

# TokyoX: DLL side-loading an unknown artifact

---

[lab52.io/blog/tokyox-dll-side-loading-an-unknown-artifact/](https://lab52.io/blog/tokyox-dll-side-loading-an-unknown-artifact/)

During Christmas holidays, Lab52 has been analyzing a sample which loads an artifact that we have decided to refer to as “TokyoX” since no similarities have been found as to any known malware, which we usually detect in open sources. However, we cannot confirm so far that it is indeed a new family of malware.

The first thing we identified was a DLL (382b3d3bb1be4f14dbc1e82a34946a52795288867ed86c6c43e4f981729be4fc) which had the following timestamps in VirusTotal at the time of the current analysis, and was uploaded from Russia via web site:

Creation Time 2021-12-09 02:46:43

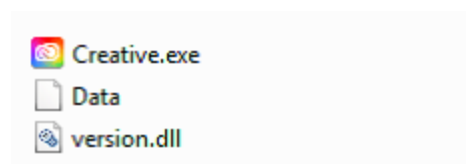
First Submission 2021-12-09 08:48:20

Last Submission 2021-12-09 08:48:20

Last Analysis 2021-12-23 23:38:08

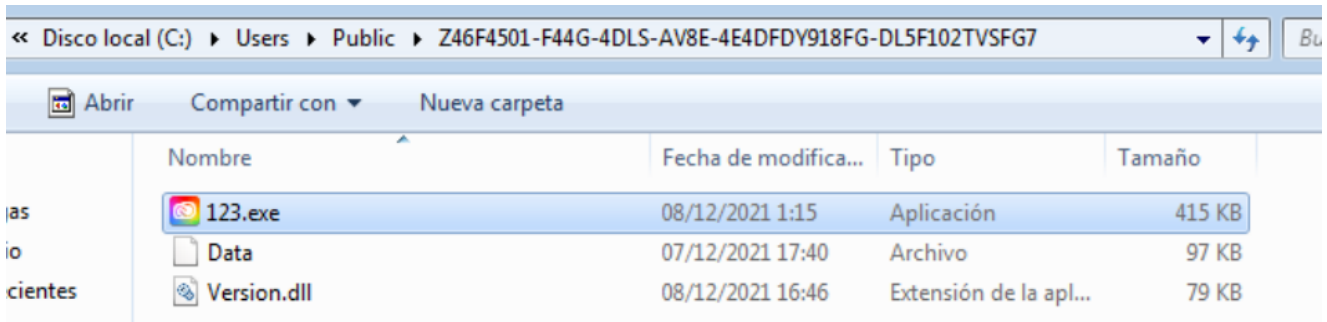
Some antivirus engines tagged the sample as PlugX, but it seems that the attribution might be due to the final payload’s loading mechanism: DLL sideloading with an encrypted payload in the same directory. After analyzing the final payload we could not find any similarities with other known samples from PlugX other than the loading TTPs.

This DLL had a related .zip file with the name планирование.zip (translated to as planning.zip). When unzipping, the following files are observed:



The legitimate file Creative.exe, an encrypted Data file and the version.dll DLL, which implements the loader function for the Data file, and therefore responsible of mapping the “TokyoX”.

If we execute it from a path which is not final or the expected by the malware, it replicates to another path and executes from there, which is something it does have in common with some PlugX dll loaders:

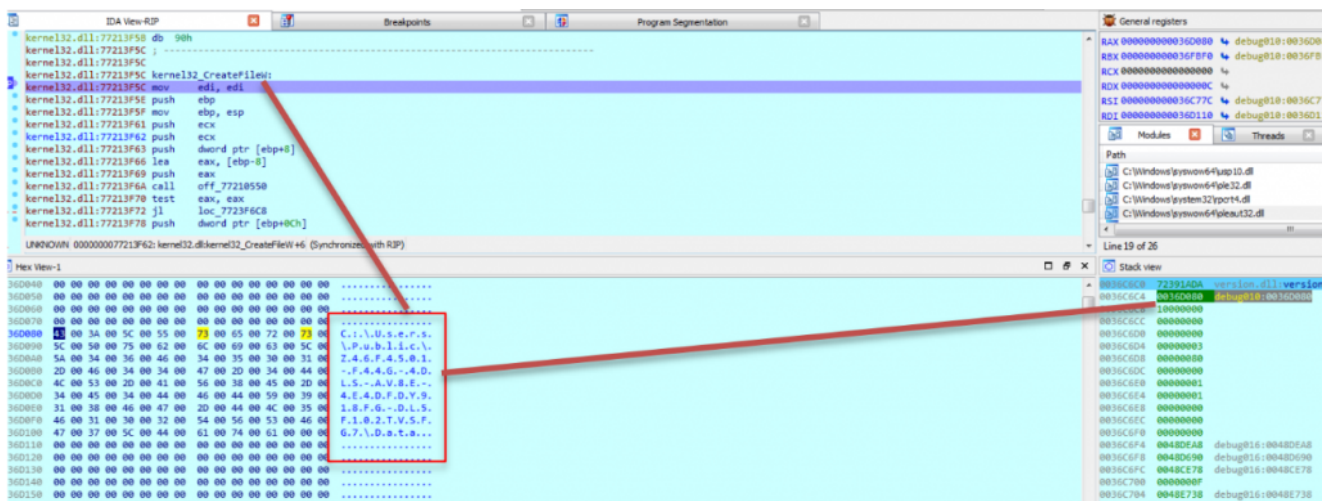


Once executed, we observe how the netsh.exe process tries to establish connections with port 443 of the IP address 31.192.107[.]187.

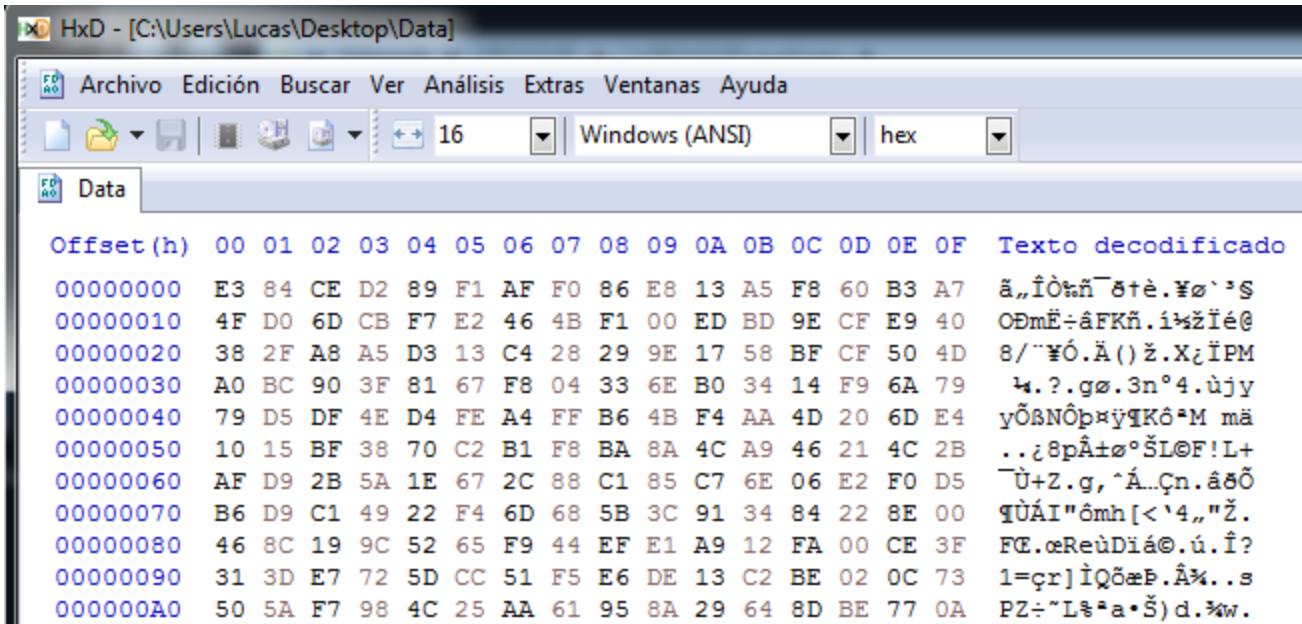
In this analysis we will focus on different aspects about the process; from double-clicking the binary 123.exe process (which is a copy of Creative.exe but in another path) to the execution of “TokyoX” already decrypted in memory.

The first thing we observe within the process is how the version.dll library prepares the decryption and the final payload’s loading in the remote process:

In fact, we can see how the content of the Data file is read in the code section of version.dll:

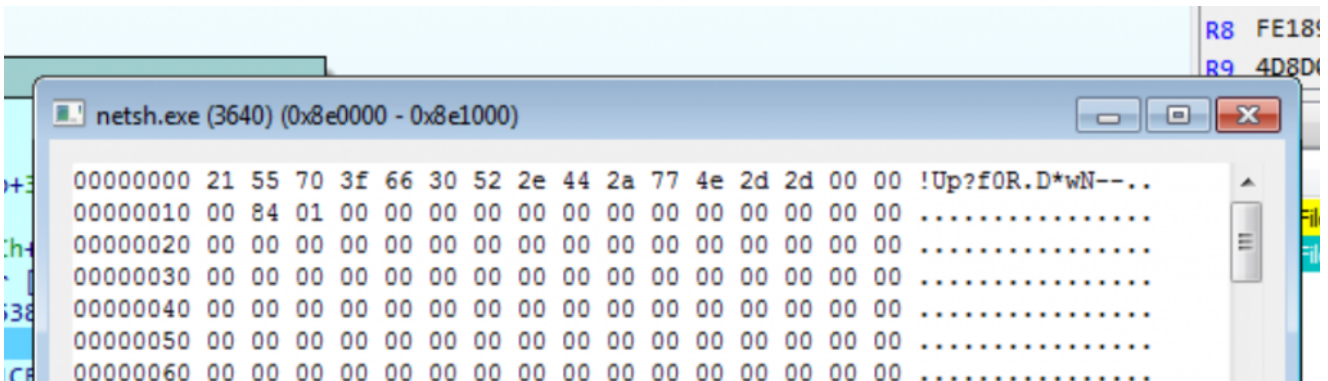


If we edit the Data file with a hexadecimal editor we will see their values, which will help us to identify it in memory later (beginning with E3 84):



After reading the file from disk, a child process netsh.exe is created. This just-created child process is where several new memory segments will be located (a total of 5, including the final decrypted payload) to decrypt the final "TokyoX" payload. The APIs which were observed for the creation and writing of the remote process are the native APIs NtAllocateVirtualmemory and NtwriteVirtualmemory.

First, it creates two segments: 100Kb where the encrypted payload is located and which comes from the disc, and another one of 4Kb. In the 4Kb segment we observe how the following string is set (which will be the string used for the decrypting process):



The other memory segment of 100Kb contains the following (encrypted content, as we see how it matches the content from Data file on Disk):

```

netsh.exe (3640) (0x8c0000 - 0x8d9000)

00000000 e3 84 ce d2 89 f1 af f0 86 e8 13 a5 f8 60 b3 a7 .....`..
00000010 4f d0 6d cb f7 e2 46 4b f1 00 ed bd 9e cf e9 40 O.m...FK.....@
00000020 38 2f a8 a5 d3 13 c4 28 29 9e 17 58 bf cf 50 4d 8/.....()...X..PM
00000030 a0 bc 90 3f 81 67 f8 04 33 6e b0 34 14 f9 6a 79 ...?.g..3n.4..jy
00000040 79 d5 df 4e d4 fe a4 ff b6 4b f4 aa 4d 20 6d e4 y..N.....K..M m.
00000050 10 15 bf 38 70 c2 b1 f8 ba 8a 4c a9 46 21 4c 2b ...8p.....L.F!L+
00000060 af d9 2b 5a 1e 67 2c 88 c1 85 c7 6e 06 e2 f0 d5 ..+Z.g,....n....
00000070 b6 d9 c1 49 22 f4 6d 68 5b 3c 91 34 84 22 8e 00 ...I".mh[<.4."..
00000080 46 8c 19 9c 52 65 f9 44 ef e1 a9 12 fa 00 ce 3f F...Re.D.....?
00000090 31 3d e7 72 5d cc 51 f5 e6 de 13 c2 be 02 0c 73 1=.r].Q.....s
000000a0 50 5a f7 98 4c 25 aa 61 95 8a 29 64 8d be 77 0a PZ..L$.a..)d..w.
000000b0 8a 4d be 97 a6 e3 02 e1 0e 25 2a 40 e2 32 6b 71 .M.....$*@.2kq
000000c0 c0 c8 f2 12 b0 3e 29 0b 05 86 45 39 f4 4d 24 5b .....>)...E9.M$[
000000d0 76 e3 c2 28 2b 91 10 8b 42 9b a4 cf d0 1f 08 0d v..(+...B.....
000000e0 53 75 7f ab b9 8e 2e 70 d2 e7 7c e3 13 ea 9f 83 Su.....p..|.....
000000f0 06 01 d8 49 2f 63 95 21 84 f1 65 92 36 ae 3f 97 ...I/c.!...e.6.?.
00000100 8e 99 1e 38 d2 81 e2 08 53 89 c0 b5 a8 04 a1 94 ...8....S.....
00000110 96 92 16 ac 51 e1 d9 2e bc 45 41 79 70 dd 69 8b ....Q....EAyp.i.
00000120 ca f9 57 71 8e 6a d2 14 49 83 30 b3 b2 6a bc a7 ..Wq.j..I.0..j..
00000130 ae 0a 15 4d 34 27 86 13 d6 9e 89 25 50 0b c6 f4 ...M4'.....$P...
00000140 b7 c1 59 bb 89 d4 9a 56 4f 5b e5 fb 98 68 2e ce ..Y....VO[...h..
00000150 88 7f 3f e4 9e 0b 70 78 3d f4 ed 96 75 0a 84 1e ..?...px=...u...
00000160 61 ea 09 dd 07 ad 1b ed d5 57 8f b5 38 2d 87 c9 a.....W..8-..
00000170 3e 44 b5 13 1a ca c0 ea ab 87 bd ba 9b e1 3b 6c >D.....;1
00000180 40 e6 26 81 18 fb 7f 80 6d cd 24 6a c9 b2 b5 4c @.a.....m.$j...L
00000190 49 c7 de f2 55 60 49 3b 8f f8 2f ee 77 ba d0 c2 I...U'I;../.w...
000001a0 46 d4 77 95 68 ef 1d 4e 35 c2 ca dc 87 0e 79 82 F.w.h..N5.....y.

```

After the creation of these two segments, a third segment is allocated, where it is loaded the absolute memory addresses from several win32 APIs (VirtualAlloc, LoadLibrary, GetProcAddress, the home address of the coded payload, etc.) for its later use by the loader:

```

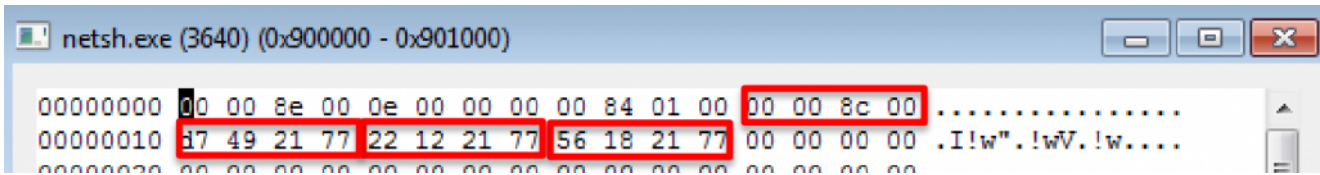
00000000 72651C00
00000000 72651C00 loc_72651C00:
00000000 72651C00 mov     al, [ecx]
00000000 72651C02 inc     ecx
00000000 72651C03 test    al, al
00000000 72651C05 jnz     short loc_72651C00

00000000 72651C07 mov     eax, off_7265D008 ; GetProcAddress
00000000 72651C0C sub     ecx, edx
00000000 72651C0E mov     [esp+31A0h+var_300C], eax
00000000 72651C15 mov     eax, off_7265D044 ; LoadLibrary
00000000 72651C1A push   40h
00000000 72651C1C mov     [esp+31A4h+var_3010], eax
00000000 72651C23 mov     eax, off_7265D048 ; VirtualAlloc
00000000 72651C28 push   3000h
00000000 72651C2D mov     [esp+31A8h+var_3008], eax
00000000 72651C34 lea    eax, [esp+31A8h+var_3194+8]
00000000 72651C38 push   eax

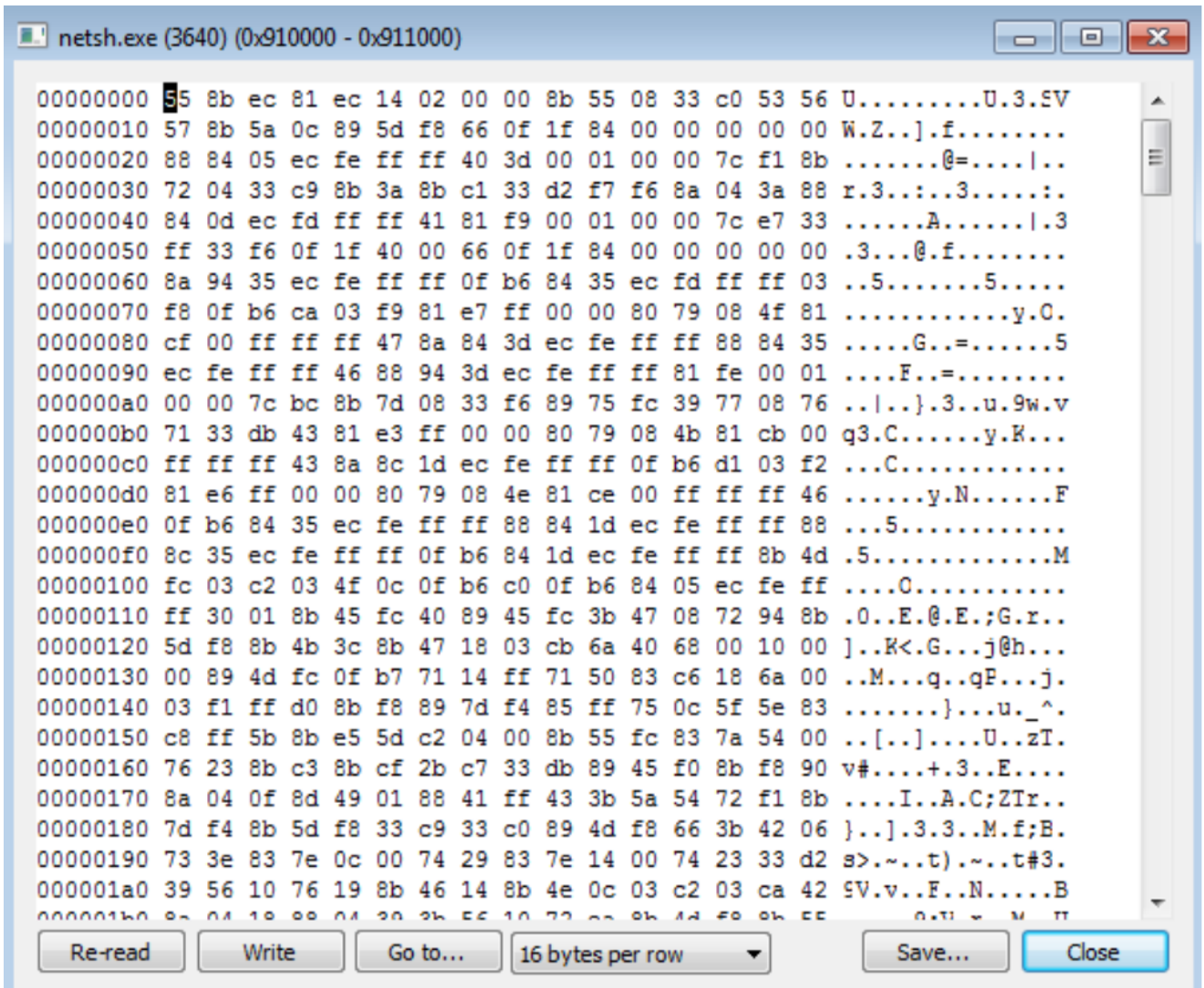
```



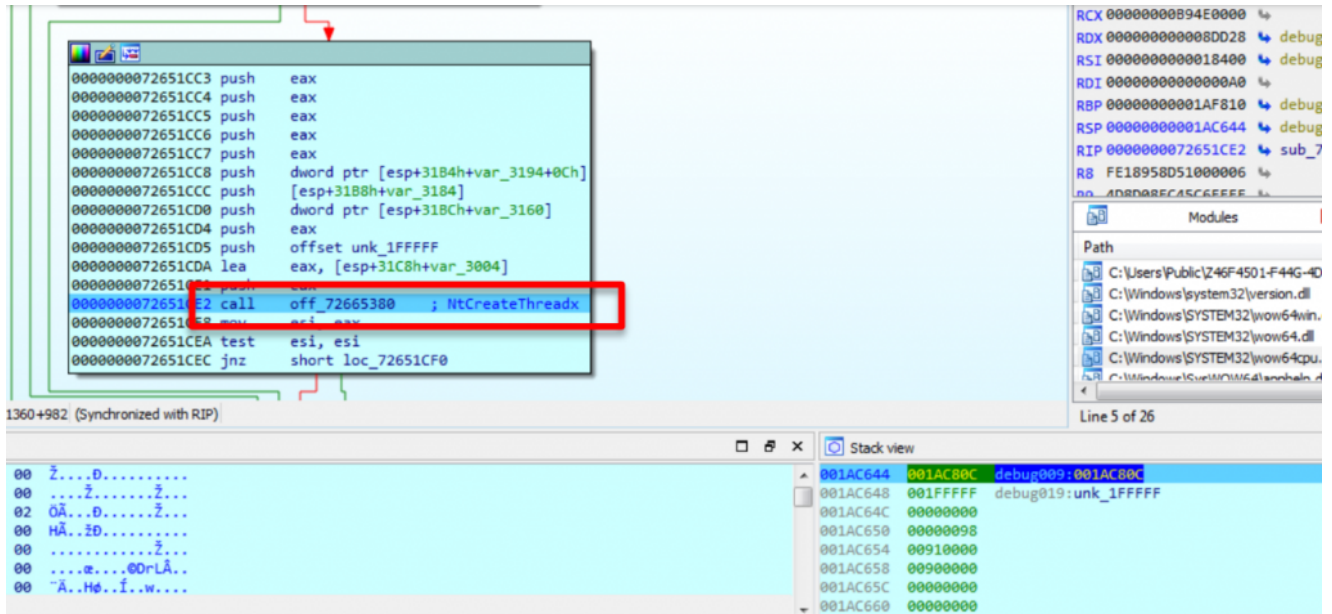
We can notice how the segment will have the memory addresses (starting from 123.exe they are located in netsh.exe segment through the version.dll code):



Then, another segment of 4Kb is created where it loads the code that will decrypt and load the final payload.



Finally, the “TokyoX” loader runs from the DLL (version.dll) in netsh.exe through the API NtcreateThreadEx and we see the start of the last page created in the stack:



After the execution of NtCreateThreadEx, as indicated, the loader is initiated in netsh.exe in the segment:

```

debug039:00910000 55          push    ebp
debug039:00910001 8B EC       mov     ebp, esp
debug039:00910003 81 EC 14 02 00 00    sub     esp, 214h
debug039:00910009 8B 55 08     mov     edx, [ebp+arg_0]
debug039:0091000C 33 C0       xor     eax, eax
debug039:0091000E 53         push   ebx
debug039:0091000F 56         push   esi
debug039:00910010 57         push   edi
debug039:00910011 8B 5A 0C    mov     ebx, [edx+0Ch]
debug039:00910014 89 5D F8    mov     [ebp+var_8], ebx
debug039:00910017 66 0F 1F 84 00 00 00 00  nop     word ptr [eax+eax+00000000h]
debug039:00910020
debug039:00910020                loc_910020:
debug039:00910020 8B 84 05 EC FE FF FF    mov     [ebp+eax+var_114], al
debug039:00910027 40         inc     eax
debug039:00910028 3D 00 01 00 00    cmp     eax, 100h
debug039:0091002D 7C F1     jl     short loc_910020
debug039:0091002F 8B 72 04     mov     esi, [edx+4]
debug039:00910032 33 C9     xor     ecx, ecx
debug039:00910034 8B 3A     mov     edi, [edx]
debug039:00910036

```

Once the execution is moved to the netsh.exe process, it takes the string located in the initial 4Kb segment, copies it into the stack and replicates it (0x100, 256 bytes) to match the specific block size of 256bytes. In the following screenshots we can observe how the block ends with the string “!Up?” when it reaches the value 0x100 in hexadecimal.

```

0000000000910036
0000000000910036 mov_string_to_stack:
0000000000910036 mov     eax, ecx
0000000000910038 xor     edx, edx
000000000091003A div     esi
000000000091003C mov     al, [edx+edi]
000000000091003F mov     [ebp+ecx+var_214], al
0000000000910046 inc     ecx
0000000000910047 cmp     ecx, 256
000000000091004D jl      short mov_string_to_stack

```

```

0313FD50  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0313FD60  00 00 90 00 21 55 70 3F 66 30 52 2E 44 2A 77 4E ...!Up?f0R.D*wN
0313FD70  2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
0313FD80  21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 !Up?f0R.D*wN--!U
0313FD90  70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F p?f0R.D*wN--!Up?
0313FDA0  66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 f0R.D*wN--!Up?f0
0313FDB0  52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E R.D*wN--!Up?f0R.
0313FDC0  44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A D*wN--!Up?f0R.D*
0313FDD0  77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E wN--!Up?f0R.D*wN
0313FDE0  2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
0313FDF0  21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 !Up?f0R.D*wN--!U
0313FE00  70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F p?f0R.D*wN--!Up?
0313FE10  66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 f0R.D*wN--!Up?f0
0313FE20  52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E R.D*wN--!Up?f0R.
0313FE30  44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A D*wN--!Up?f0R.D*
0313FE40  77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E wN--!Up?f0R.D*wN
0313FE50  2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
0313FE60  21 55 70 3F 00 01 02 03 04 05 06 07 08 09 0A 0B !Up?.....

```

After the block is created with the replicated string, the values from 00 to FF are found and used for the decrypting process.

```

00 00 90 00 21 55 70 3F 66 30 52 2E 44 2A 77 4E ....!Up?f0R.D*wN
2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 !Up?f0R.D*wN--!U
70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F p?f0R.D*wN--!Up?
66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 f0R.D*wN--!Up?f0
52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E R.D*wN--!Up?f0R.
44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A D*wN--!Up?f0R.D*
77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E wN--!Up?f0R.D*wN
2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 !Up?f0R.D*wN--!U
70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F p?f0R.D*wN--!Up?
66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 f0R.D*wN--!Up?f0
52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E R.D*wN--!Up?f0R.
44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A D*wN--!Up?f0R.D*
77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E wN--!Up?f0R.D*wN
2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 !Up?.....
00 01 02 03 04 05 06 07 08 09 0A 0B .....
0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B .....
1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B ....!"#$%&'()*+
2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B ,-. /0123456789:;
3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B <=>?@ABCDEFGHIJK
4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B LMNOPQRSTUVWXYZ[
5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B \]^_`abcdefghijkl
6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B mnopqrstuvwxyz{
7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B |}~.€.,f,,m+#+^%Š<
8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B Œ.Ž..‘’“”•—~™ξ>
9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB æ.žÿ·;¢£¥¦§¨©ª«
AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB ¬°±²³´µ¶·¸¹º»
BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB %&ç;ÀÁÂÃÄÅÆÇÈÉÊË
CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB ÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛ
DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB ŸŸPßàáâãäåæçèéêë
EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB ìíîïðñòóôõö÷øùúû
FC FD FE FF 00 00 00 00 00 00 00 00 00 00 00 00 üýþÿ.....

```

At this point, the loader transforms the 00-FF block with a series of additions combining the replicated string's block with the 00-FF block, as we can see:



```
0000000000910060
0000000000910060 offset_calculate:      ; Value from chunk 00 - FF ; inital value edi = 0
0000000000910060 mov     dl, [ebp+esi+chunk_00_FF]
0000000000910067 movzx  eax, [ebp+esi+chunk_pass_repeated] ; Value from chunk string repeated
000000000091006F add     edi, eax      ; add string value + edi
0000000000910071 movzx  ecx, dl
0000000000910074 add     edi, ecx
0000000000910076 and     edi, 800000FFh
000000000091007C jns     short save_new_value_stack
```

```
000000000091007E dec     edi
000000000091007F or      edi, 0FFFFFF0h
0000000000910085 inc     edi
```

```
0000000000910086
0000000000910086 save_new_value_stack:
0000000000910086 mov     al, [ebp+edi+chunk_00_FF]
000000000091008D mov     [ebp+esi+chunk_00_FF], al ; overwrite chunk_00_FF
0000000000910094 inc     esi
0000000000910095 mov     [ebp+edi+chunk_00_FF], dl ; overwrite chunk_00_FF
000000000091009C cmp     esi, 100h
00000000009100A2 jl     short offset_calculate ; Value from chunk 00 - FF ; inital value edi = 0
```

The combination of the blue block (in following image) and the 00-FF block (pointed in red in previous image) results in the following block in memory, marked in red in the image:

```

0313FD60 00 00 90 00 21 55 70 3F 66 30 52 2E 44 2A 77 4E ...!Up?f0R.D*wN
0313FD70 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
0313FD80 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 !Up?f0R.D*wN--!U
0313FD90 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F p?f0R.D*wN--!Up?
0313FDA0 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 f0R.D*wN--!Up?f0
0313FDB0 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E R.D*wN--!Up?f0R.
0313FDC0 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A D*wN--!Up?f0R.D*
0313FDD0 77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E wN--!Up?f0R.D*wN
0313FDE0 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
0313FDF0 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 !Up?f0R.D*wN--!U
0313FE00 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F p?f0R.D*wN--!Up?
0313FE10 66 30 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 f0R.D*wN--!Up?f0
0313FE20 52 2E 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E R.D*wN--!Up?f0R.
0313FE30 44 2A 77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A D*wN--!Up?f0R.D*
0313FE40 77 4E 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E wN--!Up?f0R.D*wN
0313FE50 2D 2D 21 55 70 3F 66 30 52 2E 44 2A 77 4E 2D 2D --!Up?f0R.D*wN--
0313FE60 21 55 70 3F 97 89 E9 2B 37 CA 22 57 A3 A6 78 B4 !Up?~%é+7Ê"WE!x´
0313FE70 28 23 52 A1 B5 48 70 A8 18 08 44 9D A7 D2 C2 9E (#RjµHp".D.ŞÔÂŽ
0313FE80 47 0F DD F5 C9 10 F0 EE 3F 1F 8C 11 02 DE E4 AA G.ÝðÉ.ðî?.Ē..Pā≡
0313FE90 24 AD EC 71 9F 3A 74 60 7C DF E0 FD F9 87 12 9C $iqÿ:t`|Bàýù‡.æ
0313FEA0 5B 8E 1C 9A B6 4C 0C 7A BF B9 96 D6 C3 49 14 0B [Ž.šŋL.zç¹-ÖÄI..
0313FEB0 35 54 01 B2 42 C5 F1 D4 0D 98 BE 0A 1A F6 68 91 5T.²BĀñÔ.~%..öh´
0313FEC0 3C 16 92 45 3D 38 94 D9 03 C4 04 D5 4E 6A 85 AC <.'E=8"Û.Ä.ÖNj...~
0313FED0 E1 D0 E5 81 CF D8 BA 17 51 75 6B CD 7E 2A F7 C0 áðå.İø°.QukÍ~÷À
0313FEE0 4F E7 4D 63 CC 4A 25 D1 99 8F 6F B3 2D 77 B8 56 OçMcÌJ%Ń™.o³-w.V
0313FEF0 33 06 59 B0 D7 55 C1 8A A5 A4 07 0E 2E 27 8B 7D 3.Y°xUÁŠÿ¤...´<}
0313FF00 AB 69 00 CE DB 5A 1D EB 90 3B 93 6D FA E8 F2 5C «i.ÎÛZ.ë.;“múèò\
0313FF10 29 76 79 FC C7 50 5F 88 6C ED DC 2C 1E 2F 7F A0 )vyüÇP_`líÛ,./..
0313FF20 26 C8 E2 15 30 A9 EF 5E 19 A2 39 32 6E 64 E6 FE &Èâ.00î^..ç92ndæp
0313FF30 40 05 80 CB FB B1 7B 53 65 43 09 73 31 DA 34 95 @.€ËÛ±{SeC.s1Û4*
0313FF40 F3 3E 62 FF E3 66 46 82 8D F4 D3 13 86 F8 BD EA ó>byäff, .óó.†ø%ê
0313FF50 84 36 1B 83 20 21 C6 4B AF AE BC B7 58 5D 41 61 „6.f·!ÆK~ø%·X]Aa
0313FF60 67 BB 9B 72 00 00 00 00 00 00 00 00 00 00 00 00 g»>r.....

```

On the next step, the loader reads the initial argument, arg0, whose value is 0x900000 and points at the 4Kb block, which stores the absolute addresses to different API from Win32:

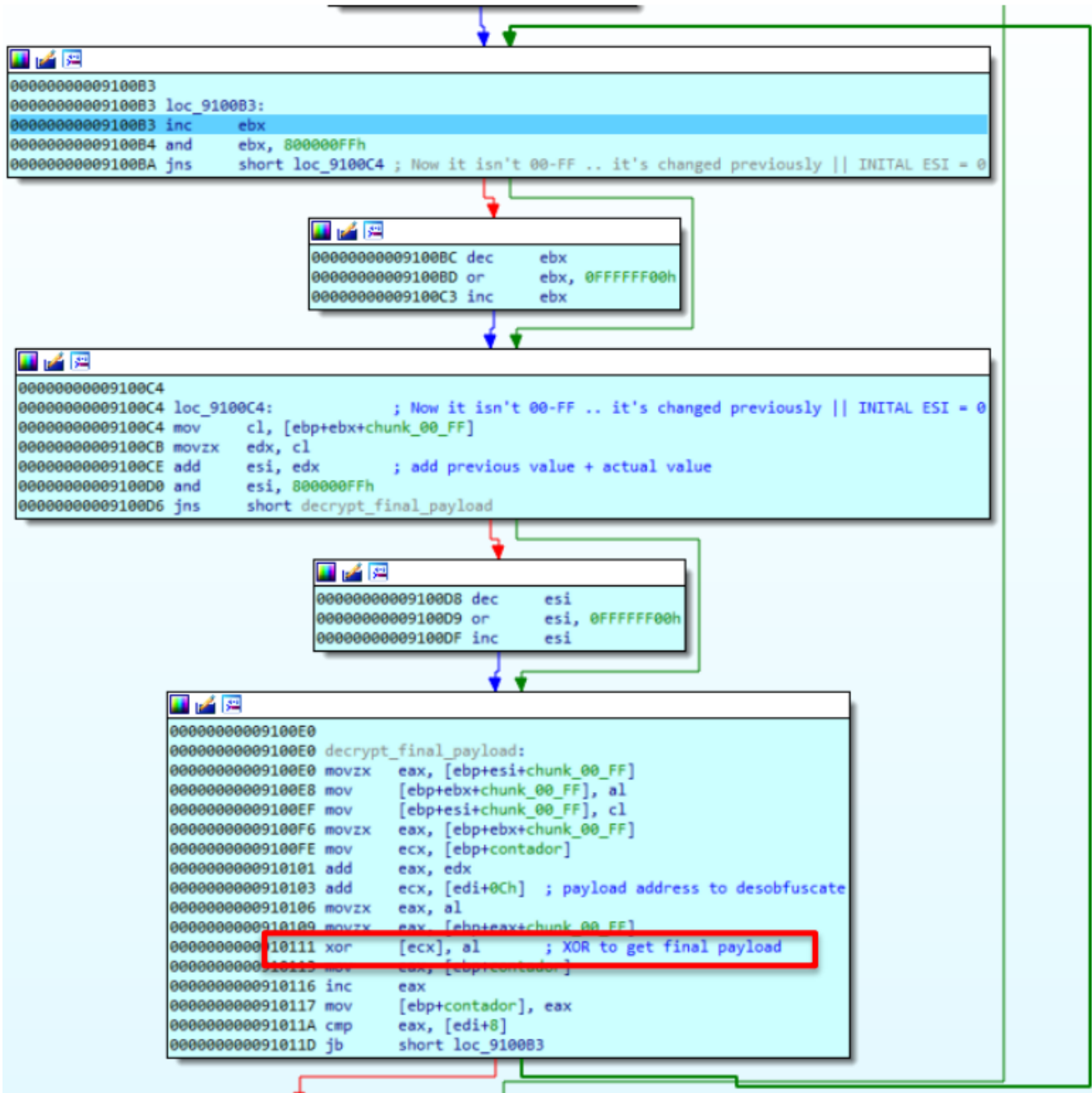
```

00000000009100A4 mov edi, [ebp+arg_0] ; Arg0 value = 0x900000
00000000009100A7 xor esi, esi
00000000009100A9 mov [ebp+var_4], esi
00000000009100AC cmp [edi+8], esi
00000000009100AF jbe short loc_910122

```

After this, the decrypting process for the final payload begins. The decrypting process gets two values from the second block, exchanges and adds them, and the result serves as a final index to recover the element from the second block with which the xor will be achieved through the coded block.

This description of the decryption algorithm has been identified as the **RC4 algorithm**.

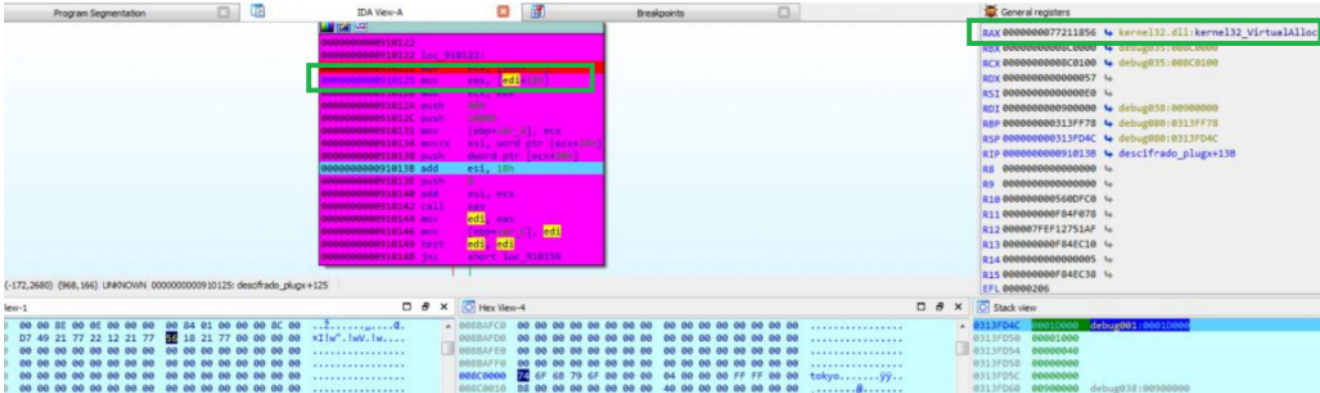


After the decryption process, we find a PE binary, as seen in the following image. In this case, the payload does not start with the traditional MZ header but the string "tokyo":

```
netsh.exe (3640) (0x8c0000 - 0x8d9000)
00000000 74 6f 6b 79 6f 00 00 00 04 00 00 00 ff ff 00 00 tokyo.....
00000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00
00000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 00 00
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000070 00 00 00 00 2e 0d 0d 0a 24 00 00 00 00 00 00 00
00000080 30 19 ba 86 74 78 d4 d5 74 78 d4 d5 74 78 d4 d5 0...tx..tx..tx..
00000090 60 13 d7 d4 7e 78 d4 d5 60 13 d1 d4 f8 78 d4 d5 `...~x..`...x..
000000a0 60 13 d0 d4 66 78 d4 d5 26 0d d0 d4 65 78 d4 d5 `...fx..&...ex..
000000b0 26 0d d7 d4 65 78 d4 d5 26 0d d1 d4 5e 78 d4 d5 &...ex..&...^x..
000000c0 60 13 d5 d4 7f 78 d4 d5 74 78 d5 d5 0b 78 d4 d5 `...x..tx...x..
000000d0 c1 0d dd d4 72 78 d4 d5 c1 0d 2b d5 75 78 d4 d5 ....rx....+ux..
000000e0 c1 0d d6 d4 75 78 d4 d5 52 69 63 68 74 78 d4 d5 ....ux..Richtx..
000000f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000100 50 45 00 00 4c 01 05 00 17 29 b0 61 00 00 00 00 PE..L....).a....
00000110 00 00 00 00 e0 00 02 01 0b 01 0e 1d 00 fc 00 00
00000120 00 90 00 00 00 00 00 00 b1 59 00 00 00 10 00 00
00000130 00 10 01 00 00 00 40 00 00 10 00 00 00 02 00 00
00000140 06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00
00000150 00 d0 01 00 00 04 00 00 00 00 00 00 03 00 40 81
00000160 00 00 10 00 00 10 00 00 00 10 00 00 10 00 00 00
00000170 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00
00000180 50 6b 01 00 78 00 00 00 a0 01 00 e0 01 00 00 Pk..x.....
00000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001a0 00 b0 01 00 f0 10 00 00 e0 5e 01 00 38 00 00 00
000001b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001c0 00 00 00 00 00 00 00 00 18 5f 01 00 40 00 00 00
000001d0 00 00 00 00 00 00 00 00 10 01 00 cc 01 00 00
000001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001f0 00 00 00 00 00 00 00 00 2e 74 65 78 74 00 00
00000200 08 fa 00 00 00 10 00 00 00 fc 00 00 00 04 00
00000210 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00
00000220 2e 72 64 61 74 61 00 00 86 65 00 00 00 10 01
00000230 00 66 00 00 00 00 01 00 00 00 00 00 00 00 00
00000240 00 00 00 00 40 00 00 40 2e 64 61 74 61 00 00
00000250 10 14 00 00 00 80 01 00 00 0a 00 00 00 66 01
00000260 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00
00000270 2e 72 73 72 63 00 00 00 e0 01 00 00 00 a0 01
00000280 00 02 00 00 00 70 01 00 00 00 00 00 00 00 00
00000290 00 00 00 00 40 00 00 40 2e 72 65 6c 6f 63 00
000002a0 f0 10 00 00 00 b0 01 00 00 12 00 00 00 72 01
000002b0 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00
000002c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Then, we see how it loads the VirtualAlloc absolute address (0x77211856) from the segment previously created:



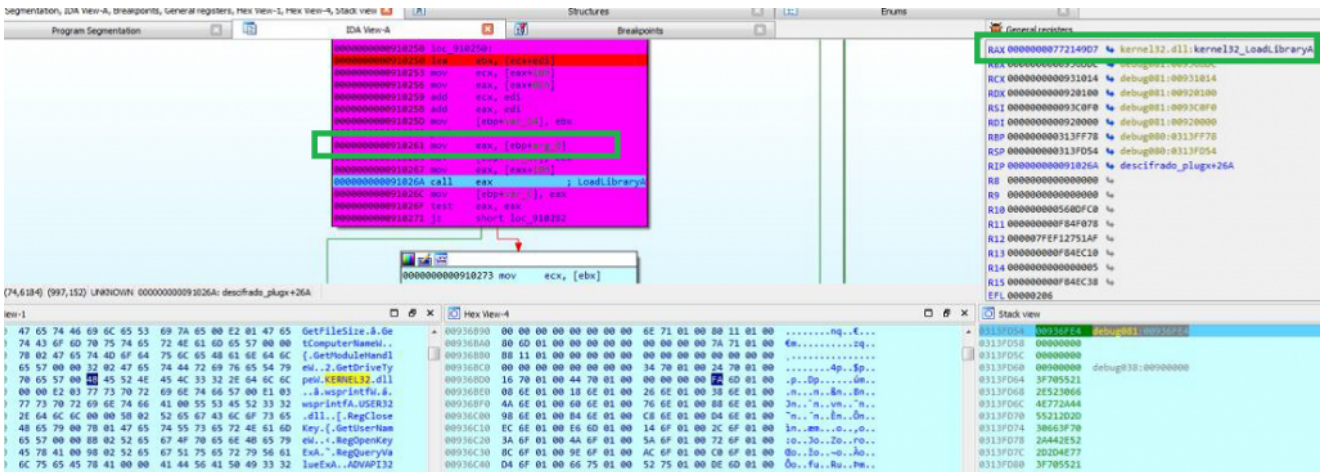


This creates another memory segment in the process netsh.exe with RWX licenses (that of 116Kb) which will be used to load the PE:

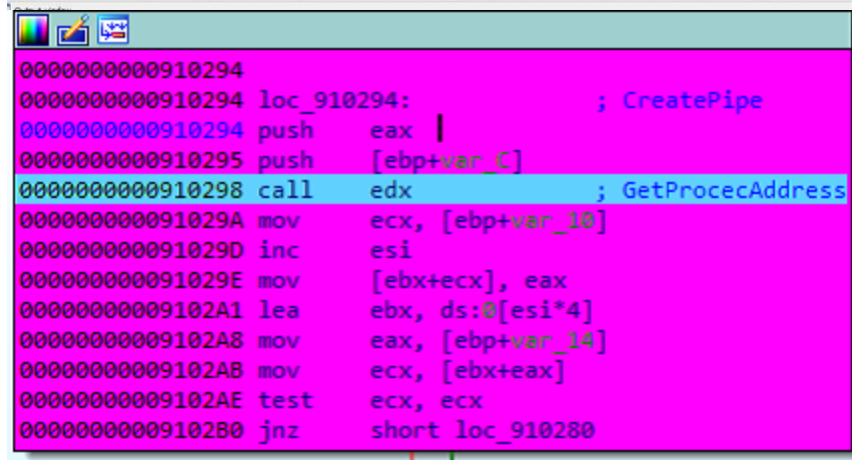
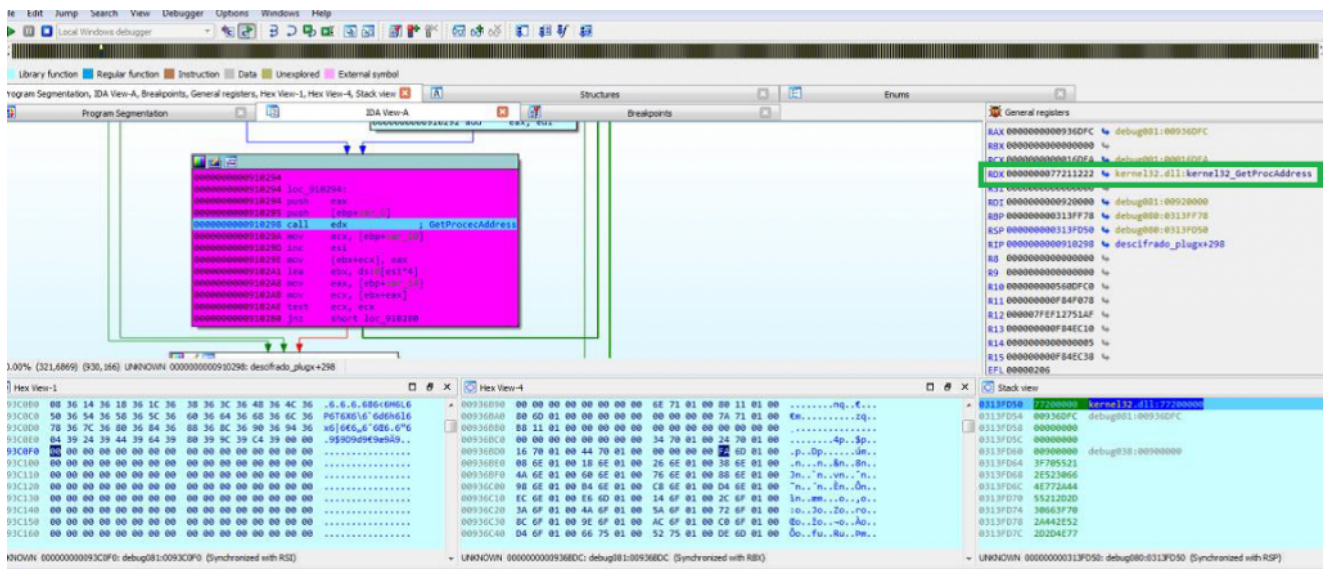
0x8c0000	Private: Commit	100 kB	RWX	100 kB	100 kB
0x8e0000	Private: Commit	4 kB	RWX	4 kB	4 kB
0x900000	Private: Commit	4 kB	RWX	4 kB	4 kB
0x910000	Private: Commit	4 kB	RWX	4 kB	4 kB
0x920000	Private: Commit	116 kB	RWX	4 kB	4 kB

In this new segment, it maps the binary using the virtual addresses as the regular Windows PE loader would do.

Then, it calls the API LoadLibraryA (it has the address since the DLL saved it in the memory segment) of the strings located in the mapped block:



Then it calls GetProcAddress() to get the addresses of certain functions:



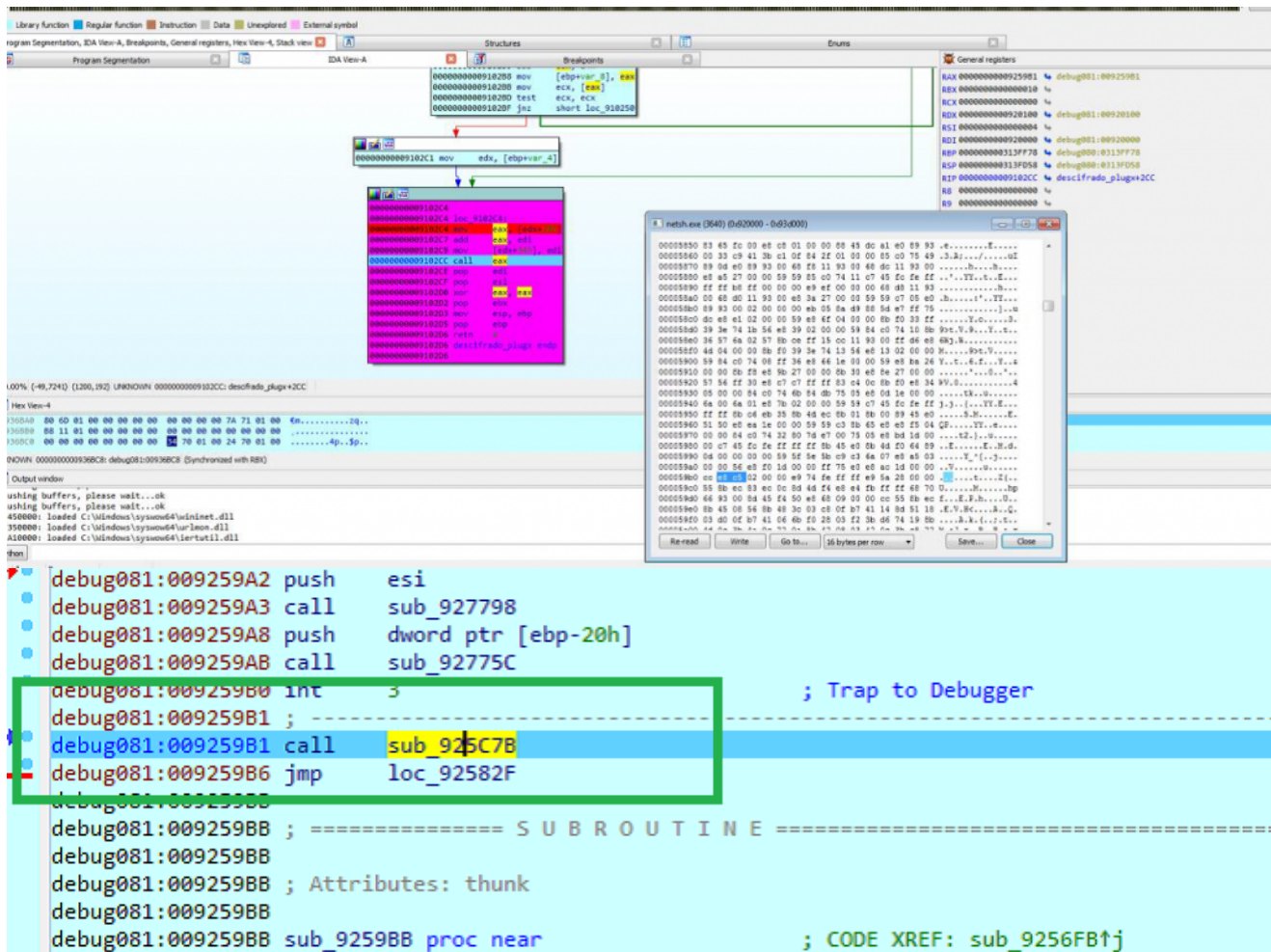
Next, the libraries and functions block may be appreciated:

## Hex View-1

00936E30	75	70	49	6E	66	6F	41	00	E3	00	43	72	65	61	74	65	upInfoA.ä.Create
00936E40	50	72	6F	63	65	73	73	41	00	00	F3	03	4D	75	6C	74	ProcessA..ó.Mult
00936E50	69	42	79	74	65	54	6F	57	69	64	65	43	68	61	72	00	iByteToWideChar.
00936E60	02	06	57	69	64	65	43	68	61	72	54	6F	4D	75	6C	74	..WideCharToMult
00936E70	69	42	79	74	65	00	83	01	46	69	6E	64	46	69	72	73	iByte.f.FindFirs
00936E80	74	46	69	6C	65	57	00	00	8F	01	46	69	6E	64	4E	65	tFileW...FindNe
00936E90	78	74	46	69	6C	65	57	00	65	01	45	78	70	61	6E	64	xtFileW.e.Expand
00936EA0	45	6E	76	69	72	6F	6E	6D	65	6E	74	53	74	72	69	6E	EnvironmentStrin
00936EB0	67	73	57	00	8B	04	52	65	6D	6F	76	65	44	69	72	65	gsW.»RemoveDire
00936EC0	63	74	6F	72	79	57	00	00	78	01	46	69	6E	64	43	6C	ctoryW..x.FindCl
00936ED0	6F	73	65	00	1F	03	47	65	74	56	6F	6C	75	6D	65	49	ose...GetVolumeI
00936EE0	6E	66	6F	72	6D	63	74	69	6F	6E	41	00	CE	00	43	72	nformationA.î.Cr
00936EF0	65	61	74	65	46	69	6C	65	57	00	6A	02	47	65	74	4C	reateFileW.j.GetL
00936F00	6F	67	69	63	61	6C	44	72	69	76	65	53	74	72	69	6E	ogicalDriveStrin
00936F10	67	73	57	00	6D	01	46	69	6C	65	54	69	6D	65	54	6F	gsW.m.FileTimeTo
00936F20	53	79	73	74	65	6D	54	69	6D	65	00	00	18	01	44	65	SystemTime...De
00936F30	6C	65	74	65	46	69	6C	65	57	00	E6	02	47	65	74	53	leteFileW.æ.GetS
00936F40	79	73	74	65	6D	49	6E	66	6F	00	F6	00	43	72	65	61	ystemInfo.ö.Crea
00936F50	74	65	54	68	72	65	61	64	00	00	28	03	47	65	74	57	teThread..(.GetW
00936F60	69	6E	64	6F	77	73	44	69	72	65	63	74	6F	72	79	41	indowsDirectoryA
00936F70	00	00	6C	01	46	69	6C	65	54	69	6D	65	54	6F	4C	6F	..l.FileTimeToLo
00936F80	63	61	6C	46	69	6C	65	54	69	6D	65	00	B1	02	47	65	calFileTime.±.Ge
00936F90	74	50	72	6F	63	41	64	64	72	65	73	73	00	00	4E	02	tProcAddress..N.
00936FA0	47	65	74	46	69	6C	65	53	69	7A	65	00	E2	01	47	65	GetFileSize.â.Ge
00936FB0	74	43	6F	6D	70	75	74	65	72	4E	61	6D	65	57	00	00	tComputerNameW..
00936FC0	78	02	47	65	74	4D	6F	64	75	6C	65	48	61	6E	64	6C	{.GetModuleHandl
00936FD0	65	57	00	00	32	02	47	65	74	44	72	69	76	65	54	79	eW..2.GetDriveTy
00936FE0	70	65	57	00	48	45	52	4E	45	4C	33	32	2E	64	6C	6C	peW.KERNEL32.dll
00936FF0	00	00	E2	03	77	73	70	72	69	6E	74	66	57	00	E1	03	..â.wsprintfW.á.
00937000	77	73	70	72	69	6E	74	66	41	00	55	53	45	52	33	32	wsprintfA.USER32
00937010	2E	64	6C	6C	00	00	5B	02	52	65	67	43	6C	6F	73	65	.dll..[.RegClose
00937020	48	65	79	00	7B	01	47	65	74	55	73	65	72	4E	61	6D	Key.{.GetUserNam
00937030	65	57	00	00	8B	02	52	65	67	4F	70	65	6E	4B	65	79	eW..<.RegOpenKey
00937040	45	78	41	00	98	02	52	65	67	51	75	65	72	79	56	61	ExA.~.RegQueryVa
00937050	6C	75	65	45	78	41	00	00	41	44	56	41	50	49	33	32	lueExA..ADVAPI32
00937060	2E	64	6C	6C	00	00	79	00	48	74	74	70	4F	70	65	6E	.dll..y.HttpOpen
00937070	52	65	71	75	65	73	74	57	00	00	CC	00	49	6E	74	65	RequestW..î.Inte
00937080	72	6E	65	74	51	75	65	72	79	4F	70	74	69	6F	6E	41	rnetQueryOptionA
00937090	00	00	EF	00	49	6E	74	65	72	6E	65	74	57	72	69	74	..i.InternetWrit
009370A0	65	46	69	6C	65	00	C9	00	49	6E	74	65	72	6E	65	74	eFile.É.Internet
009370B0	4F	70	65	6E	57	00	DC	00	49	6E	74	65	72	6E	65	74	OpenW.Û.Internet
009370C0	53	65	74	4F	70	74	69	6F	6E	41	00	00	7E	00	48	74	SetOptionA.~.Ht
009370D0	74	70	51	75	65	72	79	49	6E	66	6F	57	00	00	72	00	tpQueryInfoW..r.
009370E0	48	74	74	70	45	6E	64	52	65	71	75	65	73	74	57	00	HttpEndRequestW.
009370F0	80	00	48	74	74	70	53	65	6E	64	52	65	71	75	65	73	€.HttpSendReques
00937100	74	45	78	41	00	00	82	00	48	74	74	70	53	65	6E	64	tExA.,.HttpSend
00937110	52	65	71	75	65	73	74	57	00	00	95	00	49	6E	74	65	RequestW..*.Inte
00937120	72	6E	65	74	43	6C	6F	73	65	48	61	6E	64	6C	65	00	rnetCloseHandle.
00937130	9C	00	49	6E	74	65	72	6E	65	74	43	6F	6E	6E	65	63	æ.InternetConnec
00937140	74	57	00	00	CE	00	49	6E	74	65	72	6E	65	74	52	65	tW..î.InternetRe
00937150	61	64	46	69	6C	65	00	00	DF	00	49	6E	74	65	72	6E	adFile..ß.Intern
00937160	65	74	53	65	74	4F	70	74	69	6F	6E	57	00	00	57	49	etSetOptionW..WI
00937170	4E	49	4E	45	54	2E	64	6C	6C	00	57	53	32	5F	33	32	NINET.dll.WS2_32
00937180	2E	64	6C	6C	00	00	4F	04	51	75	65	72	79	50	65	72	.dll..O.QueryPer
00937190	66	6F	72	6D	61	6E	63	65	43	6F	75	6E	74	65	72	00	formanceCounter.
009371A0	1B	02	47	65	74	43	75	72	72	65	6E	74	50	72	6F	63	..GetCurrentProc

After the correct mapping and having loaded the necessary libraries for its proper functioning, it calls EAX to run the decrypted and mapped payload:





To summarize, this article goes through the process followed in memory after executing the Creative Cloud application until deploying TokyoX in memory. This DLL sideloading style is often linked to APT groups whose attribution is also linked to China, however being a known technique as it is, we are not able to consider any feasible attribution at the moment.

As reviewed at the beginning of the article, what we have named as “TokyoX” has not been identified as a known malware so far (at least, with the sources that we have).

Additionally, at some point of the analysis we identified a tool used by this group for the creation of version.dll, which pretends to be a Windows DLL located in SysWOW/System32. The string “AheadLib” found among the code of the malicious version.dll drew our attention, and we quickly found two chinese (casually or not) GitHub repositories with the source code of some tool called AheadLib.



<https://github.com/strivexjun/AheadLib-x86-x64>

## GitHub - strivexjun/AheadLib-x86-x64: hijack dll Source ...

AheadLib-x86-x64 hijack dll Source Code Generator. support x86/x64 snapshot screen. 不支持导出符号带有??的方法! NOTE. Pay attention to the generated file header prompt information

[Actions](#) · [Releases 1](#) · [Notifications](#) · [Issues](#)

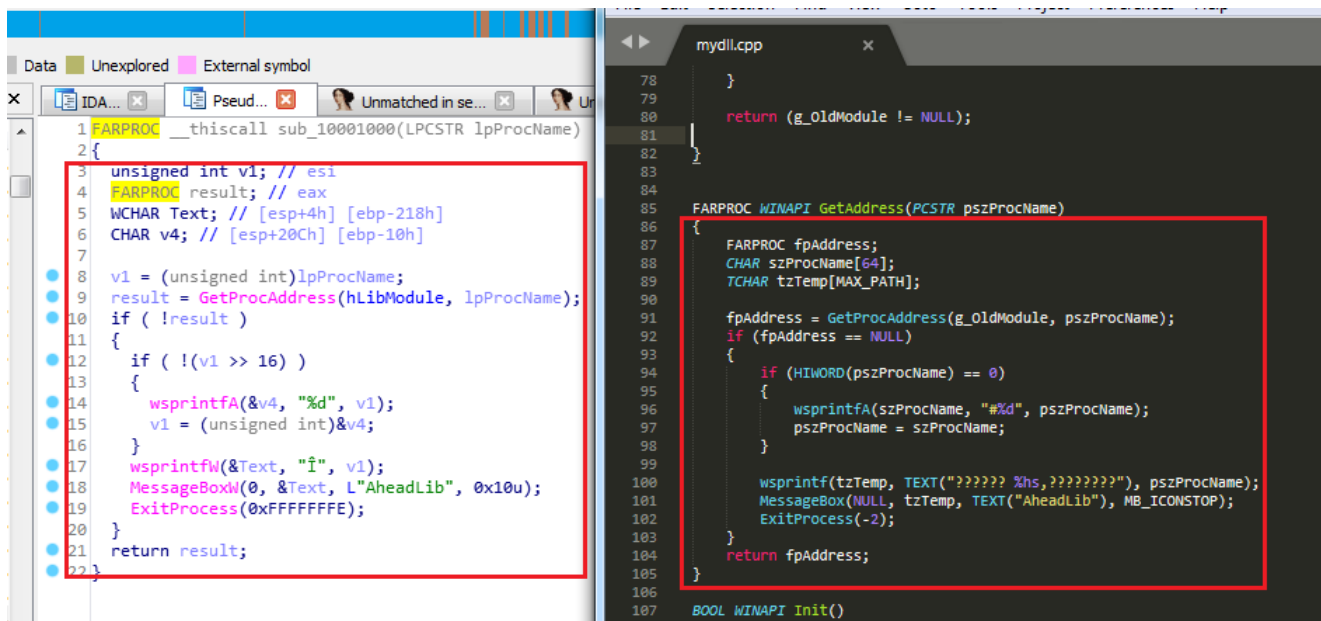
<https://github.com/Yonism/AheadLib>

## Yonism/AheadLib: Fake DLL Source Code Generator - GitHub

AheadLib. Fake DLL Source Code Generator. AheadLib 2.2.150 - 自动生成一个特洛伊 DLL 分析代码的工具 ...



Basically, this tool will allow you to create a C++ source code file, implementing a DLL with the same exported functions as a given DLL. For the purpose of the current analysis we generated a source code file using this tool and giving the legitimate version.dll as input.



In the shown screenshot we can see on the left side the pseudocode generated by IDA Pro while analyzing the malicious version.dll sample. On the right side, we can observe the source code automatically generated by AheadLib using the legitimate version.dll as input. Even though the exported functions are not shown in the previous image, we can appreciate how there is a perfect match between both snippets.

Read the second part of the analysis of the final “TokyoX” RAT and its capacities [here](#).

## IOCs

---

- 382b3d3bb1be4f14dbc1e82a34946a52795288867ed86c6c43e4f981729be4fc
- 31.192.107[.]187:443

Customers with Lab52's APT intelligence private feed service already have more tools and means of detection for this campaign.

In case of having threat hunting service or being client of S2Grupo CERT, this intelligence has already been applied.

If you need more information about Lab52's private APT intelligence feed service, you can contact us through the [following link](#)