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:

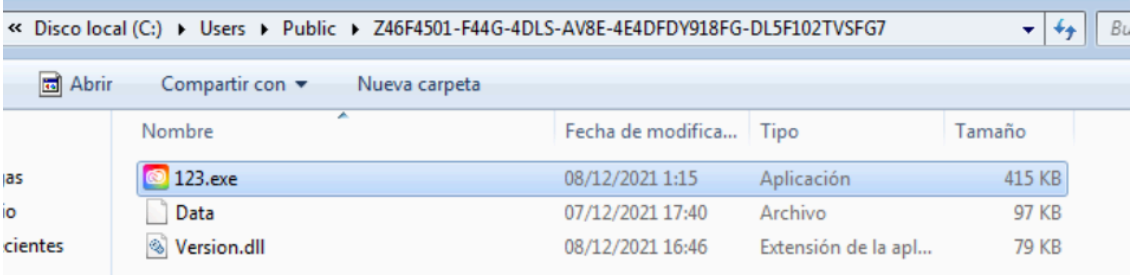
Creation Time 2021-12-09 02:46:43
First Submission 2021-12-09 08:48:20
Last Submission 2021-12-09 08:48:20
Last Analysis 2021-12-23 23:38:08

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:

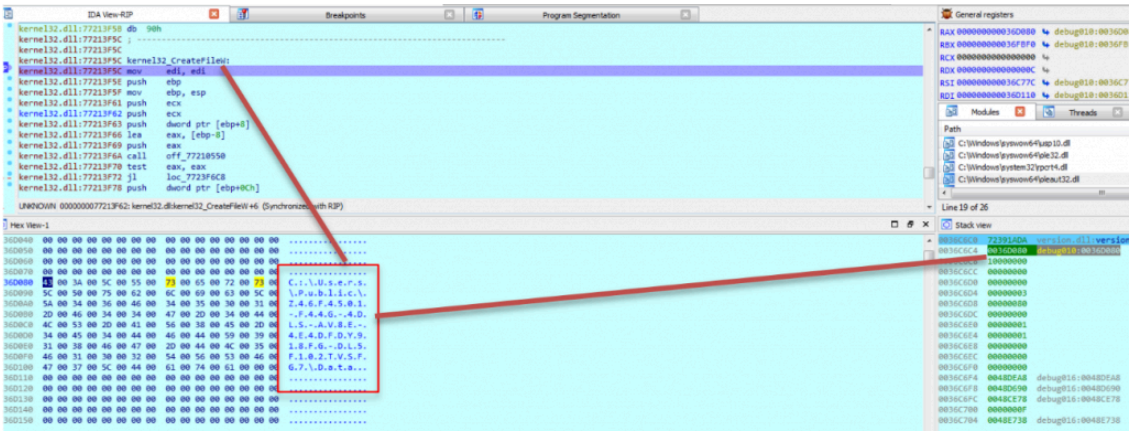


Once executed, we observe how the netsh.exe process tries to establish connections with port 443 of the IP address 31.192.107[.]187.

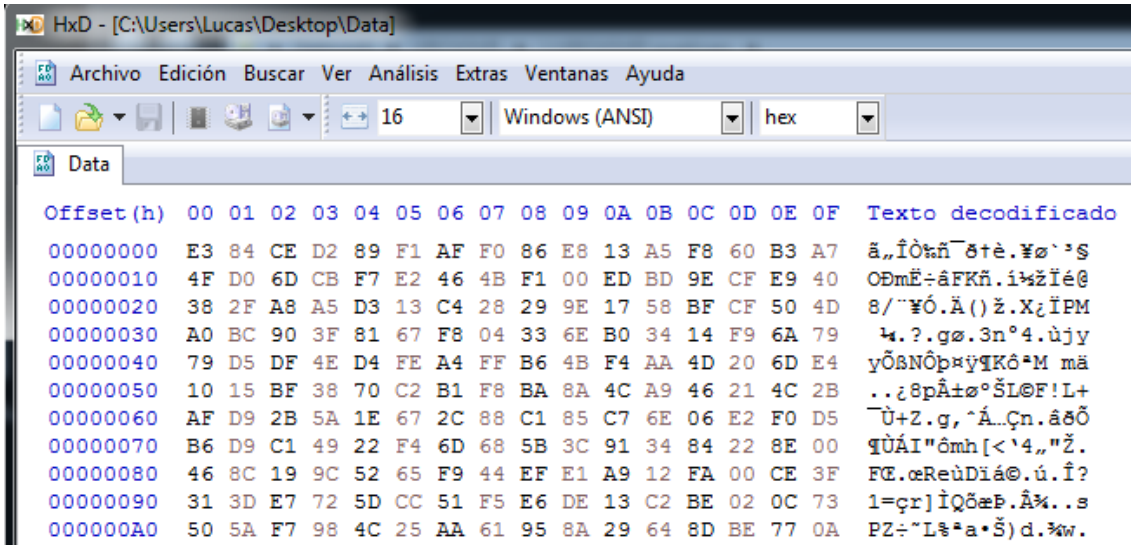
In this analysis we will focus on different aspects about the process; from double-clicking the binary 123.exe process (which is a copy of Creative.exe but in another path) to the execution of “TokyoX” already decrypted in memory.

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:

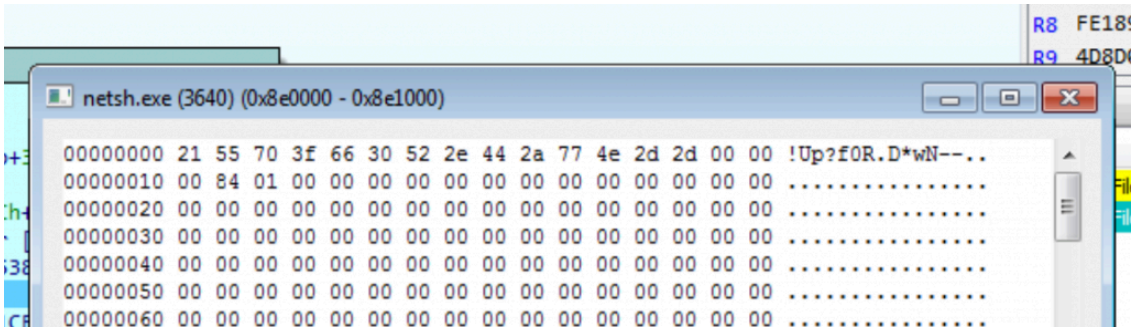


If we edit the Data file with a hexadecimal editor we will see their values, which will help us to identify it in memory later (beginning with E3 84):

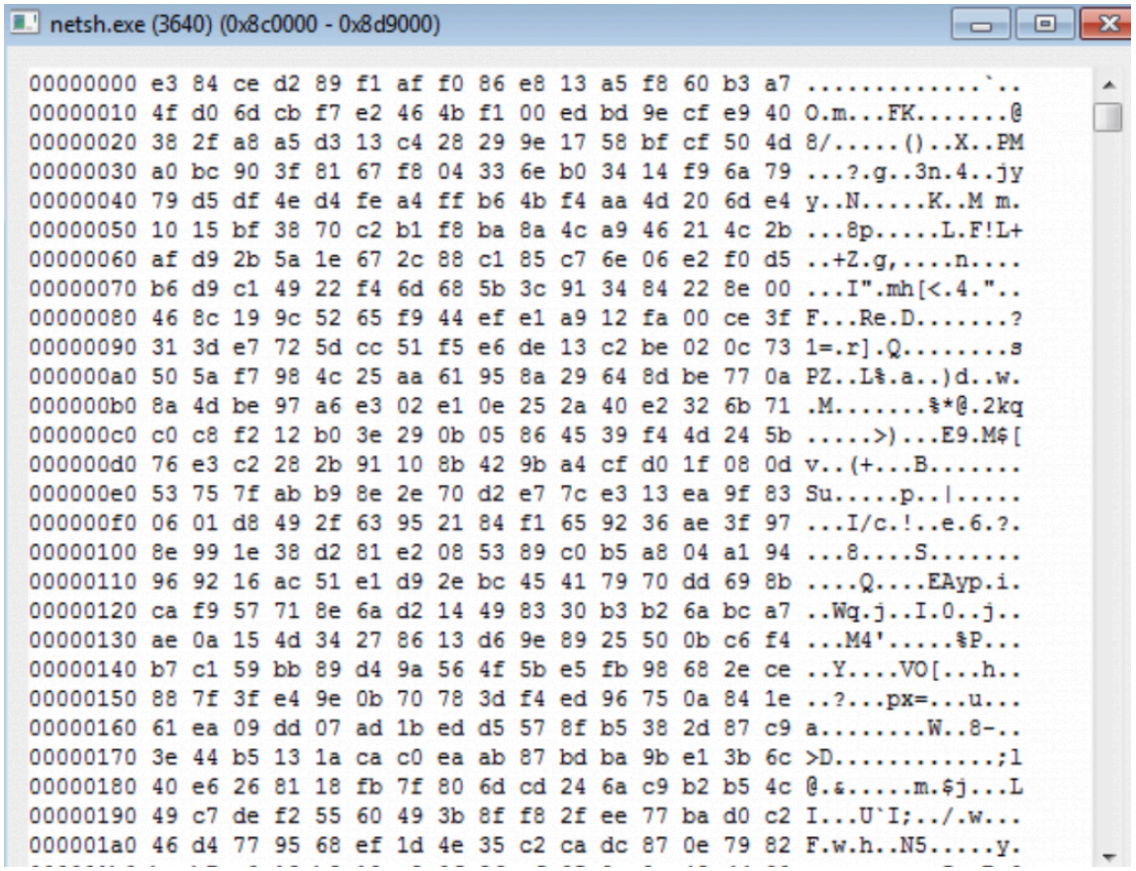


After reading the file from disk, a child process netsh.exe is created. This just-created child process is where several new memory segments will be located (a total of 5, including the final decrypted payload) to decrypt the final “TokyoX” payload. The APIs which were observed for the creation and writing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

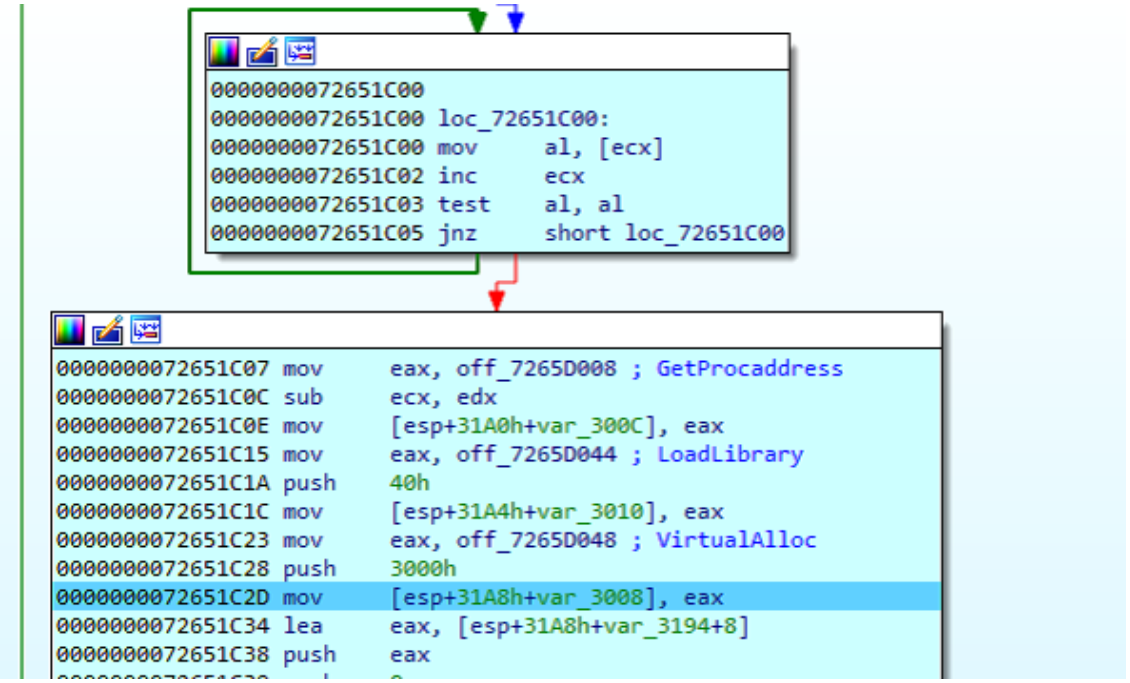
First, it creates two segments: 100Kb where the encrypted payload is located and which comes from the disc, and another one of 4Kb. In the 4Kb segment we observe how the following string is set (which will be the string used for the decrypting process):



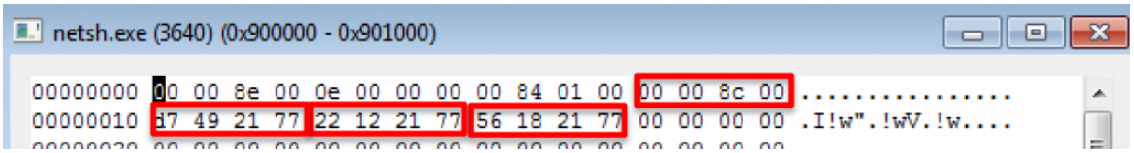
The other memory segment of 100Kb contains the following (encrypted content, as we see how it matches the content from Data file on Disk):



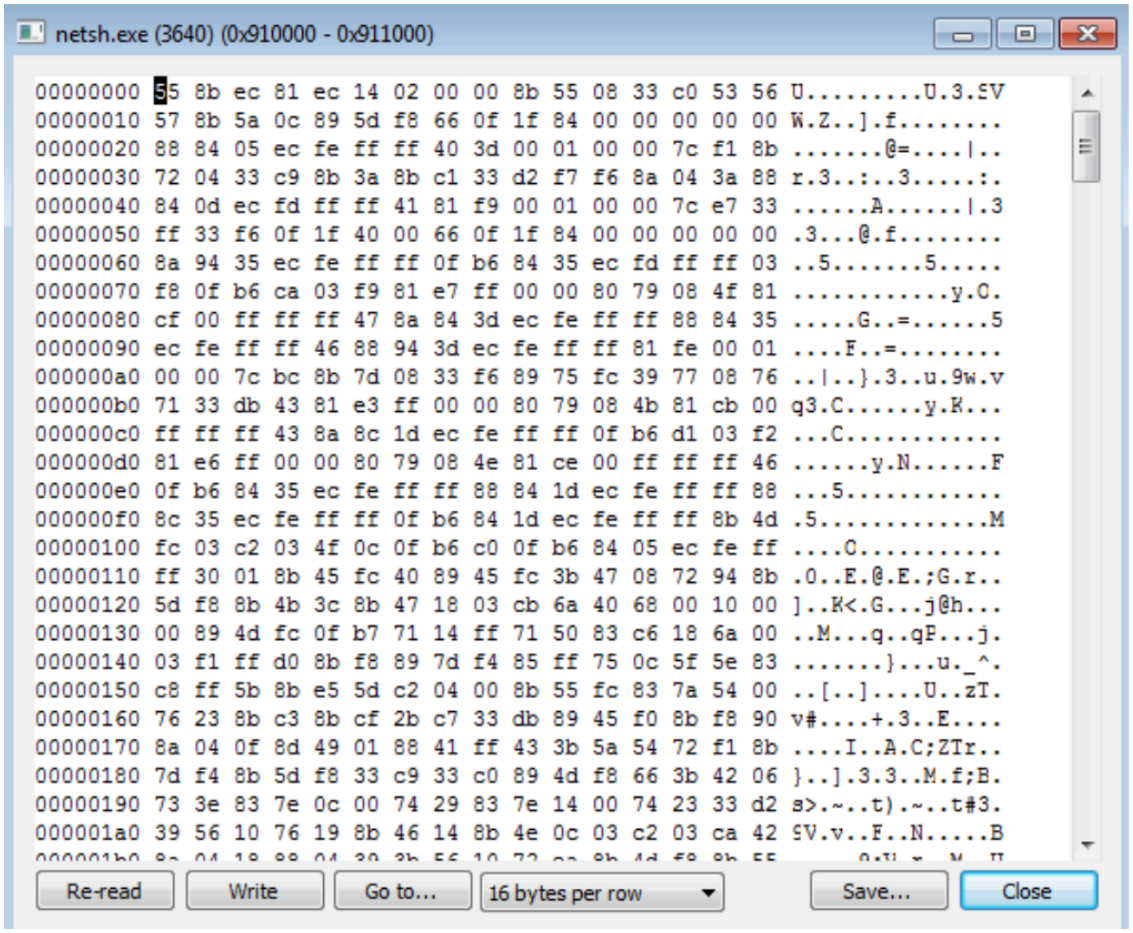
After the creation of these two segments, a third segment is allocated, where it is loaded the absolute memory addresses from several win32 APIs (VirtualAlloc, LoadLibrary, GetProcAddress, the home address of the coded payload, etc.) for its later use by the loader:



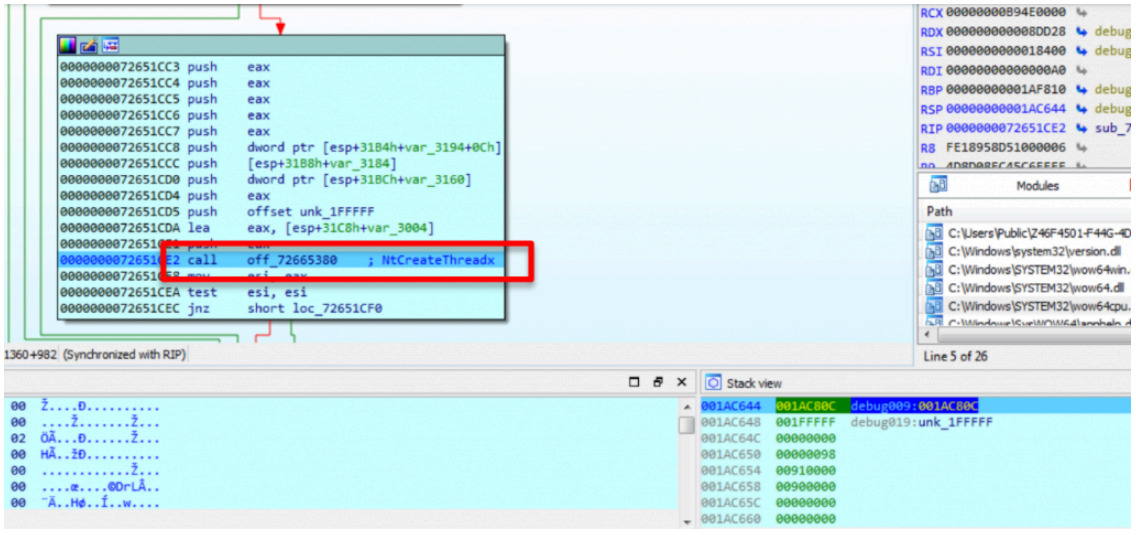
We can notice how the segment will have the memory addresses (starting from 123.exe they are located in netsh.exe segment through the version.dll code):



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



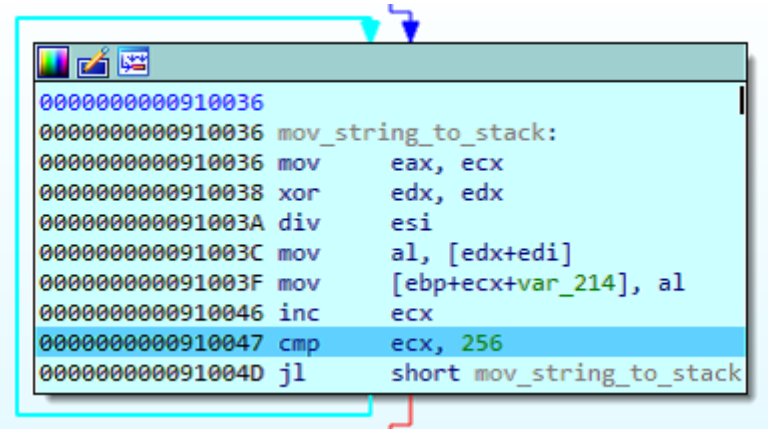
Finally, the “TokyoX” loader runs from the DLL (version.dll) in netsh.exe through the API NtcreateThreadEx and we see the start of the last page created in the stack:



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 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
```

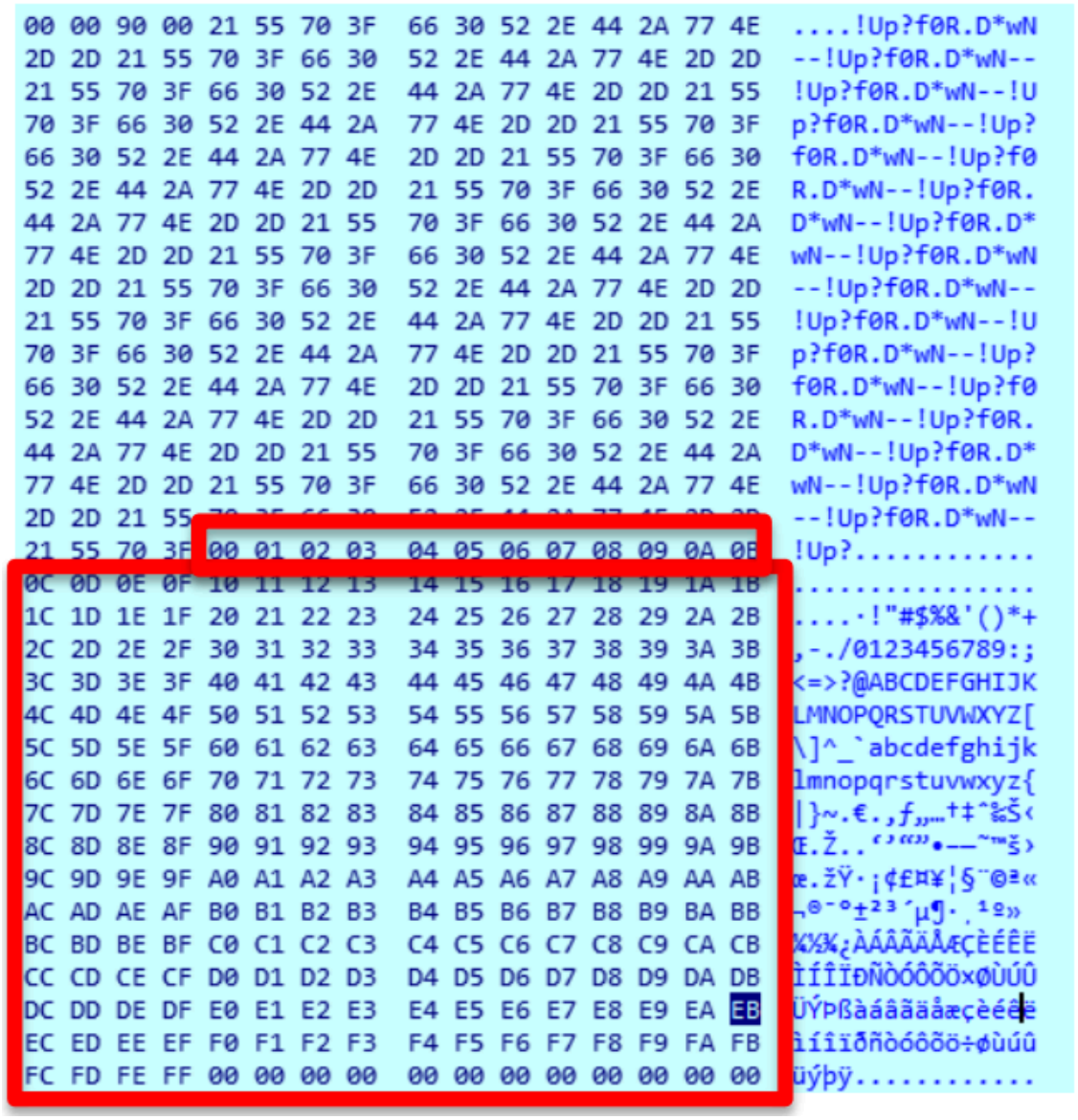
Once the execution is moved to the netsh.exe process, it takes the string located in the initial 4Kb segment, copies it into the stack and replicates it (0x100, 256 bytes) to match the specific block size of 256bytes. In the following screenshots we can observe how the block ends with the string “!Up?”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



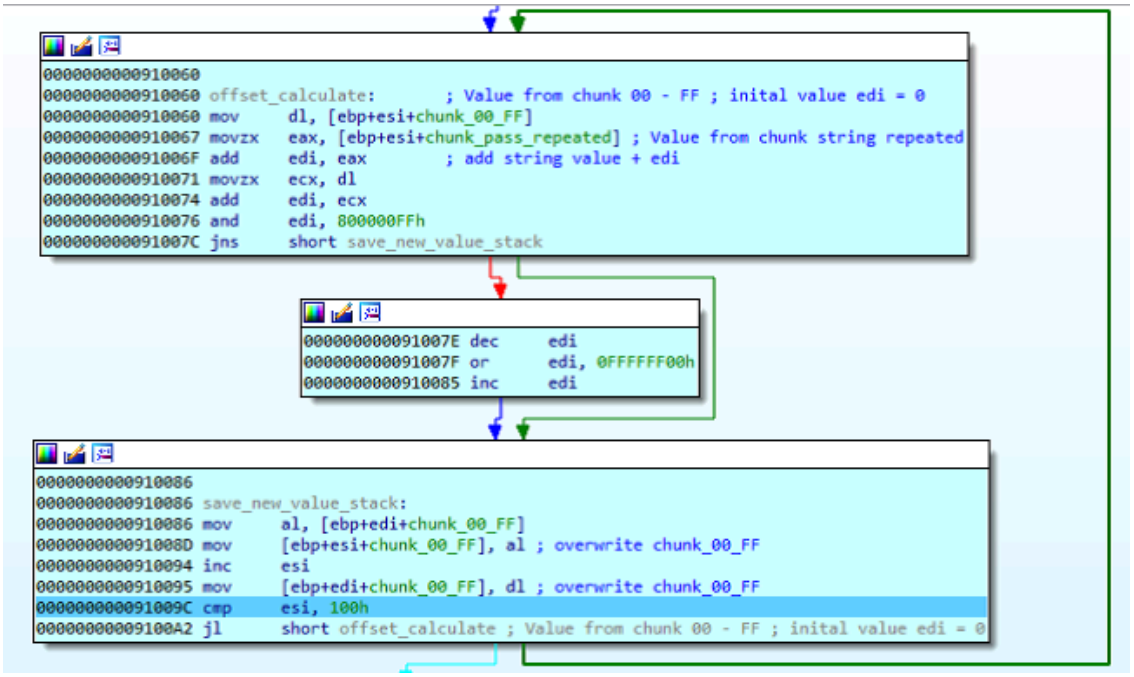
```
0000000000910036
0000000000910036 mov_string_to_stack:
0000000000910036 mov     eax, ecx
0000000000910038 xor     edx, edx
000000000091003A div     esi
000000000091003C mov     al, [edx+edi]
000000000091003F mov     [ebp+ecx+var_214], al
0000000000910046 inc     ecx
0000000000910047 cmp     ecx, 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.

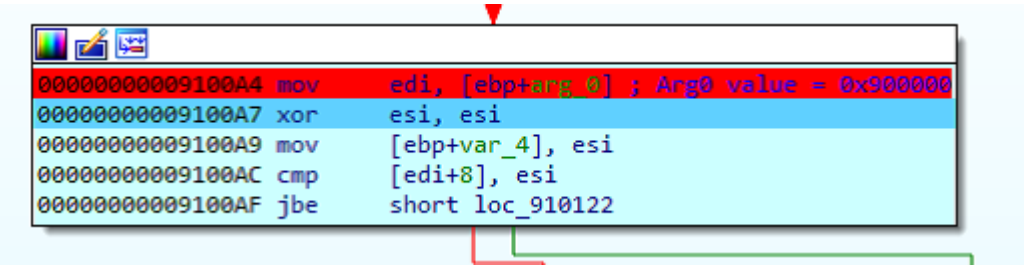


At this point, the loader transforms the 00-FF block with a series of additions combining the replicated string's block with the 00-FF block, as we can see:



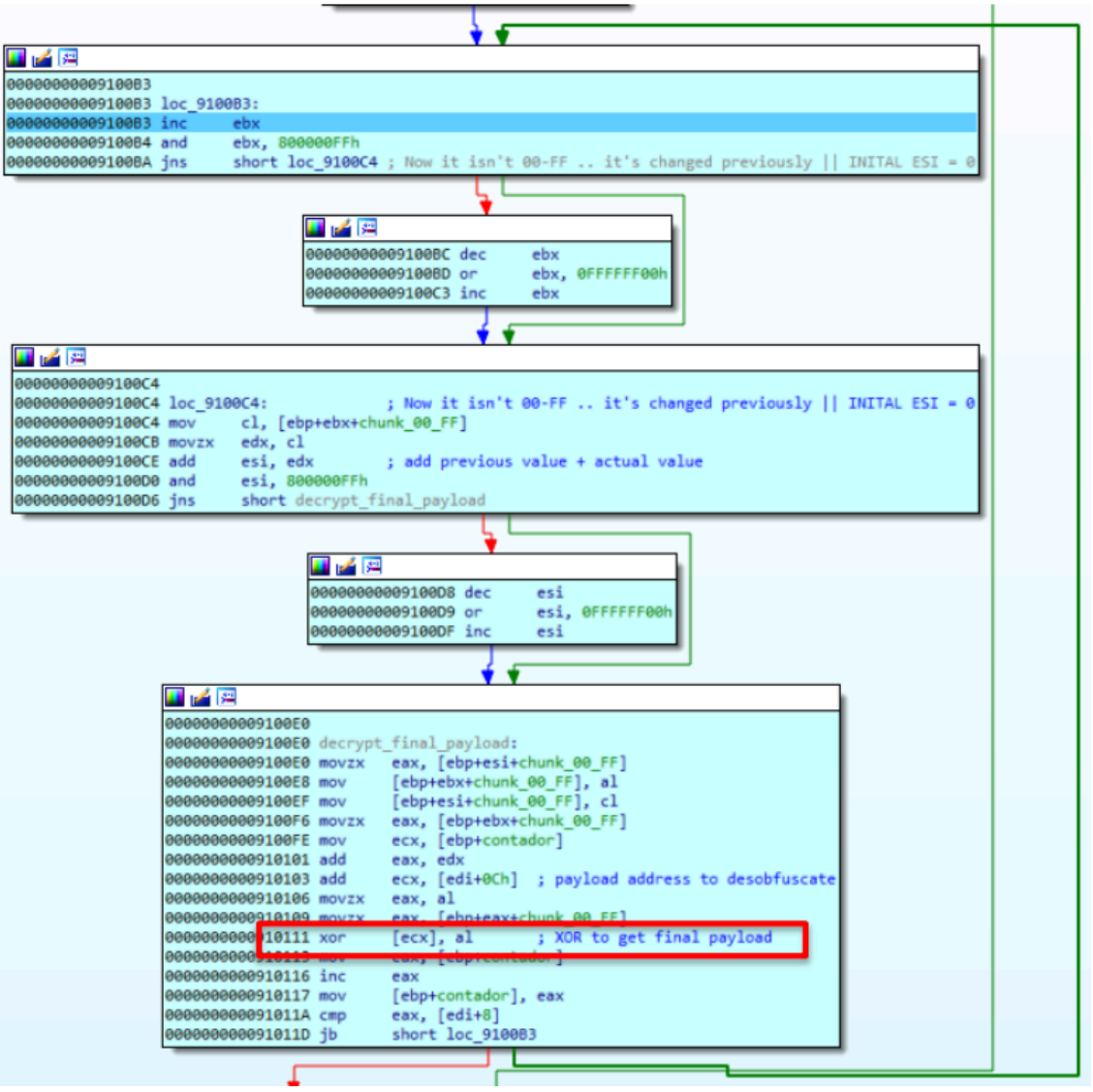
The combination of the blue block (in following image) and the 00-FF block (pointed in red in previous image) results in the following block in memory, marked in red in the image:

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:

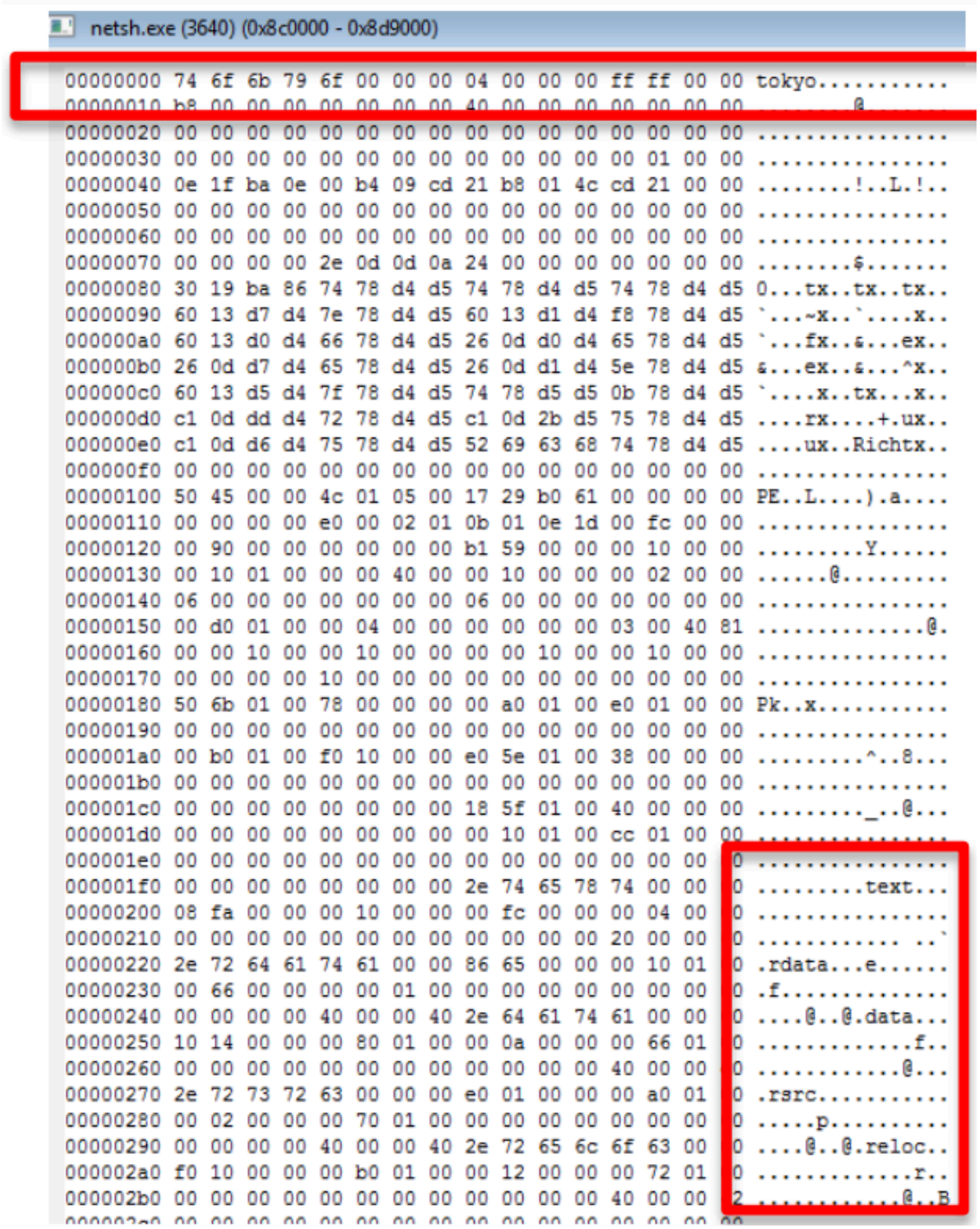


After this, the decrypting process for the final payload begins. The decrypting process gets two values from the second block, exchanges and adds them, and the result serves as a final index to recover the element from the second block with which the xor will be achieved through the coded block.

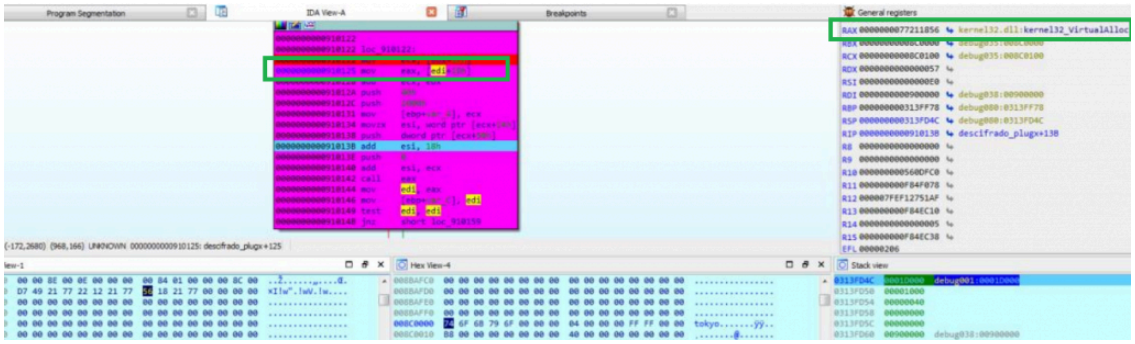
This description of the decryption algorythm has been identified as the **RC4 algorythm**.



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”:



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

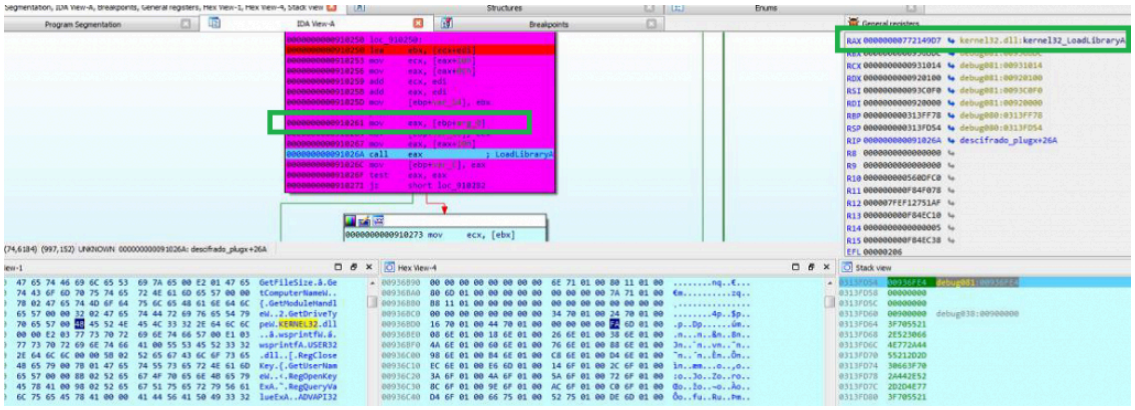


This creates another memory segment in the process netsh.exe with RWX licenses (that of 116Kb) which will be used to load the PE:

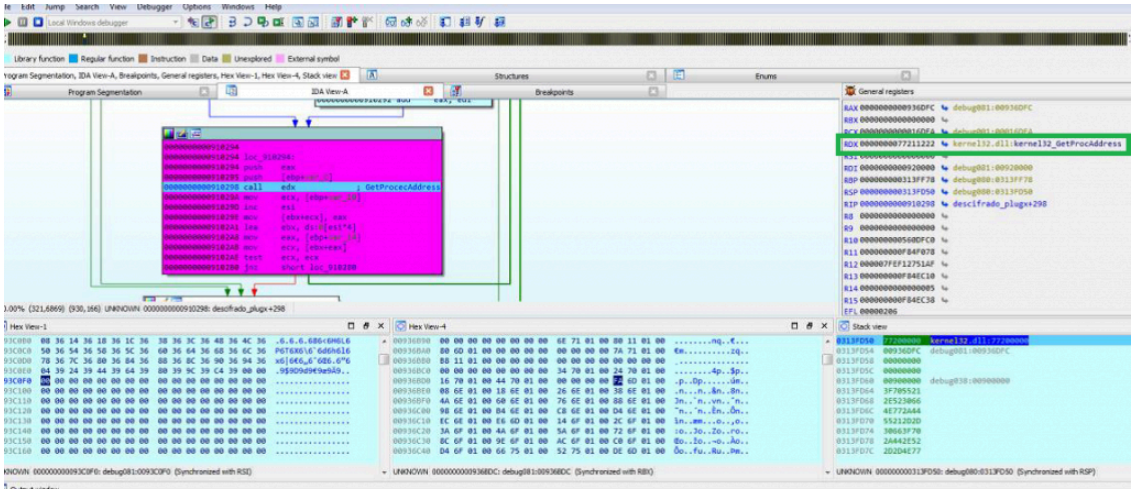
0x8c0000	Private: Commit	100 kB	RWX	100 kB	100 kB
0x8e0000	Private: Commit	4 kB	RWX	4 kB	4 kB
0x900000	Private: Commit	4 kB	RWX	4 kB	4 kB
0x910000	Private: Commit	4 kB	RWX	4 kB	4 kB
0x920000	Private: Commit	116 kB	RWX	4 kB	4 kB

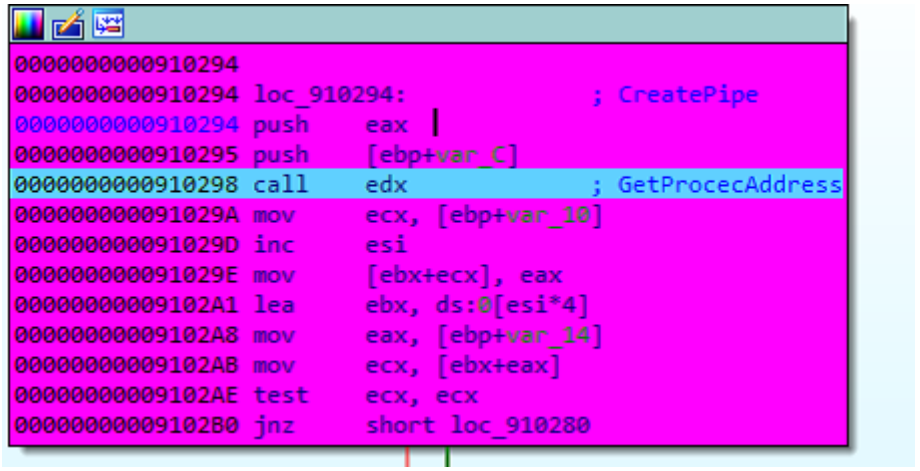
In this new segment, it maps the binary using the virtual addresses as the regular Windows PE loader would do.

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:

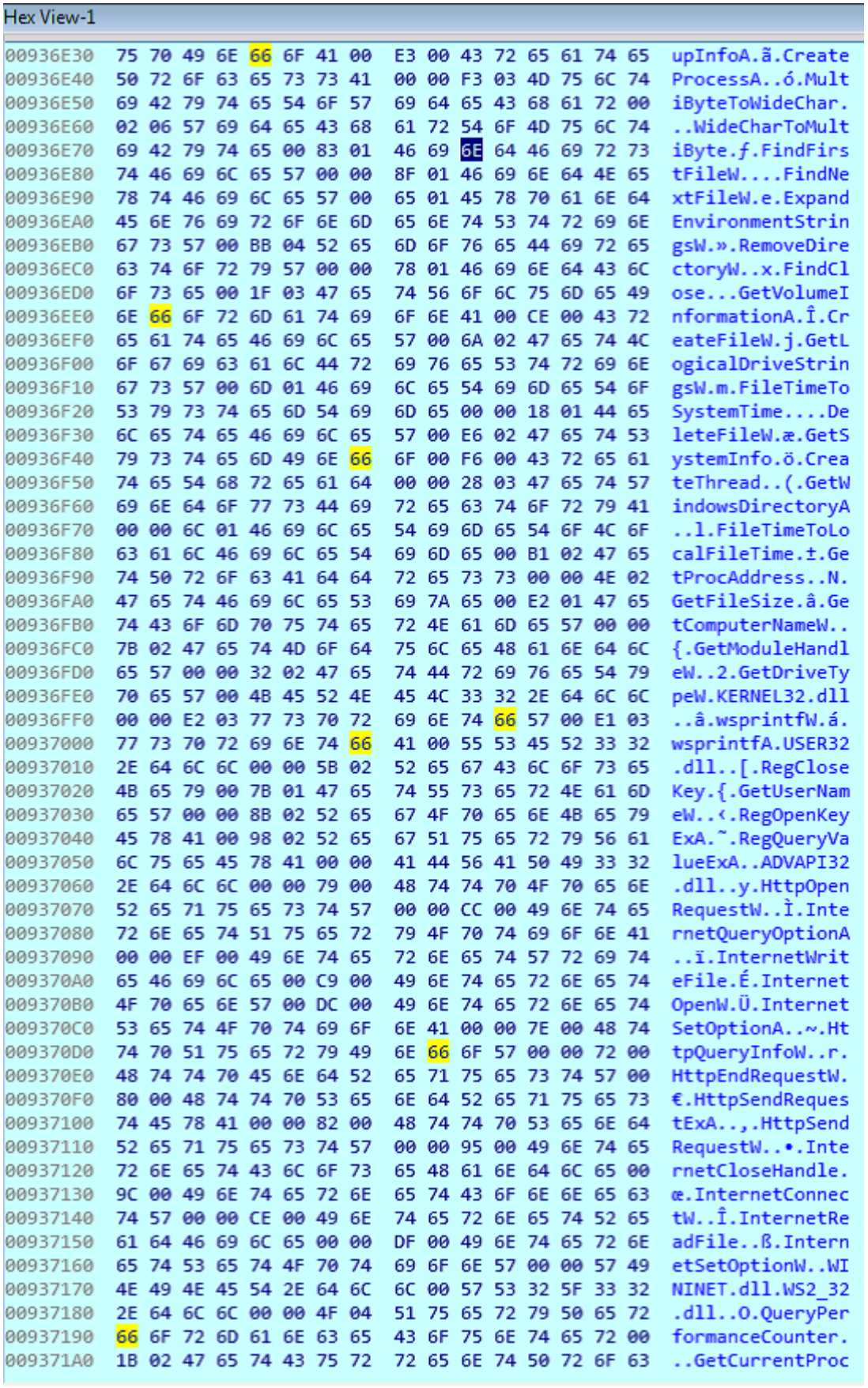


Then it calls GetProcAddress() to get the addresses of certain functions:

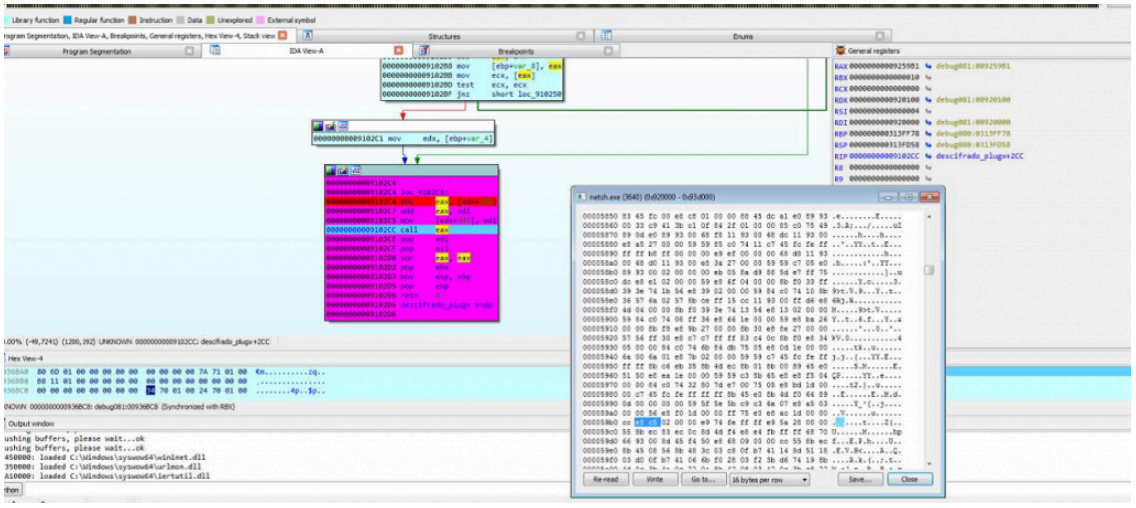


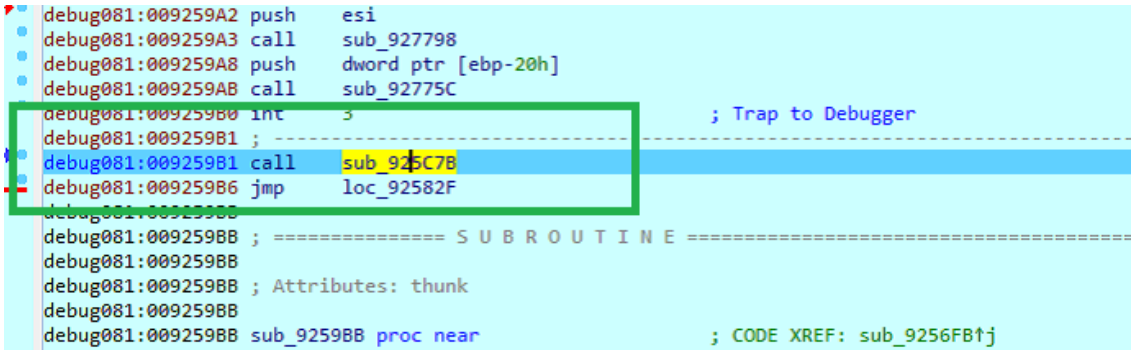


Next, the libraries and functions block may be appreciated:



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 › 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 ...

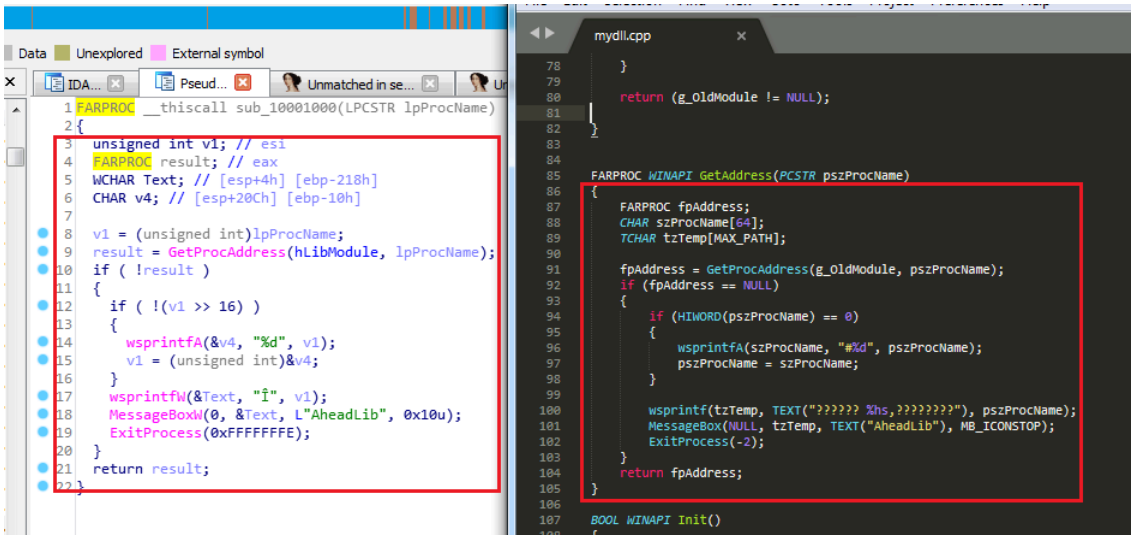
README.md

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

---