

TokyoX: DLL side-loading an unknown artifact

Published: 2022-01-10 · Archived: 2026-04-05 23:04:25 UTC

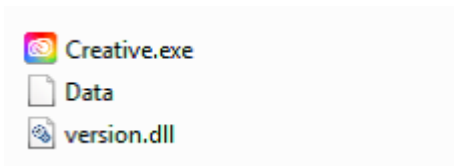
During Christmas holidays, Lab52 has been analyzing a sample which loads an artifact that we have decided to refer to as “TokyoX” since no similarities have been found as to any known malware, which we usually detect in open sources. However, we cannot confirm so far that it is indeed a new family of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

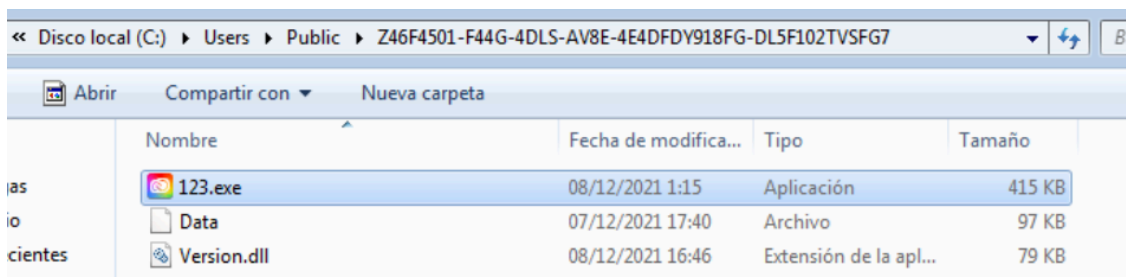
Some antivirus engines tagged the sample as PlugX, but it seems that the attribution might be due to the final payload’s loading mechanism: DLL sideloading with an encrypted payload in the same directory. After analyzing the final payload we could not find any similarities with other known samples from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

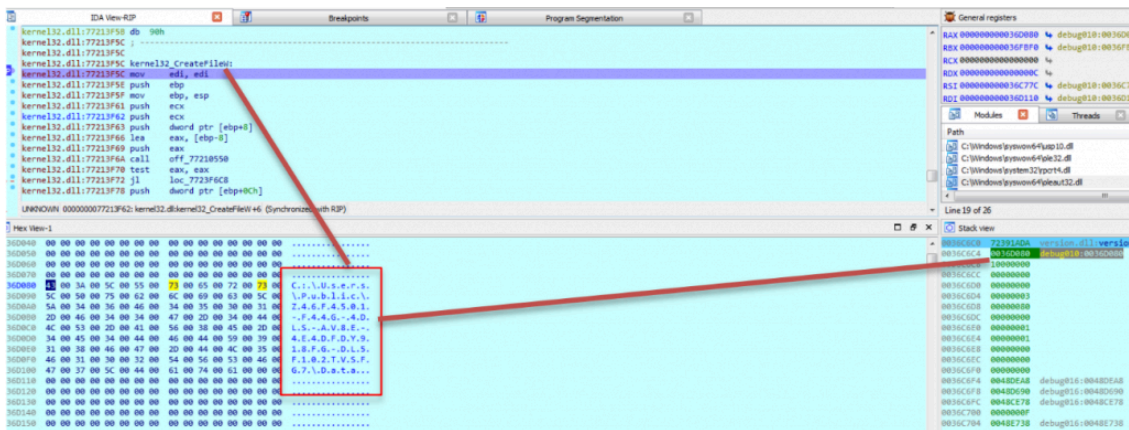


Once executed, we observe how the netsh.exe process tries to establish connections with port 443 of the IP address 31.192.107[.]187.

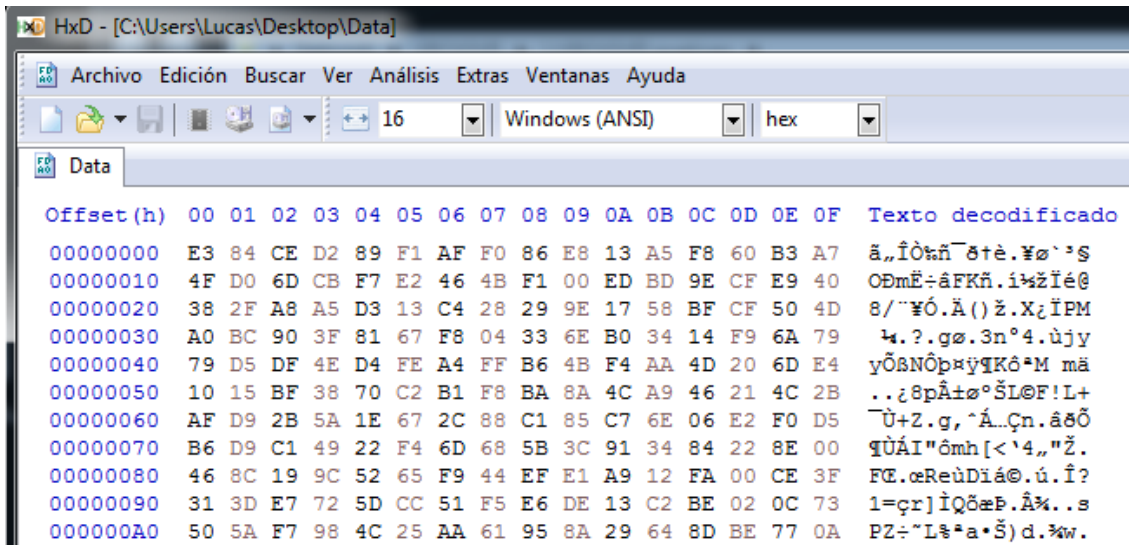
In this analysis we will focus on different aspects about the process; from double-clicking the binary 123.exe process (which is a copy of Creative.exe but in another path) to the execution of “TokyoX” already decrypted in memory.

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:

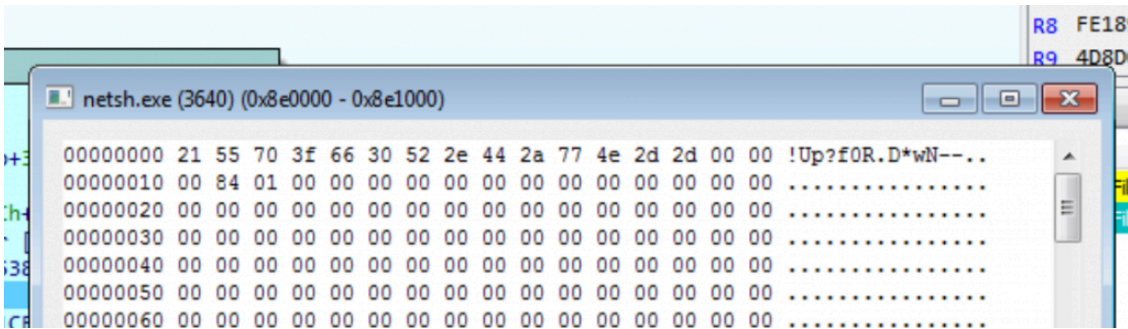


If we edit the Data file with a hexadecimal editor we will see their values, which will help us to identify it in memory later (beginning with E3 84):

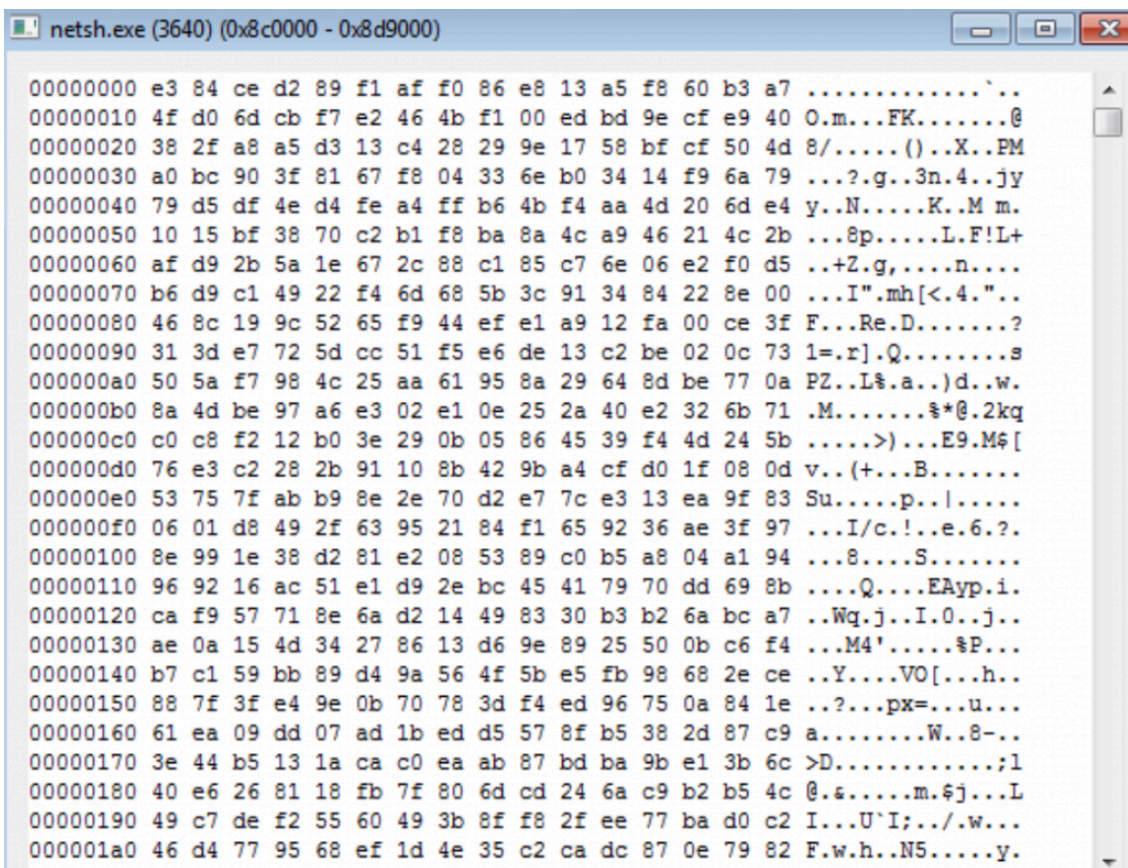


After reading the file from disk, a child process netsh.exe is created. This just-created child process is where several new memory segments will be located (a total of 5, including the final decrypted payload) to decrypt the final “TokyoX” payload. The APIs which were observed for the creation and writing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

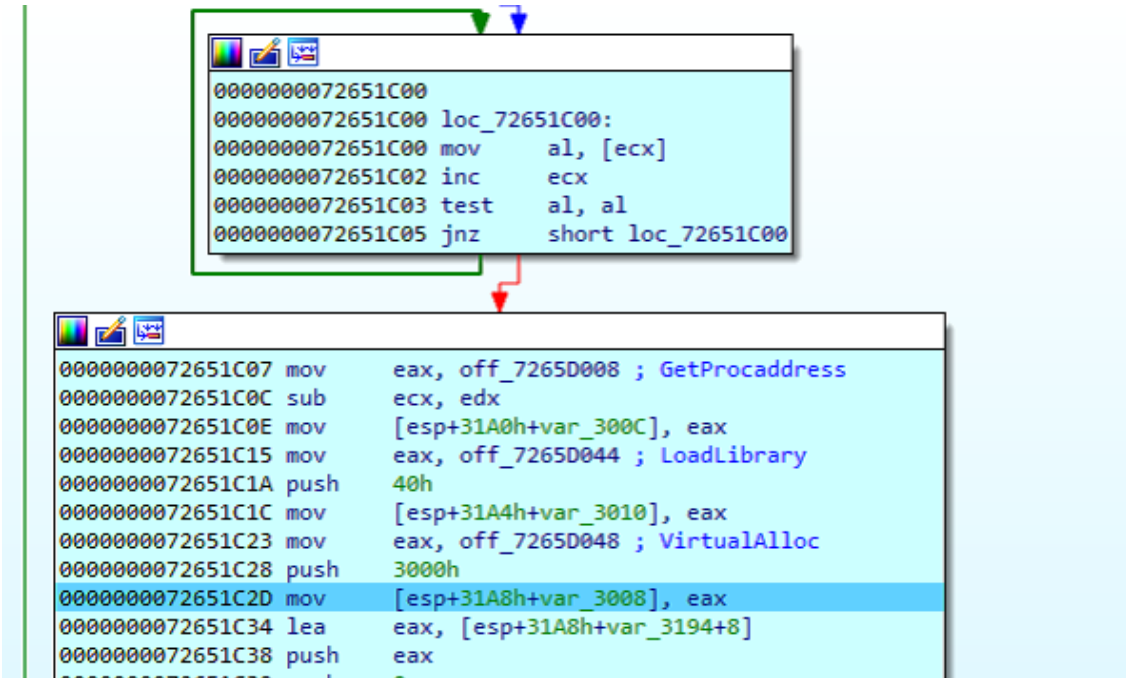
First, it creates two segments: 100Kb where the encrypted payload is located and which comes from the disc, and another one of 4Kb. In the 4Kb segment we observe how the following string is set (which will be the string used for the decrypting process):



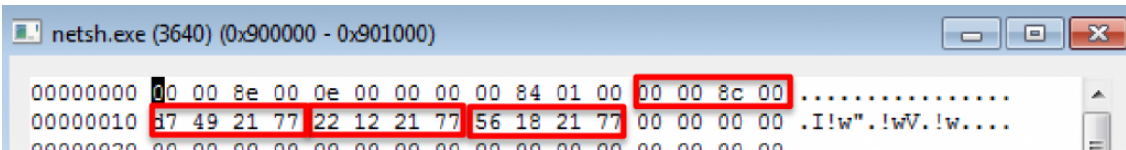
The other memory segment of 100Kb contains the following (encrypted content, as we see how it matches the content from Data file on Disk):



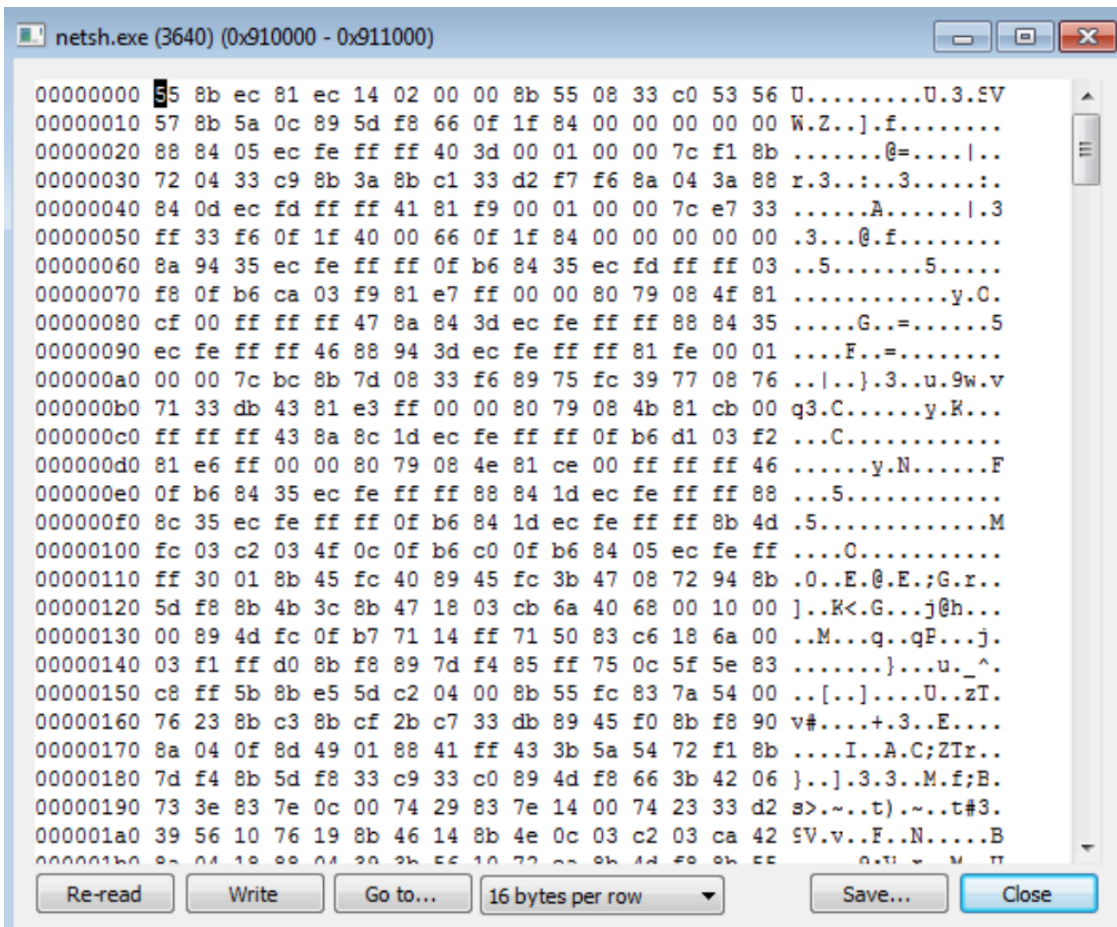
After the creation of these two segments, a third segment is allocated, where it is loaded the absolute memory addresses from several win32 APIs (VirtualAlloc, LoadLibrary, GetProcAddress, the home address of the coded payload, etc.) for its later use by the loader:



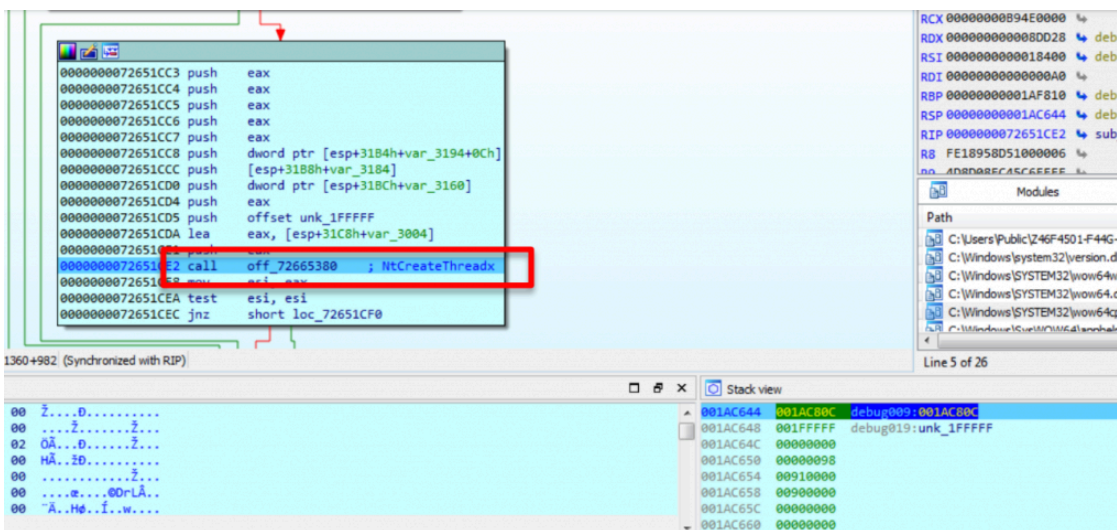
We can notice how the segment will have the memory addresses (starting from 123.exe they are located in netsh.exe segment through the version.dll code):



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



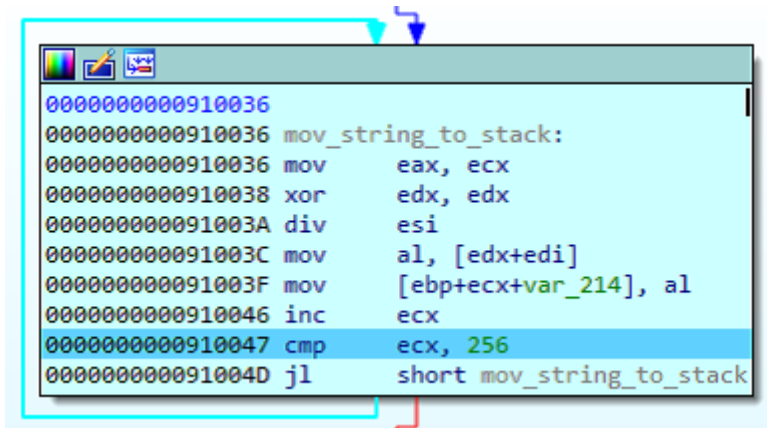
Finally, the “TokyoX” loader runs from the DLL (version.dll) in netsh.exe through the API NtcreateThreadEx and we see the start of the last page created in the stack:



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 88 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.



```
000000000910036
000000000910036 mov_string_to_stack:
000000000910036 mov     eax, ecx
000000000910038 xor     edx, edx
00000000091003A div     esi
00000000091003C mov     al, [edx+edi]
00000000091003F mov     [ebp+ecx+var_214], al
000000000910046 inc     ecx
000000000910047 cmp     ecx, 256
00000000091004D 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?.....
0313FE70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

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 00 01 02 03 04 05 06 07 08 09 0A 0B !Up?.....
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 lmnopqrstuvwxyz{
7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B |}~.€.,f,,m+?^%$<
8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B E.Ž... ' ) ( ) ) . _ ~ ^ % $ >
9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB æ.žÿ·;ç£H¥!§~@³«
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 %%;ÁÁÁÁÁÁÆÇÈÉÊË
CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB ÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚ
DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB ÜÝÞàáâãäåæçèéêë
EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB ìíîïðñóôõö÷øùúû
FC FD FE FF 00 00 00 00 00 00 00 00 00 00 00 00 üýþÿ.....
```

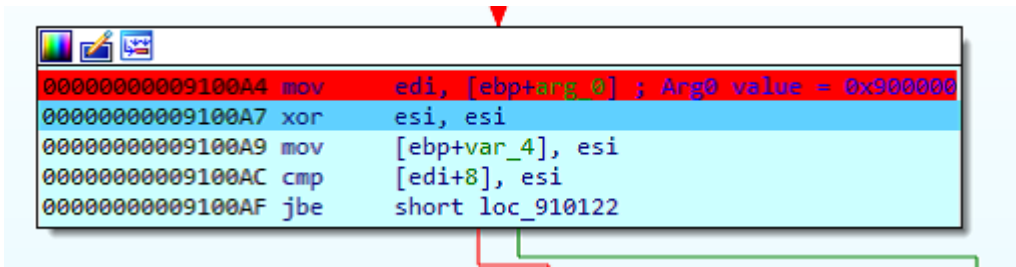
At this point, the loader transforms the 00-FF block with a series of additions combining the replicated string's block with the 00-FF block, as we can see:



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:



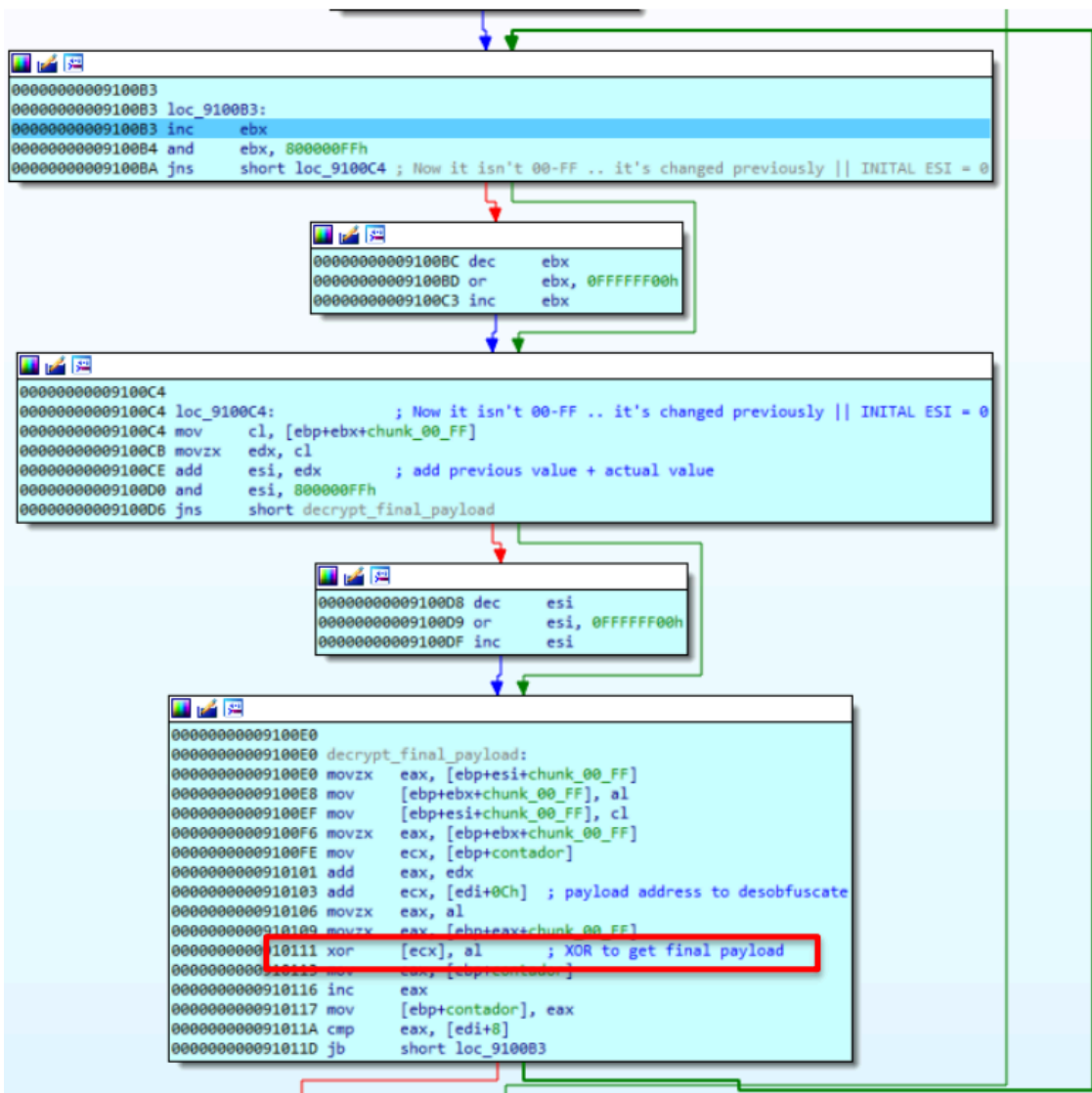
On the next step, the loader reads the initial argument, arg0, whose value is 0x900000 and points at the 4Kb block, which stores the absolute addresses to different API from Win32:



```
0000000009100A4 mov     edi, [ebp+arg_0] ; Arg0 value = 0x900000
0000000009100A7 xor     esi, esi
0000000009100A9 mov     [ebp+var_4], esi
0000000009100AC cmp     [edi+8], esi
0000000009100AF 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**.



```
0000000009100B3 loc_9100B3:
0000000009100B3 inc     ebx
0000000009100B4 and     ebx, 800000FFh
0000000009100BA jns     short loc_9100C4 ; Now it isn't 00-FF .. it's changed previously || INITIAL ESI = 0

0000000009100BC dec     ebx
0000000009100BD or      ebx, 0FFFFFF0h
0000000009100C3 inc     ebx

0000000009100C4 loc_9100C4:
0000000009100C4 mov     cl, [ebp+ebx+chunk_00_FF] ; Now it isn't 00-FF .. it's changed previously || INITIAL ESI = 0
0000000009100CB movzx   edx, cl
0000000009100CE add     esi, edx ; add previous value + actual value
0000000009100D0 and     esi, 800000FFh
0000000009100D6 jns     short decrypt_final_payload

0000000009100D8 dec     esi
0000000009100D9 or      esi, 0FFFFFF0h
0000000009100DF inc     esi

0000000009100E0 decrypt_final_payload:
0000000009100E0 movzx   eax, [ebp+esi+chunk_00_FF]
0000000009100E8 mov     [ebp+ebx+chunk_00_FF], al
0000000009100EF mov     [ebp+esi+chunk_00_FF], cl
0000000009100F6 movzx   eax, [ebp+ebx+chunk_00_FF]
0000000009100FE mov     ecx, [ebp+contador]
000000000910101 add     eax, edx
000000000910103 add     ecx, [edi+0Ch] ; payload address to desobfuscate
000000000910106 movzx   eax, al
000000000910109 movzx   eax, [ebp+contador]
000000000910111 xor     [ecx], al ; XOR to get final payload
000000000910112 mov     ecx, [ebp+contador]
000000000910116 inc     eax
000000000910117 mov     [ebp+contador], eax
00000000091011A cmp     eax, [edi+8]
00000000091011D jnb     short loc_9100B3
```

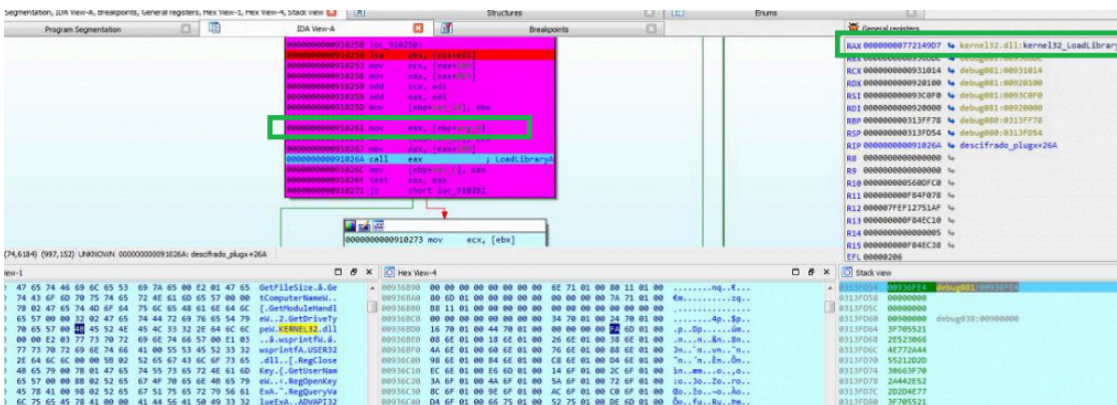
After the decryption process, we find a PE binary, as seen in the following image. In this case, the payload does not start with the traditional MZ header but the string “tokyo”:

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

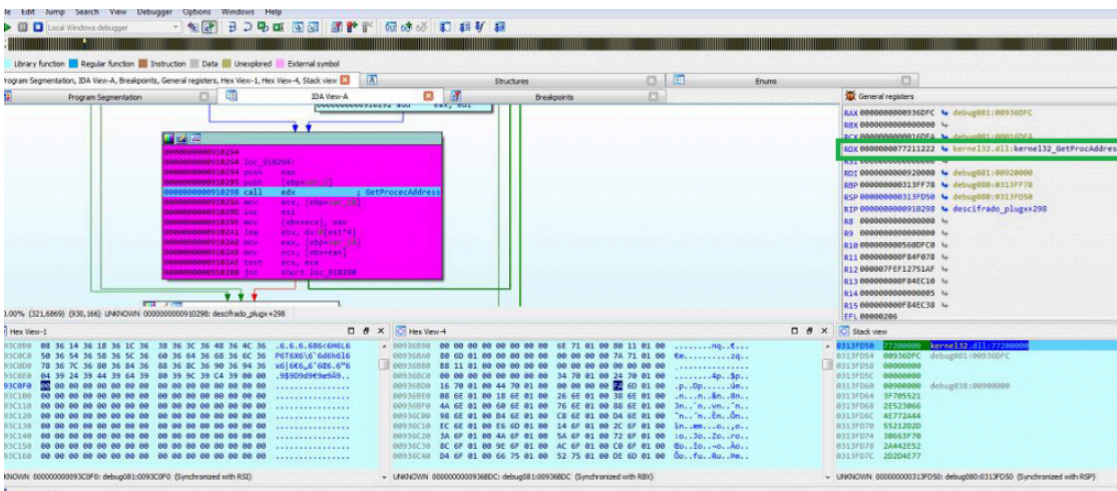
0x8c0000	Private: Commit	100 kb	RWX	100 kb	100 kb
0x8e0000	Private: Commit	4 kb	RWX	4 kb	4 kb
0x900000	Private: Commit	4 kb	RWX	4 kb	4 kb
0x910000	Private: Commit	4 kb	RWX	4 kb	4 kb
0x920000	Private: Commit	116 kb	RWX	4 kb	4 kb

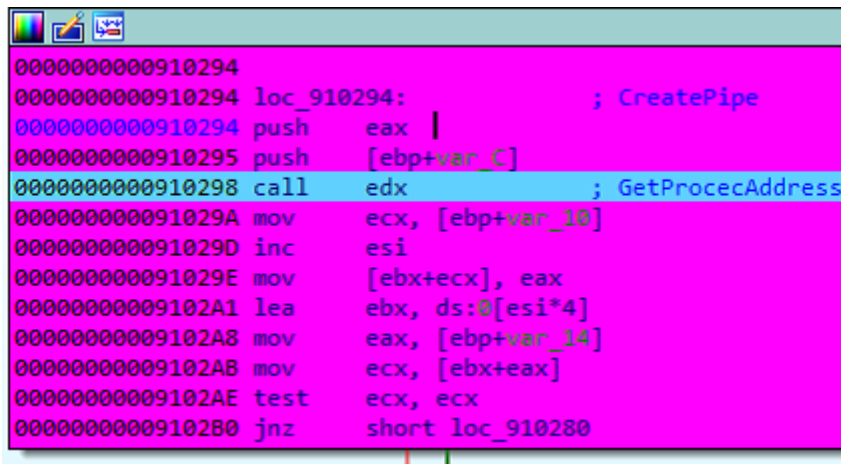
In this new segment, it maps the binary using the virtual addresses as the regular Windows PE loader would do.

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:



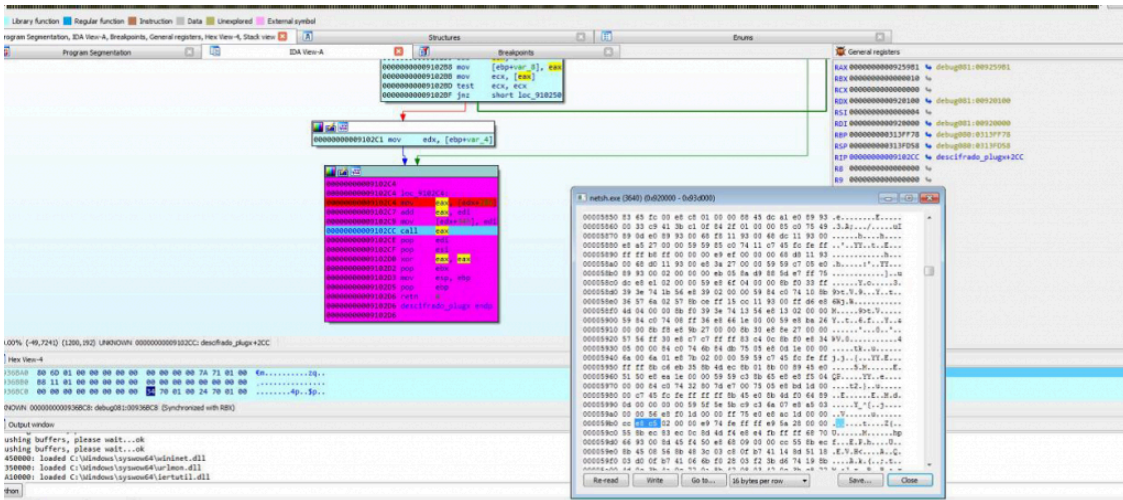


```
0000000000910294
0000000000910294 loc_910294:                ; CreatePipe
0000000000910294 push    eax |
0000000000910295 push    [ebp+var_C]
0000000000910298 call   edx                ; GetProceAddress
000000000091029A mov    ecx, [ebp+var_10]
000000000091029D inc    esi
000000000091029E mov    [ebx+ecx], eax
00000000009102A1 lea   ebx, ds:0[esi*4]
00000000009102A8 mov    eax, [ebp+var_14]
00000000009102AB mov    ecx, [ebx+eax]
00000000009102AE test   ecx, ecx
00000000009102B0 jnz   short loc_910280
```

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.ã.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	61	74	69	6F	6E	41	00	CE	00	43	72	nformationA.Ï.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.t.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.ã.Ge
00936FB0	74	43	6F	6D	70	75	74	65	72	4E	61	6D	65	57	00	00	tComputerNameW..
00936FC0	7B	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	4B	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	..ã.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	4B	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	48	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..Ï.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	..Ï.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..Ï.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.WS_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:



```

debug081:009259A2 push  esi
debug081:009259A3 call  sub_927798
debug081:009259A8 push  dword ptr [ebp-20h]
debug081:009259AB call  sub_92775C
debug081:009259B0 int   3 ; Trap to Debugger
debug081:009259B1 ;
debug081:009259B1 call  sub_925C7B
debug081:009259B6 jmp   loc_92582F
debug081:009259B8
debug081:009259BB ; ===== SUBROUTINE =====
debug081:009259BB ; Attributes: thunk
debug081:009259BB
debug081:009259BB sub_9259BB proc near ; CODE XREF: sub_9256FB1j
    
```

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](#)

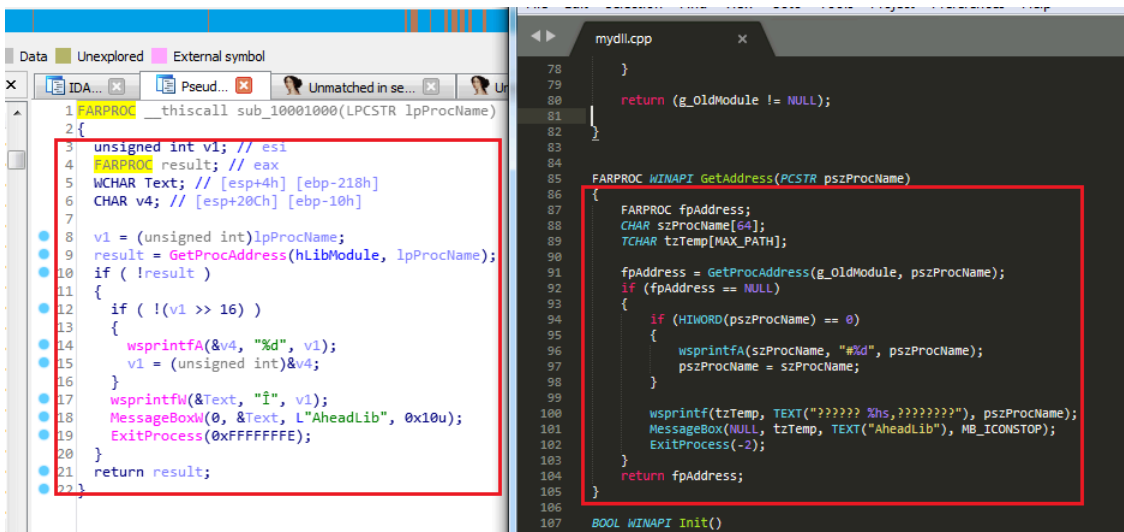
<https://github.com> > Yonsm > AheadLib

Yonsm/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](#)

Source: <https://lab52.io/blog/tokyox-dll-side-loading-an-unknown-artifact/>