dll” and have a rather low detection rate.



Hash	Detections	Size	First seen	Last seen	Submitters
08C98045A0338F448C7280881F8DC480545843282C2136681E8846987721	11 / 68	864.00 KB	2022-05-07 01:33:18	2022-05-07 01:33:18	1
84893f360ac388a68f8e4840a50789688852f76a7c251430e8e58ec980c50	9 / 68	103.00 KB	2022-05-05 12:42:34	2022-05-05 17:38:50	2
5171285c48463680579688f530c48a5c988fe32800e18cf802689122576f4e	13 / 67	377.50 KB	2022-04-25 14:04:36	2022-04-25 14:04:36	1
6848282c815c97c7c8a74412e1f88e843c88a288e1340c6885188947c133a9	13 / 69	52.00 KB	2022-04-12 02:36:42	2022-04-12 02:36:42	1
0a28e84f46a6c2561ce189e827c07c8e4b184f8e3ef082e3cc2118178c007a	10 / 55	575.00 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:

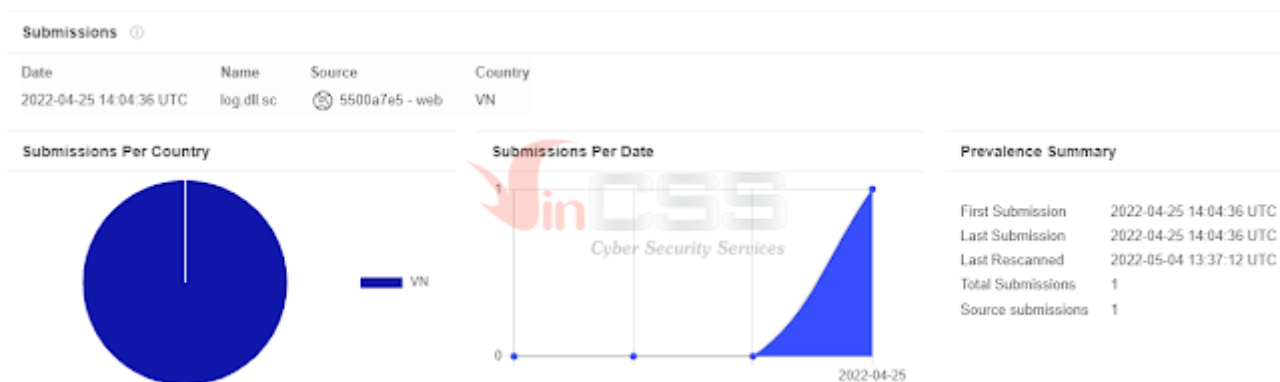
- Technical analysis of the **log.dll** file.
- Technical analysis of shellcode decrypted from **log.dat**.
- Analyze **PlugX Dll** as well as decrypt PlugX configuration information.

## 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)
<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**:

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	

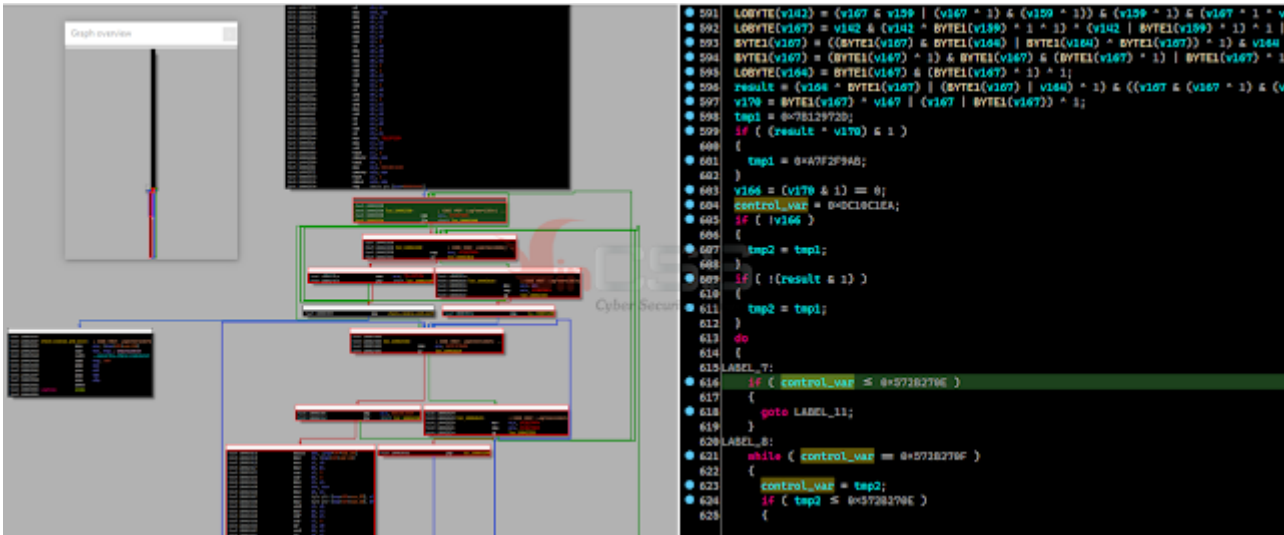
  

Exported Functions [ 2 entries ]					
Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
5BCB8	1	1000	5D0D5	LogFree	
5CBC	2	4E5E0	5D0DD	LogInit	

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

- **LogFree** function:

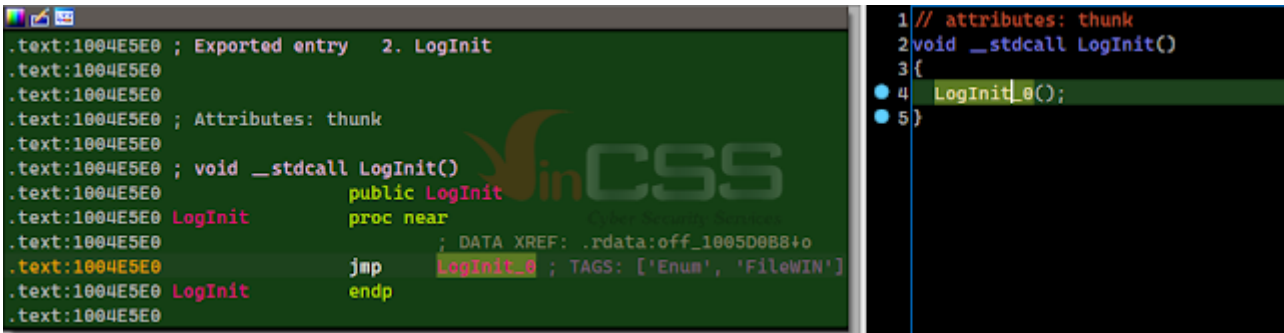
Looking at this function, it can be seen that its code has been completely obfuscated by **Obfuscator-LLVM**, using the **Control Flow Flattening** technique:



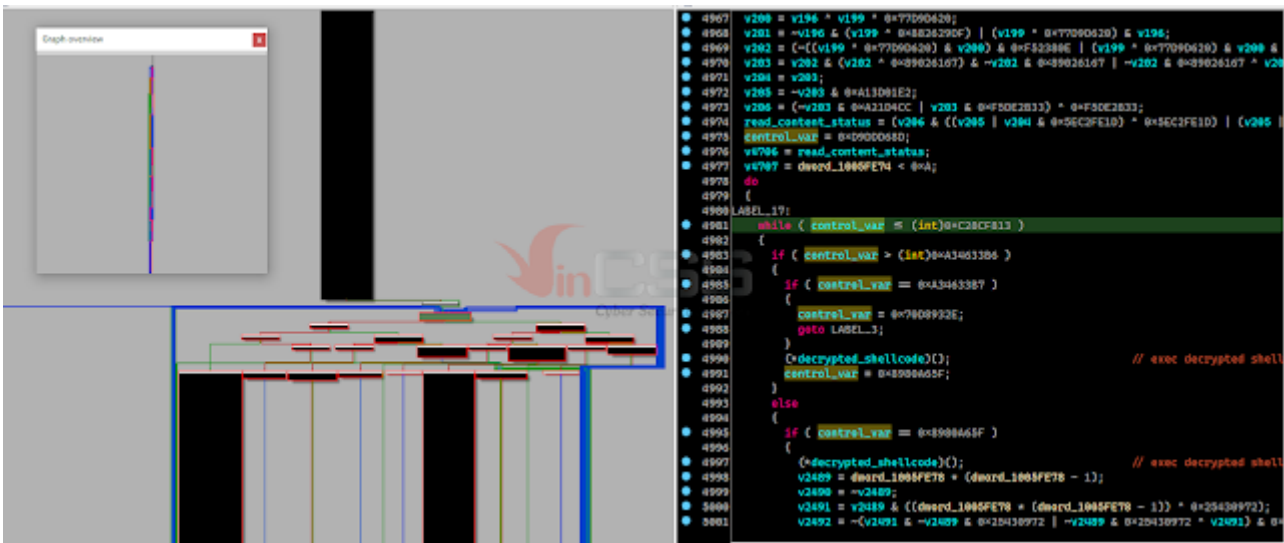
After further analysis, I found that this function has no special task.

- **LogInit** function:

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



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:



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+o
jmp LogInit_9 ; TAGS: ['Enum', 'File\IN']
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:

```

69214 {
69215     break;
69216 }
69217 LABEL_17:
69218     IF ( control_var ≤ 0x2BE93A )
69219     {
69220         goto LABEL_18;
69221     }
69222
69223     kernel32_handle = GetModuleHandleW(aszKernel32);
69224     v095 = dword_1005FEA8 + (dword_1005FEA8 - 1);
69225     log_dat_content_id = ~v095;
69226     v096 = ((dword_1005FEA8 * (dword_1005FEA8 - 1)) & 0x2E40B691) ~v095 & 0x01B27966;
69227     v097 = (((~v096 & 0x274BCC9 | v096 & 0xFDB8433F) * 0x103090) & 0x4C98311E | ((~v096
69228     v098 = ~v095 & (~v095 & 0xA8D7B9FE | (dword_1005FEA8 * (dword_1005FEA8 - 1)) & 0x
69229     v099 = v097 & 0x73D698D5 | ~v097 & 0x3C29642A;
69230     v010 = ((v098 | v097) & 0x37D7B6D2 | ~(v098 | v097) & 0xC52B492D) * (~v099 * (v0
69231     v011 = v010 & (v010 * 0x600D028) & ~v010 & 0x0D0D028 | ~v010 & 0x600D028 * v01
69232     v012 = ~v011 & 0x59FAB490 | v011 & 0xA0854B6F;
69233     v013 = ~v012 & v012 | ~v012 | ~v012 * ~v012 & v012;
69234     v014 = ~v013;
69235     v015 = v013 & 0x14B684UF & v014 & (v013 * 0xEB497B00) | v014 & (v013 * 0xEB497B00
69236     v016 = (~v015 & 0x26982EUC | v015 & 0xD964C1B3) * 0xCDD245FD;
69237     v017 = (~v014 & 0xFFFFFFFF | v013 * 1) & v016 | v016 * (~v014 & 0xFFFFFFFF | v013
69238     v018 = v017 & (v017 * 0x2DAB84AF) & ~v017 & 0x2DAB84AF | ~v017 & 0x2DAB84AF * v01
69239     v019 = v095 & ((dword_1005FEA8 * (dword_1005FEA8 - 1)) ^ 0x11D587DC);
69240     v020 = (~v018 & 0x02B54B50 | v018 & 0x2DAB84AF) & 0x6A99A520 | (~v018 & 0x02B54B
69241     v021 = (v020 * 0x2218A500) & 0xA2BEEF51 | (v020 * 0x1B20108E) & 0x3DA110AE;
69242     v022 = (v021 * 0x001108E) & 0x8F4F520F | (v021 * 0x2010AD50) & 0x7000ADF0;
69243     v023 = (~v019 & ~v095 & 0x11D587DC | ~v095 & 0x11D587DC * v019) & 0x70057966 | (v
69244     v024 = (v023 * 0x93FF3145) & 0xFFFFFFFF;
69245     v025 = (v023 * 0x6C09CEBA) & 0xFFFFFFFF | (v023 * 0x93FF3145) & 1;
69246     v026 = v025 & v024;
69247     v027 = v024 * v025;
69248     v028 = v022 * 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+document_shellcode]
test eax, ebx
mov ecx, 11
mov [ecx], calls:
mov eax, 7A7
cmpvz eax, edx
cmp eax, 0E1
jg loc_1002
call ds:GetProcAddress ; call GetProcAddress 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.

```

13800 LOGBYTE(v4939) = v4939 & BYTE1(v4939) | BYTE1(v4939) * v4939;
13801 BYTE1(v5065) = -[BYTE]v5065 & 0x1A;
13802 BYTE1(v4939) = ((v5065 & 0x79 | -(BYTE)v5065 & 0x80) * 0x30) & ((v5065 & 0
13803 LOGBYTE(v5350) = v5044;
13804 LOGBYTE(v5064) = -BYTE1(v5064) | v5044;
13805 BYTE1(v5064) = (v5030 & 0x15 | BYTE1(v5065) * -(BYTE)v5044) & 0x45 | BYTE
13806 BYTE1(v5065) = -(BYTE1(v4939) & 0x8E | BYTE1(v4939) & 0x71) * -(BYTE)v50
13807 BYTE1(v5065) = BYTE1(v5065) & -(BYTE1(v5064) | -(BYTE)v5044) | (BYTE1(v504
13808 LOGBYTE(v5065) = -BYTE1(v5065);
13809 log_dat_content[idx] = ((v5065 | v4939) & 0x35 | -(v5065 | v4939) & 0x1A) *
13810 v5046 * idx + 1;
13811 control_var_1 = 0x92E699EC;
13812 }
13813 else if ( control_var_1 == 0x92E699EC )
13814 {
13815 v4341 = -(dword_1005FE00 + (dword_1005FE00 - 1));
13816 v5310 = dword_1005FE00 + (dword_1005FE00 - 1);
13817 v4342 = (v4341 & 0x3A211E02 | (dword_1005FE00 * (dword_1005FE00 - 1)) & 0CE
13818 v4343 = ((-v4342 & 0x70FABA20 | v4342 & 0x4F06600F) * 0x1587001) & 0x1D89F0
13819 v4344 = (v4343 * 0x17FB289C) & (v4343 * 0x276032E);
13820 v4345 = -v4344;
13821 v4346 = (((v4341 & 0x7360720 | (dword_1005FE00 + (dword_1005FE00 - 1)) & 0
13822 v4347 = ((v4346 | v4345) & 0x02021F1 | -(v4346 | v4345) & 0x4FD7CE0C) * (0

```

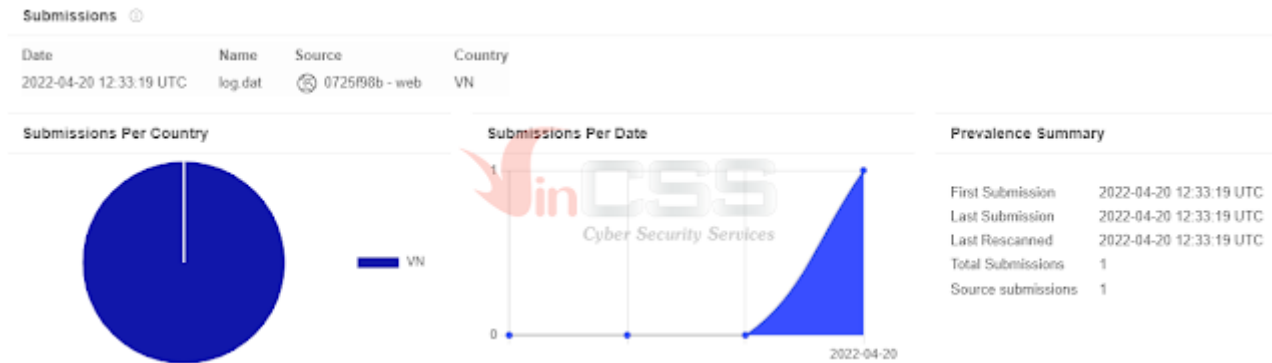
### 3. Shellcode analysis

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
3268DC1CE6c29206f14b120E22f691A76428A5c2882c4715F2E8f9c19E8B log.dat	0 / 57	194.66 KB	2022-05-07 01:32:51	2022-05-07 01:32:51	1
0208038EA3437544E2788f9f7638A4F260c633A6806FE771882487339FARCS log.dat	2 / 59	189.23 KB	2022-05-05 12:44:31	2022-05-05 12:44:31	1
BE3E2782437A51F88891EE246F347757B71328220850A9C9F18B17538CB log.dat.sc	0 / 57	194.66 KB	2022-04-25 14:07:46	2022-04-25 14:07:46	1
2DE77804E2BD9B843A826f194389c2605cfc17fd2fafde1b8eb2f819fc6c0c84 log.dat	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 (2de77804e2bd9b843a826f194389c2605cfc17fd2fafde1b8eb2f819fc6c0c84) was submitted to VirusTotal on 2022-04-20 12:33:19 UTC (5 days before the above **log.dll** file).

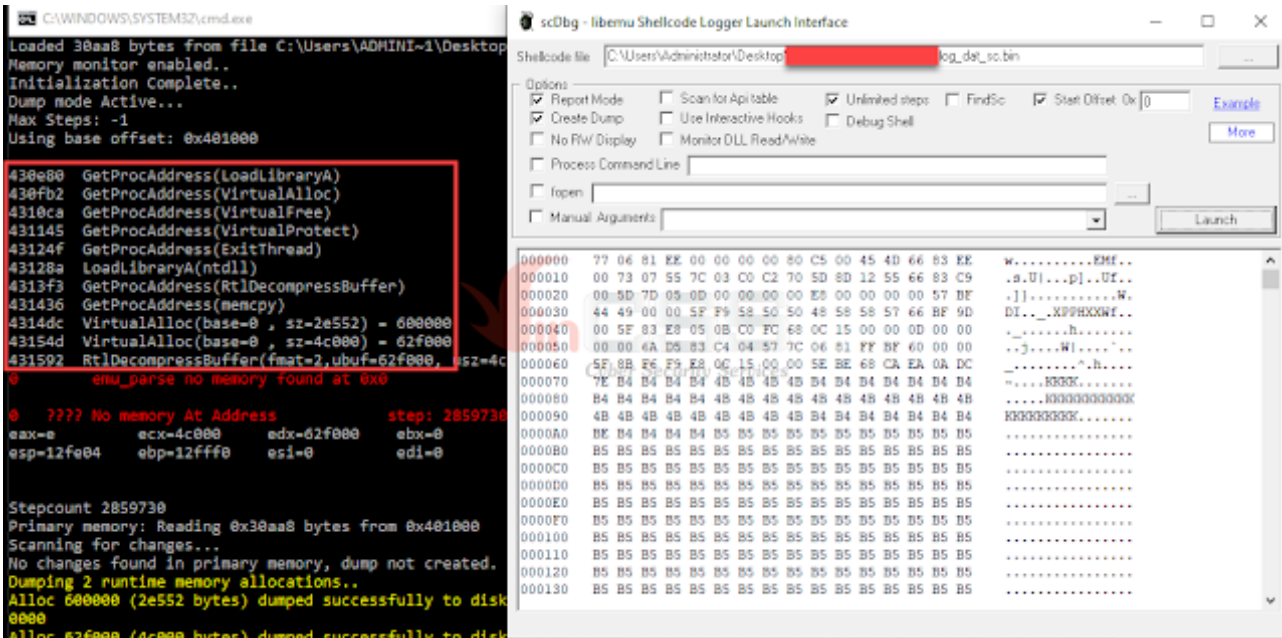


Debugging and dump the decrypted shellcode look like this:

```
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.EMEf1
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 .].....e...Wz
00000030 44 49 00 00 5F F9 58 50 50 48 58 58 57 66 BF 9D DI.._xPPHXWz.
00000040 00 5F 83 E8 05 0B C0 FC 68 0C 15 00 00 0D 00 00 ..fè..Àgh.....
00000050 00 00 6A D5 83 C4 04 57 7C 06 81 FF BF 60 00 00 ..jÖfA.Wj..yç...
00000060 5F 8B F6 F9 E8 0C 15 00 00 5E BE 68 CA EA 0A DC <ôè....^NhÈ.Ü
00000070 7E B4 B4 B4 B4 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B -''''KKKK''''''
00000080 B4 B4 B4 B4 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B ''''''KKKKKKKKKK
00000090 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B KKKKKKKK''''''
000000A0 BE B4 B4 B4 B4 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 N''''''pppppppppp
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
```

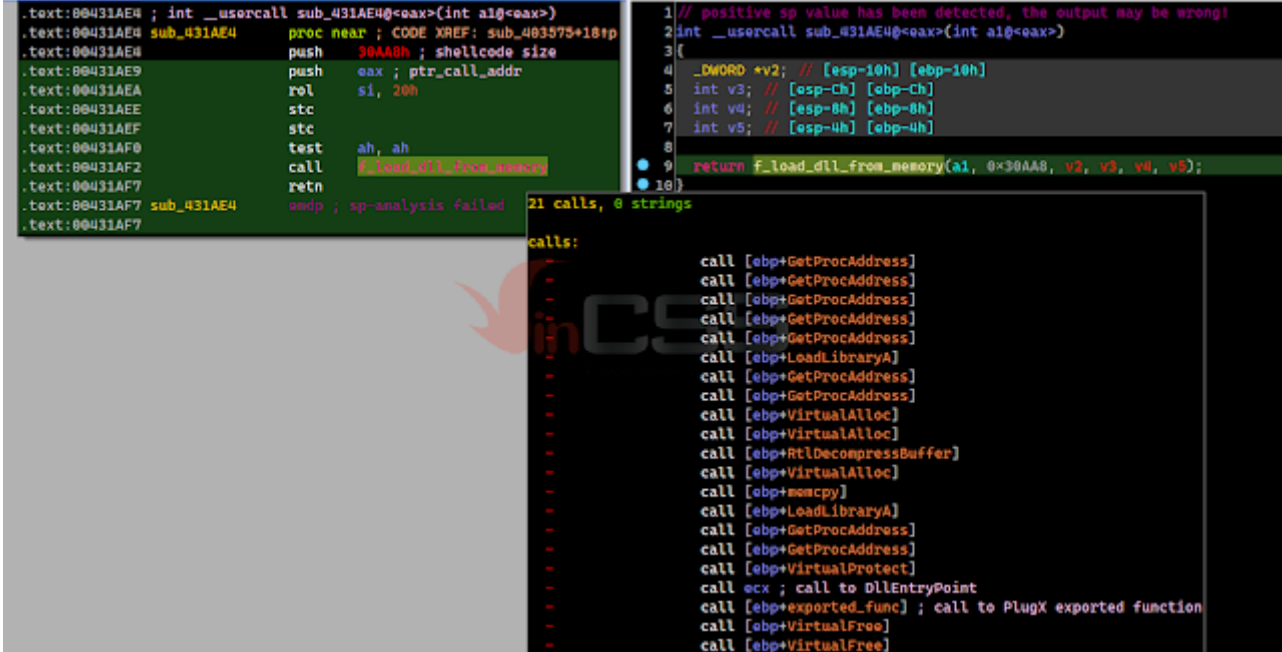
I use two tools, [FLOSS](#) and [scdbg](#) to get an overview of this shellcode. The results can be seen in the screenshots below:

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



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 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:



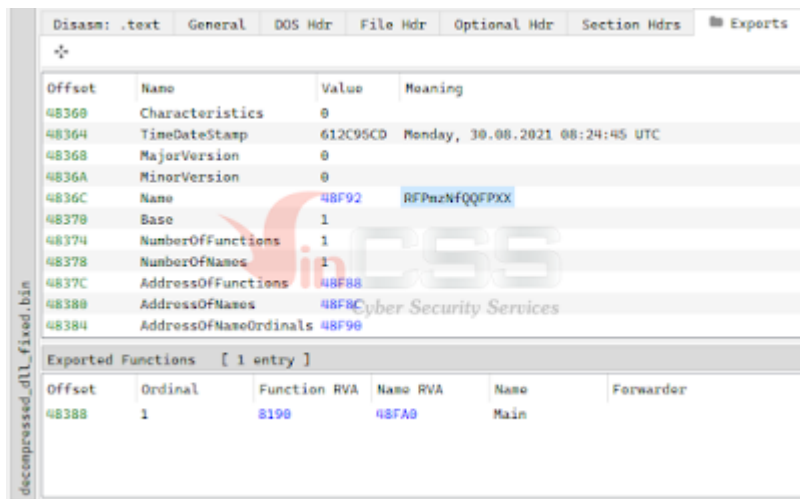
Based on the above analysis results, it is easy to get the extracted Dll file (however, the file header information was destroyed):

```

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 ExZE0oTeypuDcKNE
00000020 74 6C 73 50 61 48 48 78 69 5A 7A 4A 6E 4E 6E 74 t1aPaHHxiZzJnNat
00000030 69 49 46 4C 42 43 4F 59 50 58 54 00 E0 00 00 00 iIFLBCOYPT.â...
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 uOsvUYUvVFXZwzBU
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 uruhaigoaXRqNYcl
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 6B 45 72 4B 77 68 55 6C 52 78 wshHskErKwhULRx
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 ...â.!......
00000100 00 3C 00 00 00 00 00 00 B0 81 00 00 00 10 00 00 <.3.....
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 00 .....
00000130 00 E0 04 00 00 00 00 00 00 00 00 40 01 00 00 00 .â.....@.
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 91 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 50 7A 00 00 40 00 00 00 00 .....Pz..@...
000001B0 00 00 00 00 00 00 00 00 90 04 00 30 01 00 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 2E 74 65 78 74 00 00 00 00 .....text...
000001E0 A5 7F 04 00 00 10 00 00 00 80 04 00 00 04 00 00 Y.....e.....
000001F0 00 00 00 00 00 00 00 00 00 00 00 60 00 00 00 60 .....
00000200 2E 69 64 61 74 61 00 00 D2 07 00 00 00 90 04 00 .idata..ô.....
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 8C 04 00 .3... ..â...@..

```

Fix the header information and check with [PE-bear](#), this Dll has the original name is **RFPmzNfQQFPXX** and only exports one function named **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;
// 00700000 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 RÅ..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 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_1000778A0
.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 ; lpProcName
.text:10027ACA 098      call   f_retrieve_kernel32_handle
.text:10027ACA
.text:10027ACF 098      push   eax ; hModule
.text:10027AD0 09C      call   edi ; GetProcAddress
.text:10027AD0
.text:10027AD2 094      mov    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: [%s:%d], p
v13 = _mm_load_si128(&xmmword_10007240);
// roxy: [%d:%s:%d:
v14 = _mm_load_si128(&xmmword_10007100);
// Protocol:[%4s], Host: [%s:%d], Proxy: [%d:%s:%d:%s:%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_d,
    port_num_hi,
    &src->event_handle_1,
    &src->field_04);
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_bufC);
if ( map_file_buf )
{
    strcpy(&sz_input_cmd[0], "Disk");
    (*map_file_buf)(0xFFFFFFFF, 0, 0x20120320, f_perform_disk_action_command);
}
f_perform_keylogger();
v15 = sub_100175F8();
if ( v15 )
{
    strcpy(&sz_input_cmd[0], "Rehoood");
    (*v15)(0xFFFFFFFF, 5, 0x20120215, f_enumerate_network_resources, &sz_input_cmd[0]);
}
v16 = sub_10017AD0();
if ( v16 )
{
    strcpy(&sz_input_cmd[0], "Necatat");
    (*v16)(0xFFFFFFFF, 4, 0x20120315, f_retrieve_network_statistics, &sz_input_cmd[0]);
}
v17 = sub_10018000();
if ( v17 )
{
    strcpy(&sz_input_cmd[0], "Option");
    (*v17)(0xFFFFFFFF, 6, 0x20120120, f_perform_option_sub_command, &sz_input_cmd[0]);
}
v18 = sub_10019000();
if ( v18 )
{
    strcpy(&sz_input_cmd[0], "PartMap");
    (*v18)(0xFFFFFFFF, 7, 0x20120315, f_start_port_mapping, &sz_input_cmd[0]);
}
v19 = sub_10019A10();
if ( v19 )
{
    strcpy(&sz_input_cmd[0], "Process");
    (*v19)(0xFFFFFFFF, 1, 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_sn_hidden_desktop(a1, cmd_info);
        break;
    case 0x300D:
        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.

```
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 pub
.data:1001E000                assume cs:_data
.data:1001E000                ;org 1001E000h
.data:1001E000 encoded_config_data db 0D9h ; 0
.data:1001E000
.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;
}

```



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**.



```
$ 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 - 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)**