Mustang Panda Employ Publoader Through ClaimLoader: Yes.. another DLL Side-Loading Technique Delivery via Phishing

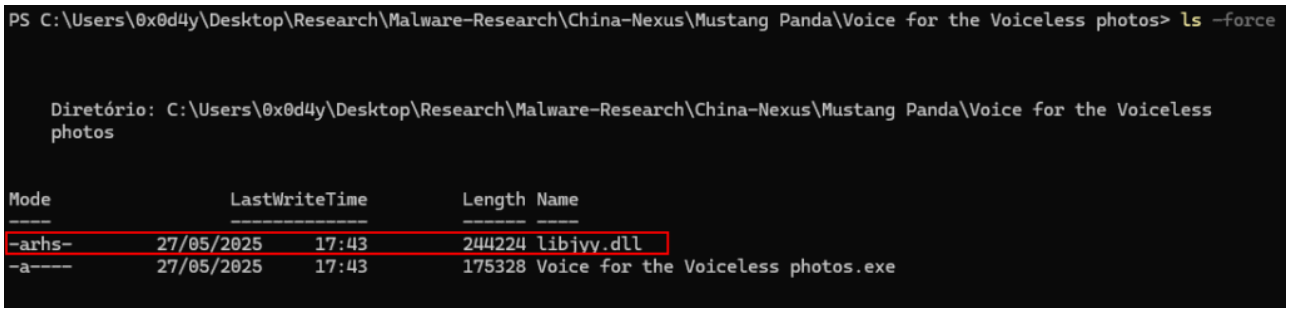
⋮ 10/6/2025

In this new post I will analyze, once again, an execution chain of payloads delivered via Phishing from another Threat Actor China-Nexus, however, implementing the same TTP, yes, DLL Side-Loading!

In this research, I will explore a campaign by Threat Actor Mustang Panda, identified in June 2025 by IBM's X-Force, which targets the Tibetan community for obviously political reasons. The initial loader is delivered via a .ZIP file containing a decoy named '**Voice for the Voiceless Photos.exe**', a clear reference to the Dalai Lama's book, and a DLL that doesn't appear in Windows Explorer through the *dir* and *ls* commands.

```
PS C:\Users\0x0d4y\Desktop\Research\Malware-Research\China-Nexus\Mustang Panda\Voice for the Voiceless photos> ls

    Diretório: C:\Users\0x0d4y\Desktop\Research\Malware-Research\China-Nexus\Mustang Panda\Voice for the Voiceless
    photos


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        27/05/2025     17:43         175328 Voice for the Voiceless photos.exe
```

But when we use the **ls -force** command, we are able to observe the DLL **libjyy.dll**, containing the **-arhs-** modes that allow it to be hidden.



I think it's worth breaking down this payload obfuscation technique, as it's part of weaponizing the **.ZIP** file, preventing the victim from seeing the DLL and consequently not finding its presence strange. This DLL has four active attributes:

- **a** → File marked for backup (Windows default flag)
- **r** → Read-only
- **h** → Hidden, that's why it doesn't show up in Explorer
- **s** → System File. Explorer tends to hide this type too, even if "Show hidden files" is enabled

In other words, Explorer doesn't show libiyv.dll because it's marked as both hidden and system. By default, Explorer ignores files with both of these attributes unless you go to **Folder Options → View → Uncheck "Hide protected operating system files."** This prevents the user from suspecting the presence of an unexpected file when opening the directory and clicking on the initial payload.

That said, let's analyze this campaign. For the entire analysis workflow, I'll be using Malcat <3.

Analyzing Voice for the Voiceless Photos.exe

Triage

When we open the decoy present in the .ZIP, we can observe some information that has already been seen and identified in previous Threat Actors China-Nexus campaigns, in addition to a pseudo product name among other information unique to this sample.

- **Company Name**: *Hefei Nora Network Technology Co., Ltd.* -> No company with this name was identified, but it has been used in other Threat Actors China-Nexus campaigns.
- **Legal Copyright**: *Hefei Nora Network Technology Co., Ltd. All rights reserved.*
- **Product Name**: *FFWallpaper Widgets Jyy*
- **PdbPath**: *G:\CLIENT\fhbemb\src\bin\Release\fhjyy.pdb*

File information | Check online intelligence | **Type**

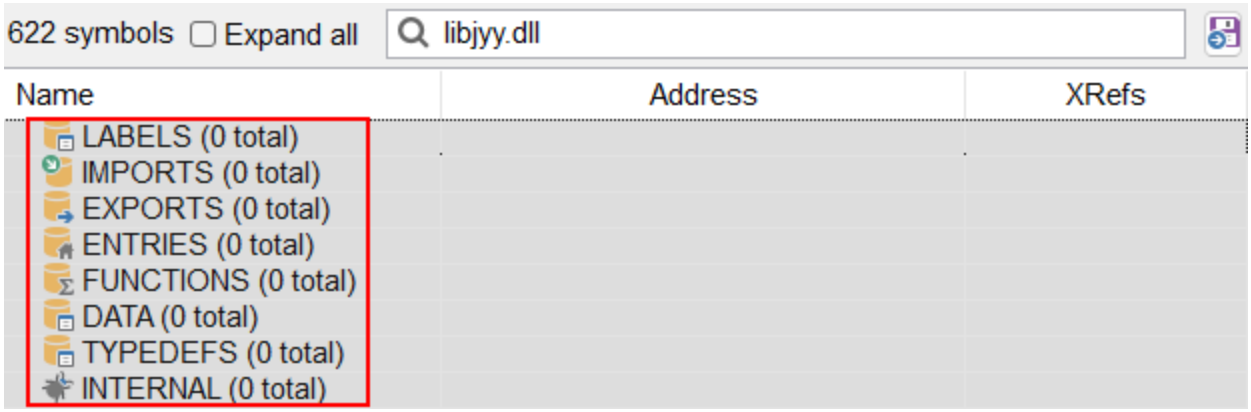
File name:	Voice for the Voiceless photos.exe
File size:	175328 bytes (171.2 KiB)
MD5:	be4ed0d3aa0b2573927a046620106b13
SHA1:	0b81544cd5e66a36d90a033f60a0ece1cd3506a8
SHA256:	79bf3258e03fd1acb395dc184fbe5496dfa4b3d6a3f9f4598c5df13422cc600d
TLSH:	T12e049d5272d1d072e46f097005a8d1a19f3efe611f308abb3398523e5fa11c19a769bf
Imphash:	d26a41ce888dfa28b9c760121338e8e6

Program
PE

Metadata

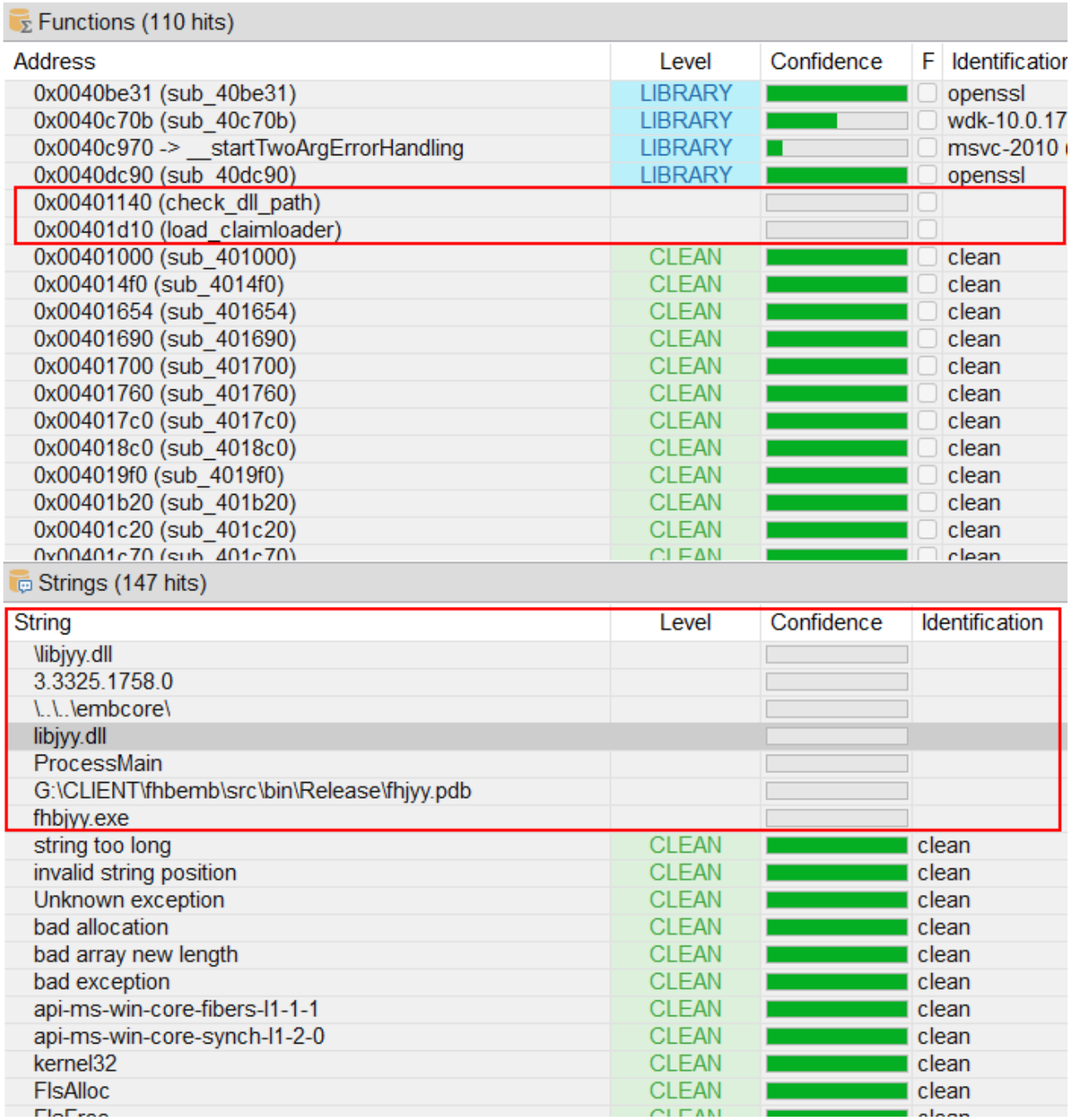
Compile date:	2023-05-12 01:39:17
VersionInfo:	
CompanyName:	合肥诺拉网络科技有限公司
FileDescription:	FFWallpaper Widgets Jyy
FileVersion:	1.0.0.1
InternalName:	fhbjyy
LegalCopyright:	合肥诺拉网络科技有限公司，保留所有权利。
OriginalFilename:	fhbjyy.exe
ProductName:	FFWallpaper Widgets Jyy
ProductVersion:	1.0.0.1
Debug:	
Date.Debug.Codeview:	2023-05-12 01:39:17
Path:	G:\CLIENT\fhbemb\src\bin\Release\fhjyy.pdb
Date.Debug.VcFeature:	2023-05-12 01:39:17
Date.Debug.Pogo:	2023-05-12 01:39:17

Since we know there's a hidden DLL in the directory, we can search for static import references to this DLL and its functions, but this search yielded no results. This suggests that this DLL will likely be loaded during decoy execution.



To triage this decoy, I used Malcat's Kesakode feature, which aims to identify patterns of known functions/strings from malware families, and functions/strings that are identified as benign. However, it's also possible to identify unknown functions and strings, which could be indicators of maliciousness, as they haven't been identified in known malware, and especially not in benign software. This can give us a direction on where to begin our more in-depth analysis.

Below you can see two unknown functions (which I renamed) and unknown strings, one of which even allows us to assume it is the name of a function (**ProcessMain**).



Functions (110 hits)

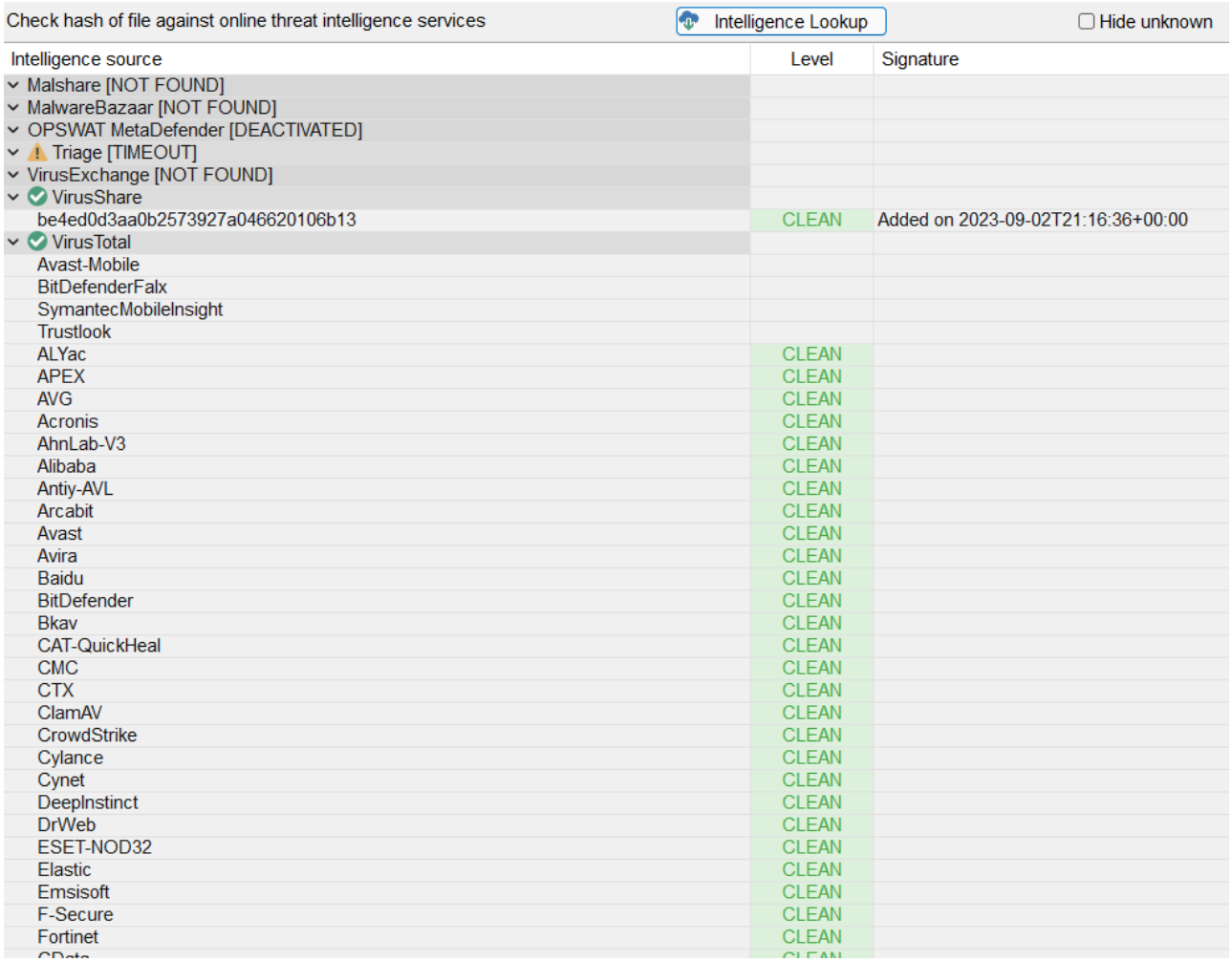
Address	Level	Confidence	F	Identificatior
0x0040be31 (sub_40be31)	LIBRARY			openssl
0x0040c70b (sub_40c70b)	LIBRARY			wdk-10.0.17
0x0040c970 -> __startTwoArgErrorHandling	LIBRARY			msvc-2010
0x0040dc90 (sub_40dc90)	LIBRARY			openssl
0x00401140 (check_dll_path)				
0x00401d10 (load_claimloader)				
0x00401000 (sub_401000)	CLEAN			clean
0x004014f0 (sub_4014f0)	CLEAN			clean
0x00401654 (sub_401654)	CLEAN			clean
0x00401690 (sub_401690)	CLEAN			clean
0x00401700 (sub_401700)	CLEAN			clean
0x00401760 (sub_401760)	CLEAN			clean
0x004017c0 (sub_4017c0)	CLEAN			clean
0x004018c0 (sub_4018c0)	CLEAN			clean
0x004019f0 (sub_4019f0)	CLEAN			clean
0x00401b20 (sub_401b20)	CLEAN			clean
0x00401c20 (sub_401c20)	CLEAN			clean
0x00401c70 (sub_401c70)	CLEAN			clean

Strings (147 hits)

String	Level	Confidence	Identification
\libjyy.dll			
3.3325.1758.0			
\..\..\embcore\			
libjyy.dll			
ProcessMain			
G:\CLIENT\fhbemb\src\bin\Release\fhjyy.pdb			
fhbjyy.exe			
string too long	CLEAN		clean
invalid string position	CLEAN		clean
Unknown exception	CLEAN		clean
bad allocation	CLEAN		clean
bad array new length	CLEAN		clean
bad exception	CLEAN		clean
api-ms-win-core-fibers-l1-1-1	CLEAN		clean
api-ms-win-core-synch-l1-2-0	CLEAN		clean
kernel32	CLEAN		clean
FlsAlloc	CLEAN		clean
FlsFree	CLEAN		clean

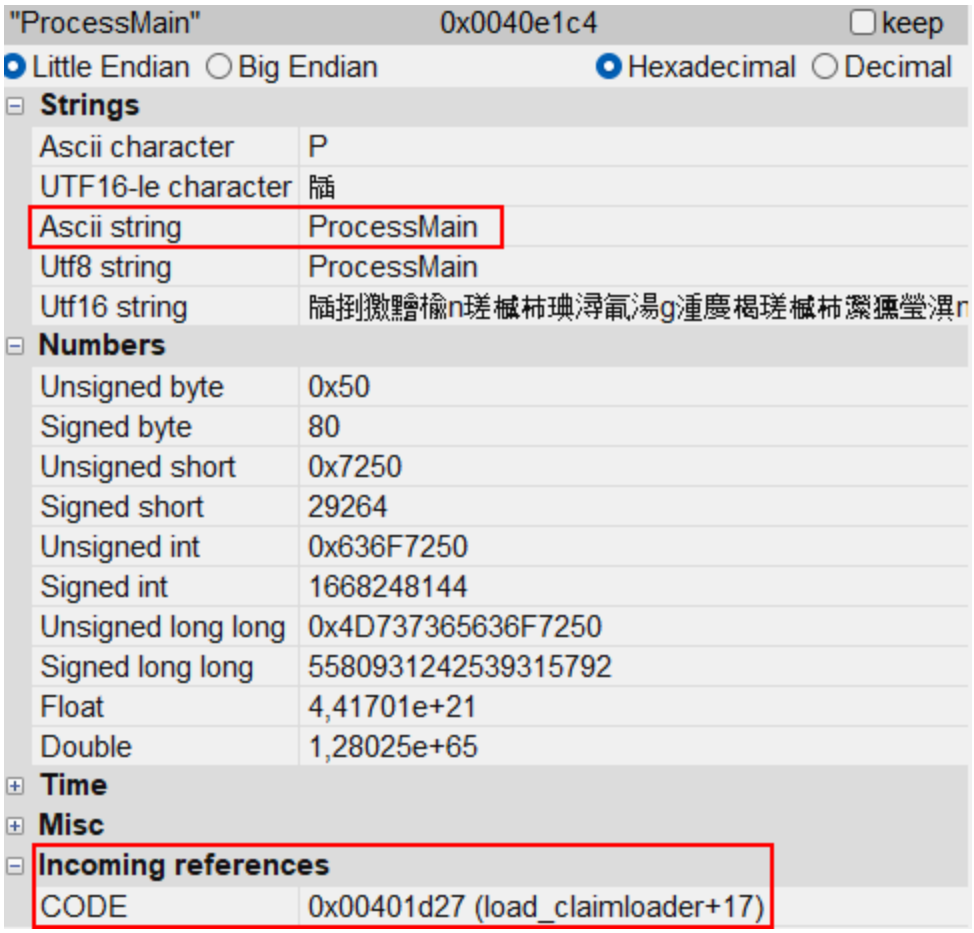
To complete this initial screening, I also used the built-in Intelligence Lookup feature, which, through a pre-configuration of API keys, allows us to check the existence and verdict of this sample in widely known services, such as **VirusTotal**, **VirusShare**, **Triage**, **Malshare**, **MalwareBazaar**, etc.

And as we can see in the image below, the lookup allowed us to identify that this binary is *Clean* based on the samples in which it exists.

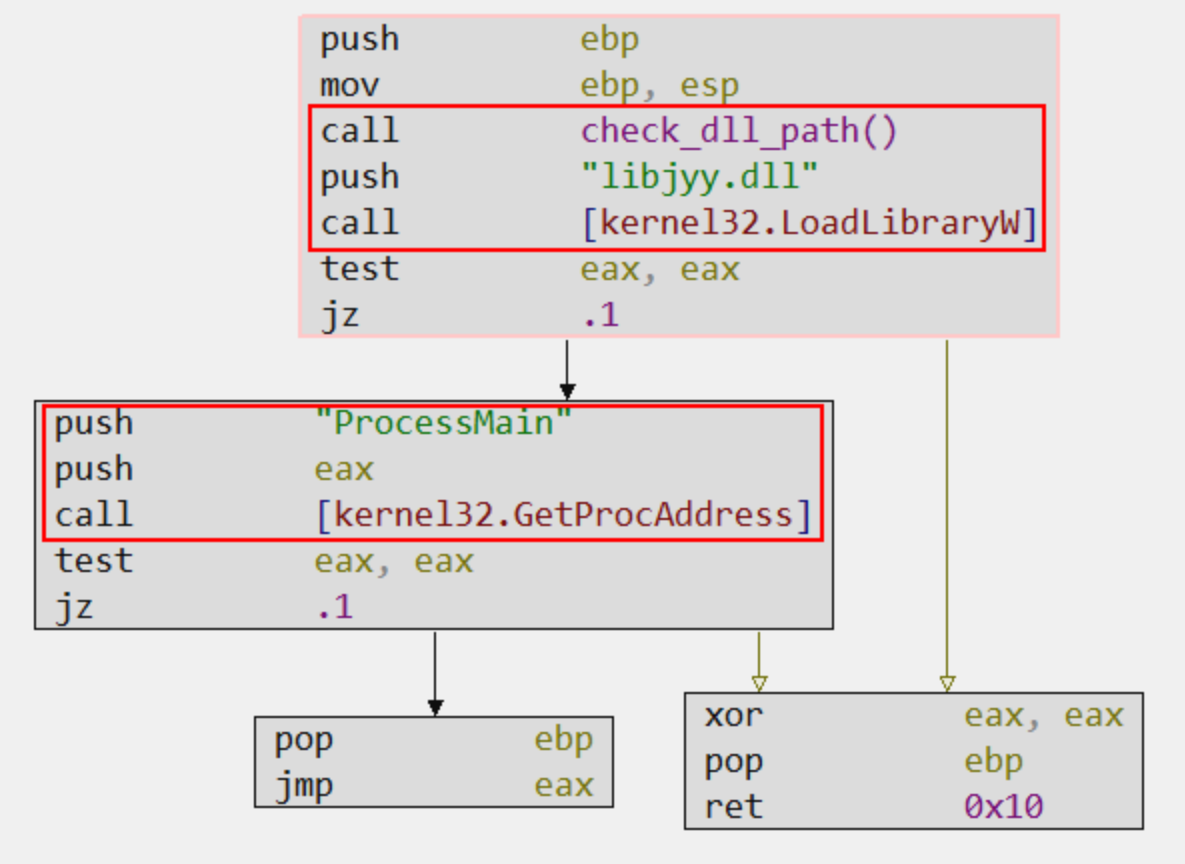


Reverse Engineering the Decoy

When analyzing the String reference identified as unknown in the previous section, we can see that it is referenced in a function, which Kesakode also identified as unknown. Therefore, let's start with this function identified at offset **0x00401d27**.



The function is quite small, divided into just four blocks of assembly code, but it focuses on its main functionality: loading the hidden DLL present in the .ZIP file delivered to the victim. And if we look closely, the decoy loads the hidden DLL dynamically, through LoadLibraryW, and specifically the ProcessMain function. This allows us to clearly identify where to begin analyzing the hidden DLL.



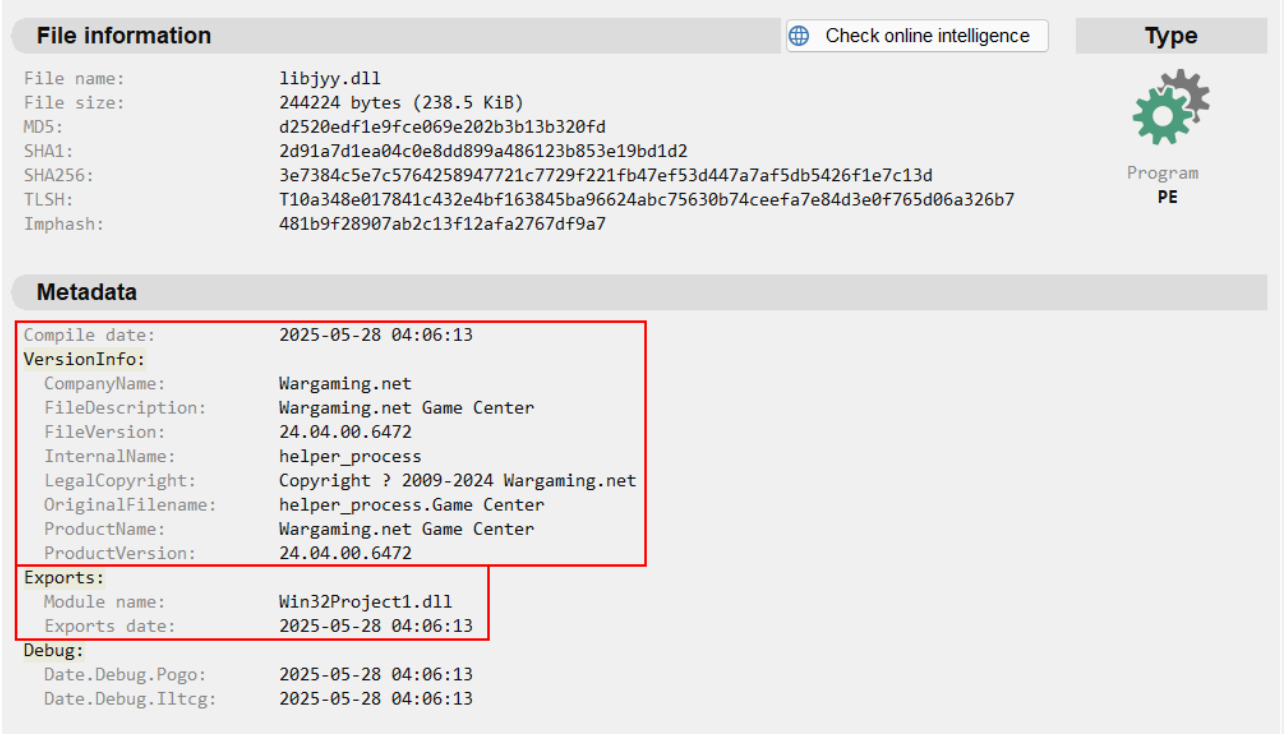
And this is the only function of this decoy, to load the real malicious payload which consists of the hidden DLL named **libjyy.dll**.

Analyzing libjyy.dll

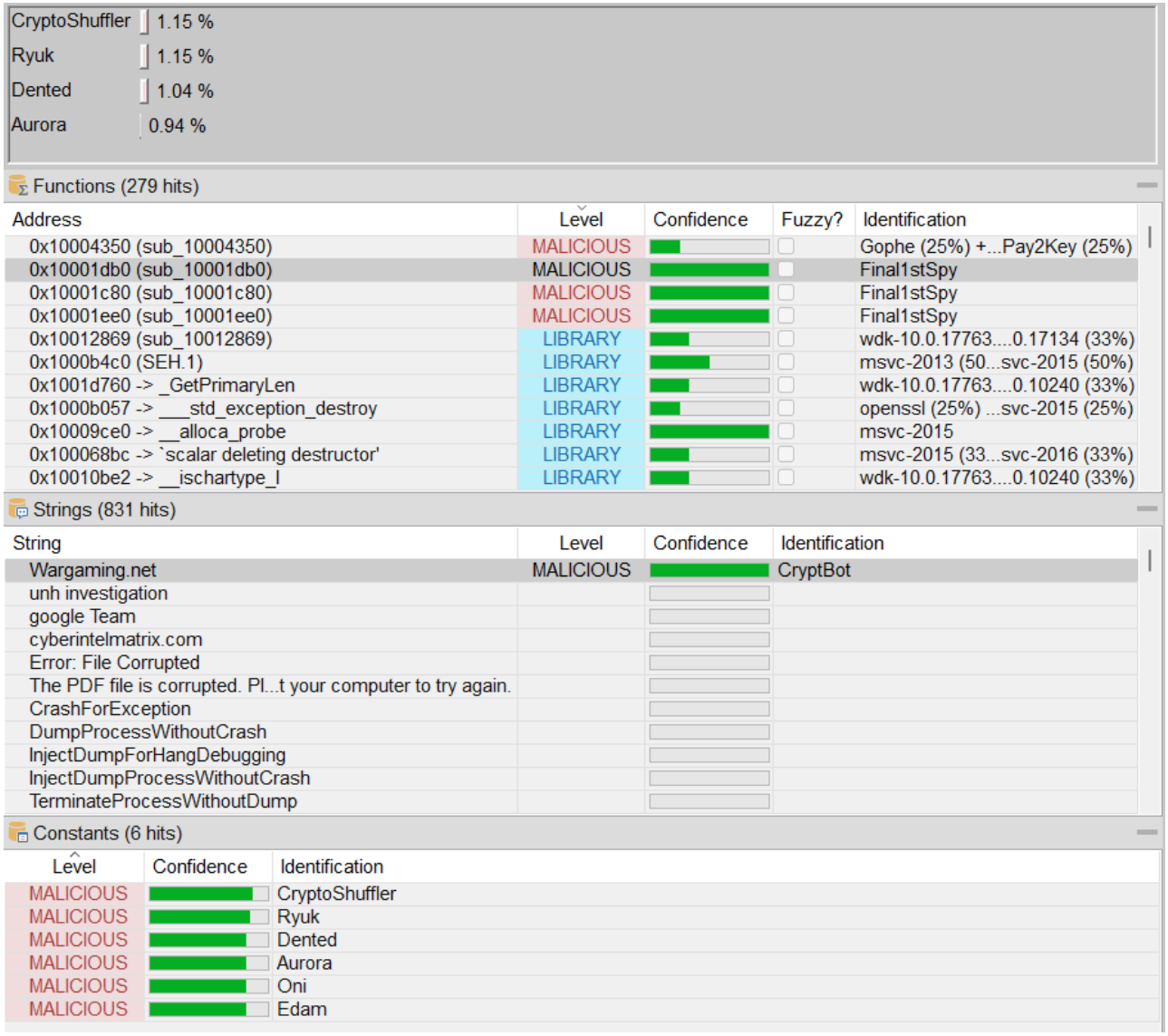
Triage

Following the same screening flow, we open libjyy.dll, we can observe some interesting information.

- **Company Name**: *Wargaming.net. Name of a real Chinese game development company*.
- **Legal Copyright**: *Copyright 2009-2024 Wargaming.net*
- **Product Name**: *Wargaming.net Game Center*
- **Module Name**: *Win32Project1.dll*



By submitting this sample to *Kesakode*, we can observe some functions identified as malicious, and further identify the malware family in which the function is related. We can also observe unknown strings identified as malicious, as well as malicious constants identified in this sample that are known to have been observed in other malware families listed in the image below.



I also submitted this sample to the native Intelligence Lookup engine, which allows us to identify the presence of this sample in the **Malware Bazaar**, **Triage**, **VirusShare** and **VirusTotal** services, bringing the verdict and the name of the signatures identified in these malware databases.

Check hash of file against online threat intelligence services | Intelligence Lookup

Intelligence source	Level	Signature
Malshare [NOT FOUND]		
MalwareBazaar		
reversinglabs	MALWARE	Win32.Trojan.DllHijack
yoroi_yomi	MALWARE	Malicious File
intezer	SUSPICIOUS	
spamhaus_hbl	SUSPICIOUS	
triage	SUSPICIOUS	
cape		
cert-pl_mwdb		
filescan-io		
vxcube	CLEAN	
OPSWAT MetaDefender [DEACTIVATED]		
Triage		
250626-jevw8asky5:helper_process.Game Center_.dll	SUSPICIOUS	
250626-kn5y8aspz4:helper_process.GameCenter_.dll	SUSPICIOUS	
VirusExchange		
helper_process		First seen on 2025-06-27T09:03:59Z
VirusShare		
d2520edf1e9fce069e202b3b13b320fd	MALWARE	Added on 2025-06-02T16:18:52+00:00
VirusTotal		
ALYac	MALWARE	Gen:Variant.Midie.167524
AVG	MALWARE	Win32:MalwareX-gen [Trj]
AhnLab-V3	MALWARE	Trojan/Win.Generic.R709736
Arcabit	MALWARE	Trojan.Midie.D28E64
Avast	MALWARE	Win32:MalwareX-gen [Trj]
BitDefender	MALWARE	Gen:Variant.Midie.167524
Bkav	MALWARE	W32.AIDetectMalware
CAT-QuickHeal	MALWARE	Trojan.Ghanarava.1754207227b320fd
CTX	MALWARE	dll.trojan.dllhijack
CrowdStrike	MALWARE	win/malicious_confidence_100% (W)
Cylance	MALWARE	Unsafe
Cynet	MALWARE	Malicious (score: 100)
DrWeb	MALWARE	Trojan.Siggen31.35843
ESET-NOD32	MALWARE	a variant of Win32/Agent.AHOT
Elastic	MALWARE	malicious (high confidence)
Emsisoft	MALWARE	Gen:Variant.Midie.167524 (B)
Fortinet	MALWARE	W32/Agent.AHOT!tr
GData	MALWARE	Gen:Variant.Midie.167524
Google	MALWARE	Detected
Ikarus	MALWARE	Trojan.Win32.Agent
K7AntiVirus	MALWARE	Riskware (00584baa1)
K7GW	MALWARE	Riskware (00584baa1)
Kaspersky	MALWARE	HEUR:Trojan.Win32.DLLhijack.gen
Kingsoft	MALWARE	Win32.Trojan.DLLhijack.gen
Lionic	MALWARE	Trojan.Win32.DLLhijack.4!c
Malwarebytes	MALWARE	Malware.AI.1622005164
MaxSecure	MALWARE	Trojan.Malware.379267698.susgen
McAfeeD	MALWARE	ti!3E7384C5E7C5

In addition to the screening features already explored above, Malcat also allows us to use CAPA to expand screening by identifying potential capabilities implemented in the sample. Below is the output of the capabilities identified by CAPA integrated with Malcat.

CAPA script

```
CAPA framework by mandiant (https://github.com/mandiant/capa) using malcat for analysis.
Everything except the malcat comes from the github repository.
Mandiant rules can be found in <analysis DATA DIR>/scripts/capa/rules.zip.
You can add your own as .yml files in <USER DATA DIR>/scripts/capa/all_rules/*.yml.
```

ATT&CK information

ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Obfuscated Files or Information::Indicator Removal from Tools [T1027.005]
	Obfuscated Files or Information [T1027]
DISCOVERY	File and Directory Discovery [T1083]
	System Information Discovery [T1082]
	System Location Discovery [T1614]
EXECUTION	Command and Scripting Interpreter [T1059]
	Shared Modules [T1129]

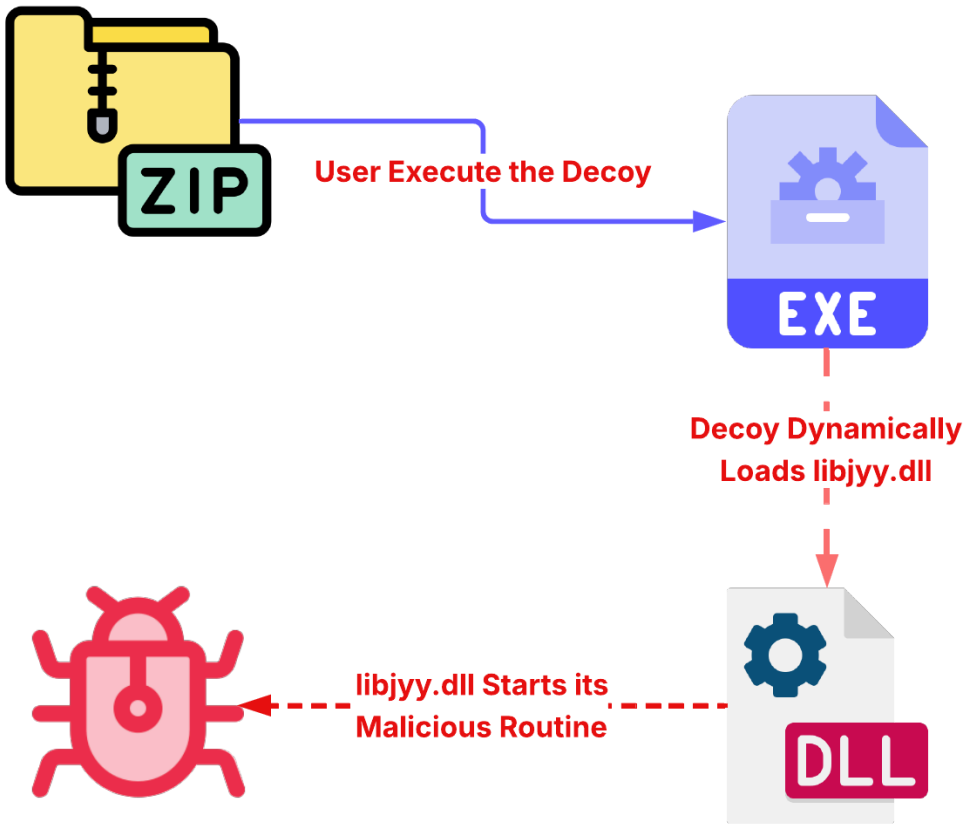
Malware Behavior Catalog

MBC Objective	MBC Behavior
ANTI-STATIC ANALYSIS	Executable Code Obfuscation::Argument Obfuscation [B0032.020]
	Executable Code Obfuscation::Stack Strings [B0032.017]
DATA	Encode Data::XOR [C0026.002]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]
DISCOVERY	File and Directory Discovery [E1083]
	System Information Discovery [E1082]
EXECUTION	Command and Scripting Interpreter [E1059]
FILE SYSTEM	Create Directory [C0046]
	Read File [C0051]
	Writes File [C0052]
PROCESS	Allocate Thread Local Storage [C0040]
	Set Thread Local Storage Value [C0041]
	Terminate Process [C0018]

111 capabilities found

CAPABILITY	NAMESPACE
contain obfuscated stackstrings (2)	anti-analysis/obfuscation/string/stackstring
get geographical location (8)	collection
encode data using XOR (2)	data-manipulation/encoding/xor
contain a resource (.rsrc) section	executable/pe/section/rsrc
contain a thread local storage (.tls) section	executable/pe/section/tls
accept command line arguments (2)	host-interaction/cli
query environment variable	host-interaction/environment-variable
reference absolute stream path on Windows (5)	host-interaction/file-system
create directory	host-interaction/file-system/create
enumerate files on Windows	host-interaction/file-system/files/list
read file on Windows (4)	host-interaction/file-system/read
write file on Windows (5)	host-interaction/file-system/write
allocate thread local storage	host-interaction/process
get thread local storage value (2)	host-interaction/process
set thread local storage value (2)	host-interaction/process
terminate process (7)	host-interaction/process/terminate
link function at runtime on Windows (6)	linking/runtime-linking
link many functions at runtime	linking/runtime-linking
linked against CPP standard library	linking/static
parse PE header (4)	load-code/pe

With this screening complete, we'll move on to the functions we've identified as suspicious based on the information we've collected so far. Below, we can see the macro flow of the infection chain identified so far.



Reverse Engineering libjyy.dll

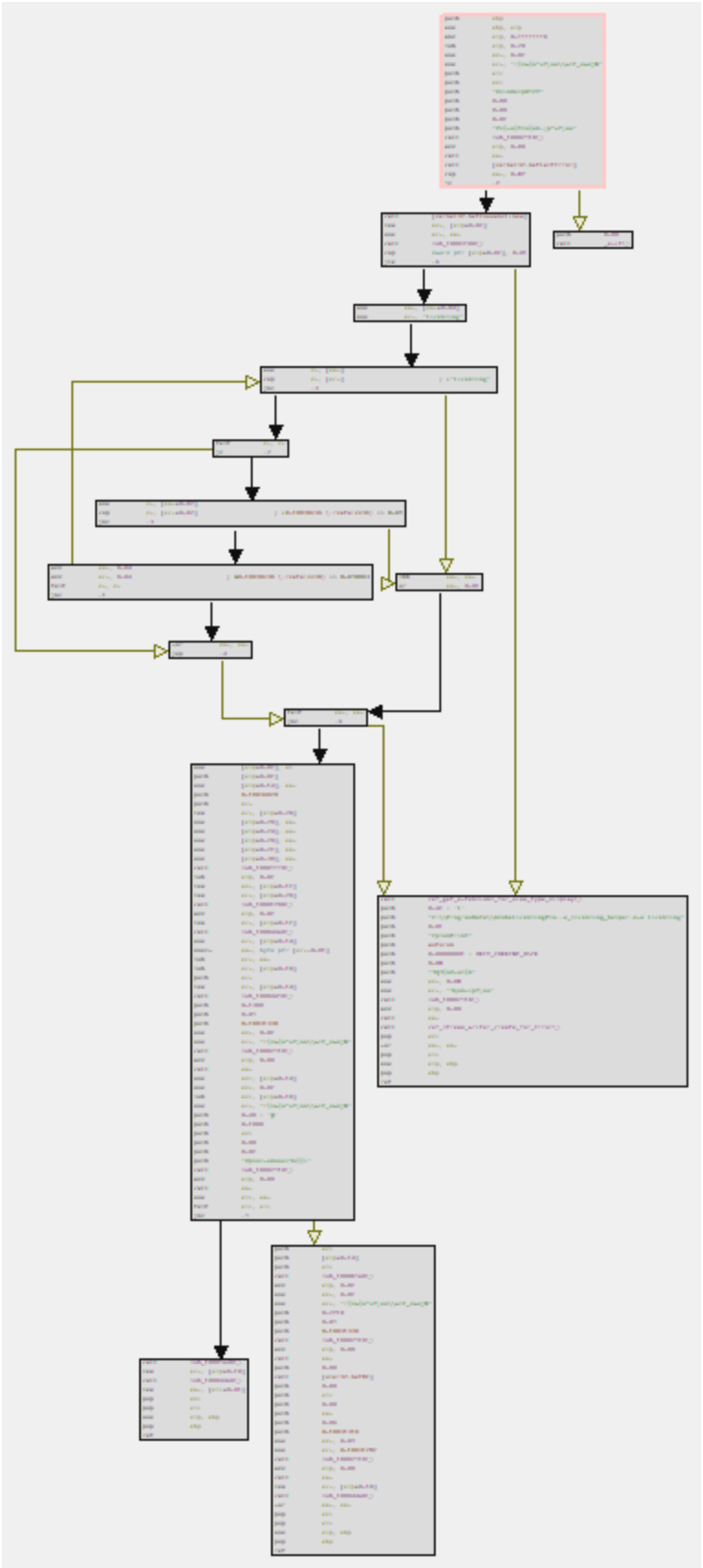
Now that we have enough information, let's start our in-depth analysis through the function in which the decoy loads and calls **libjyy.dll**, **ProgramMain**.

Basically ProcessMain is a wrapper for two functions:

- cef_string_utf16_utf8_clear;
- `cef_string_Start_Handler`.

```
ProcessMain() {
    E8DBFAFFFF                  call        cef_string_utf16_utf8_clear()
    E986E3FFFF                  jmp         cef_string_Start_Handler()
}

    CC                          int3
    CC                          int3
    CC                          int3
    CC                          int3
    CC                          int3
    CC                          int3
```

Let's start with the first function, named **cef_string_utf16_utf8_clear** by the sample itself. Below, we can observe its execution flow in a macro view.



This is the malware's main function, which, according to the IBM X-Force report, is the Claimloader. Basically, this function performs the following actions:

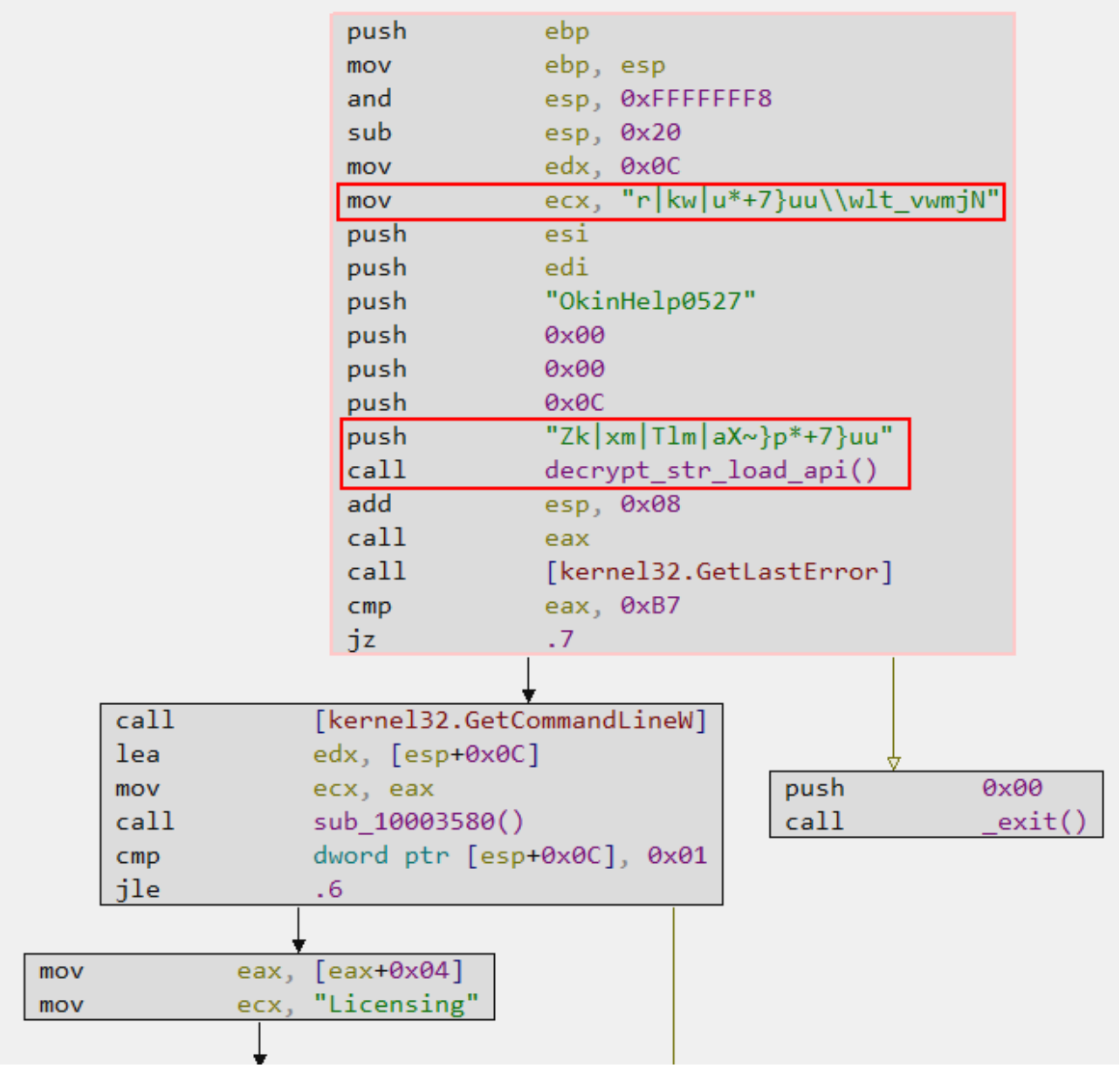
- **Decrypts Strings**;
- **Checks the Passed Argument**;
- **Creates a Mutex**;
- **Creates Persistence on the Infected System**;
- **Executes the Decryption Routine**;
- And one of the subroutines called through this main function appears to execute code through a Thread.

We will analyze each aspect in detail below, to better understand the implementations carried out by adversaries, and consequently their purposes.

String Decryption

The string decryption routine is quite simple, essentially an XOR operation on a single-byte key. It's called whenever the *Claimloader* needs to dynamically load an API. Therefore, all strings encrypted by this algorithm refer to APIs that will be dynamically loaded to implement specific capabilities, primarily allocation, injection, and execution of the next stage (*Publoader*).

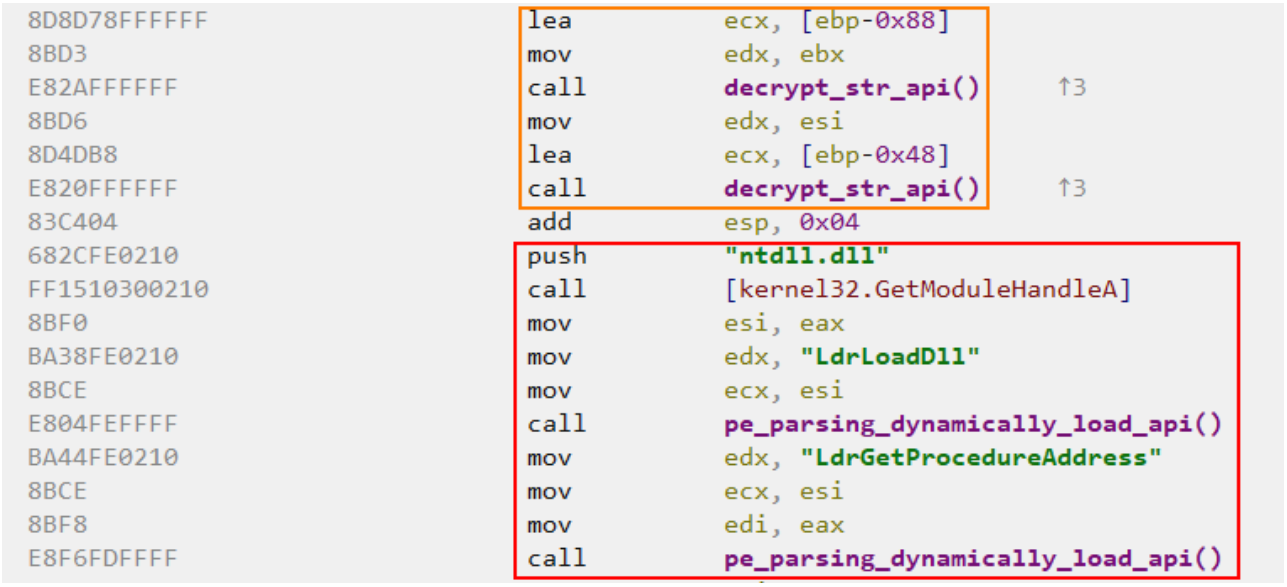
Below, we can see two encrypted strings being moved to the Stack, with the aim of being decrypted and finally, dynamically loaded through the function (renamed by me) **decrypt_str_load_api**.



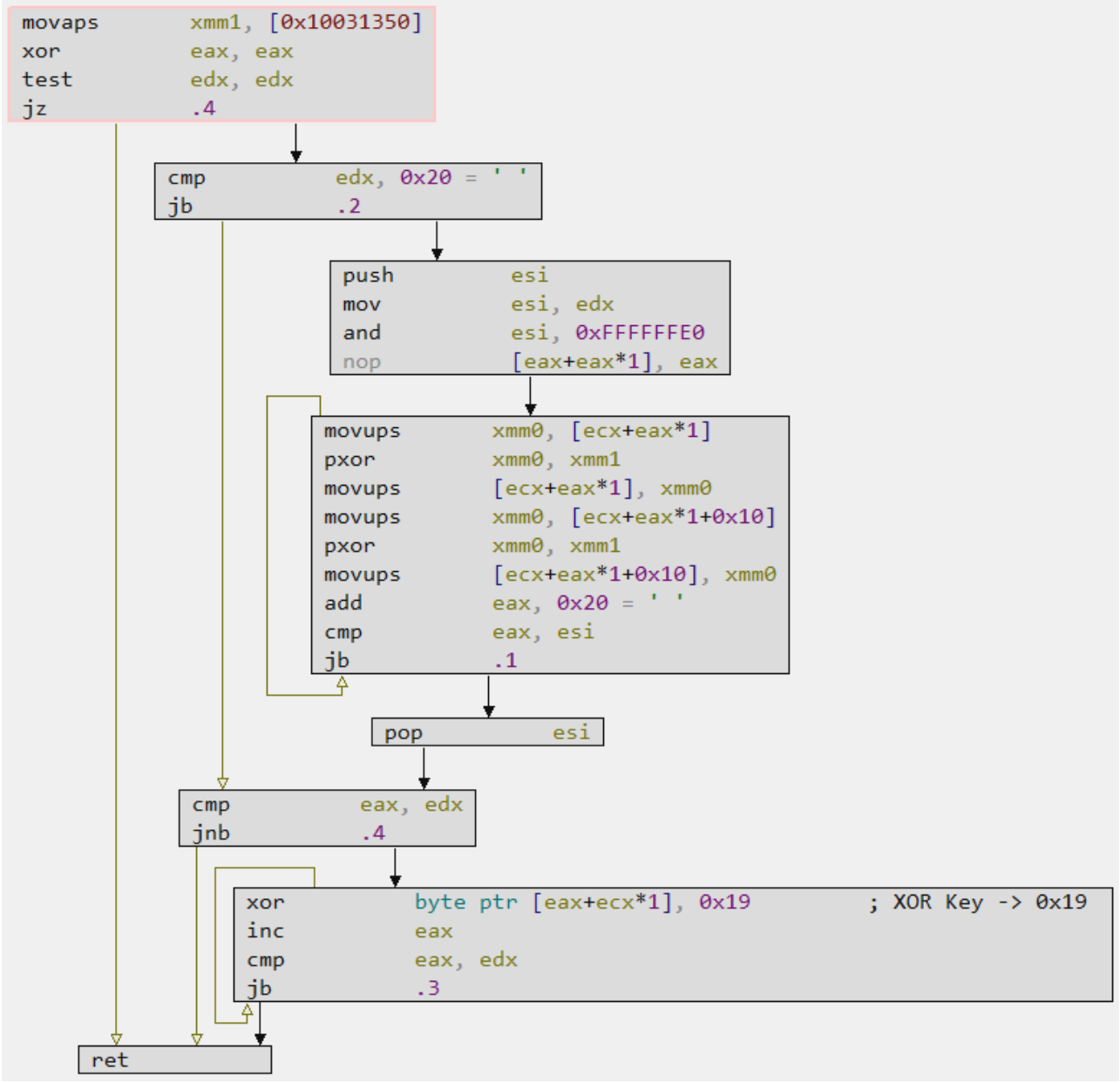
Within the **decrypt_str_load_api**function, we can observe its two main functionalities, implemented through the functions:

- **decrypt_str_api**;
- **pe_parsing_dynamically_load_api**.

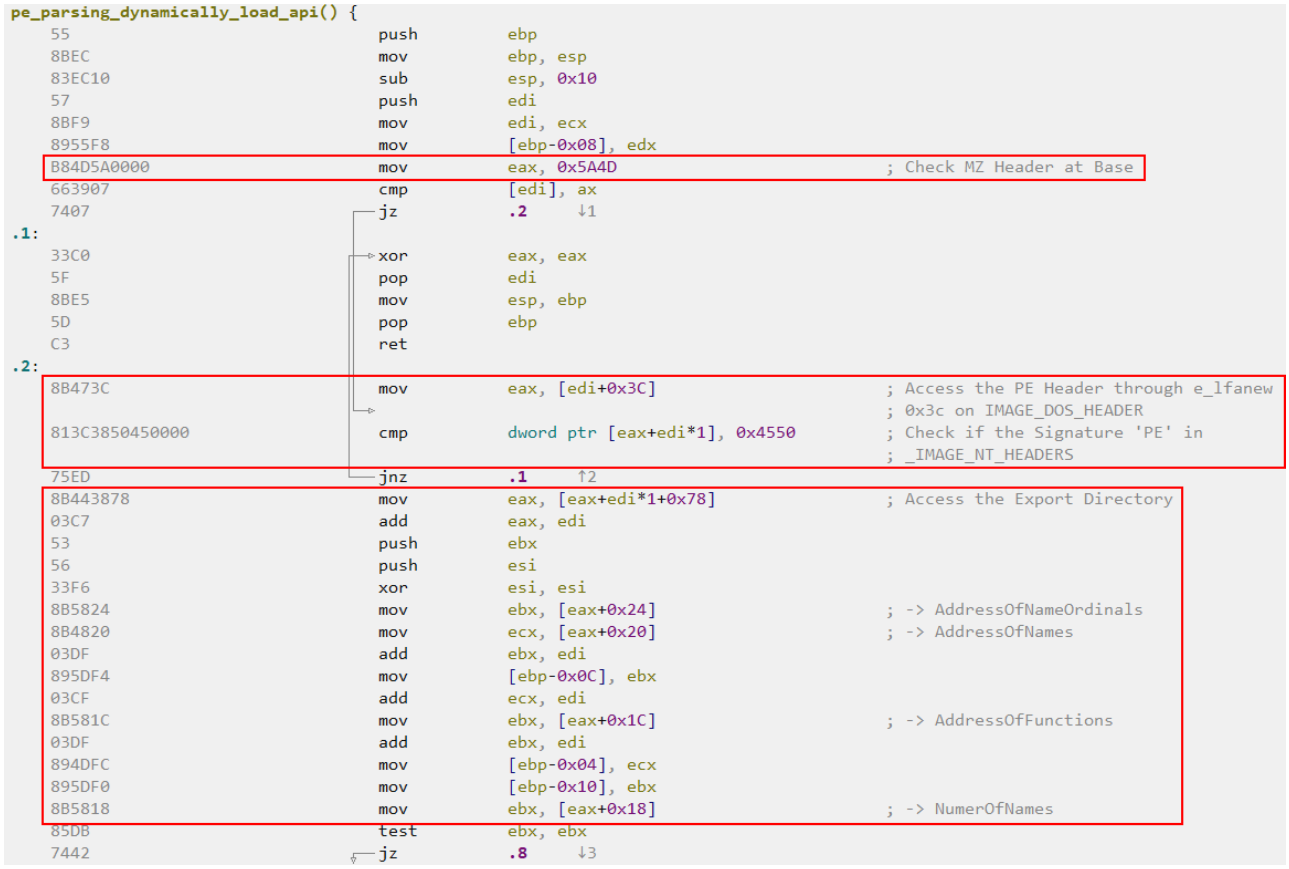
Below, we can see both functions executing in the expected order. First, the strings will be decrypted, then the **pe_parsing_dynamically_load_api** function will parse the **ntdll.dll** module to load the LdrLoadDll API. Finally, the DLL whose name was decrypted will be loaded, followed by the decrypted API being loaded via the LdrGetProcedureAddress API.



By analyzing the code for the **decrypt_str_api function**, we are able to identify the single-byte key **0x19**, used to decrypt the strings.

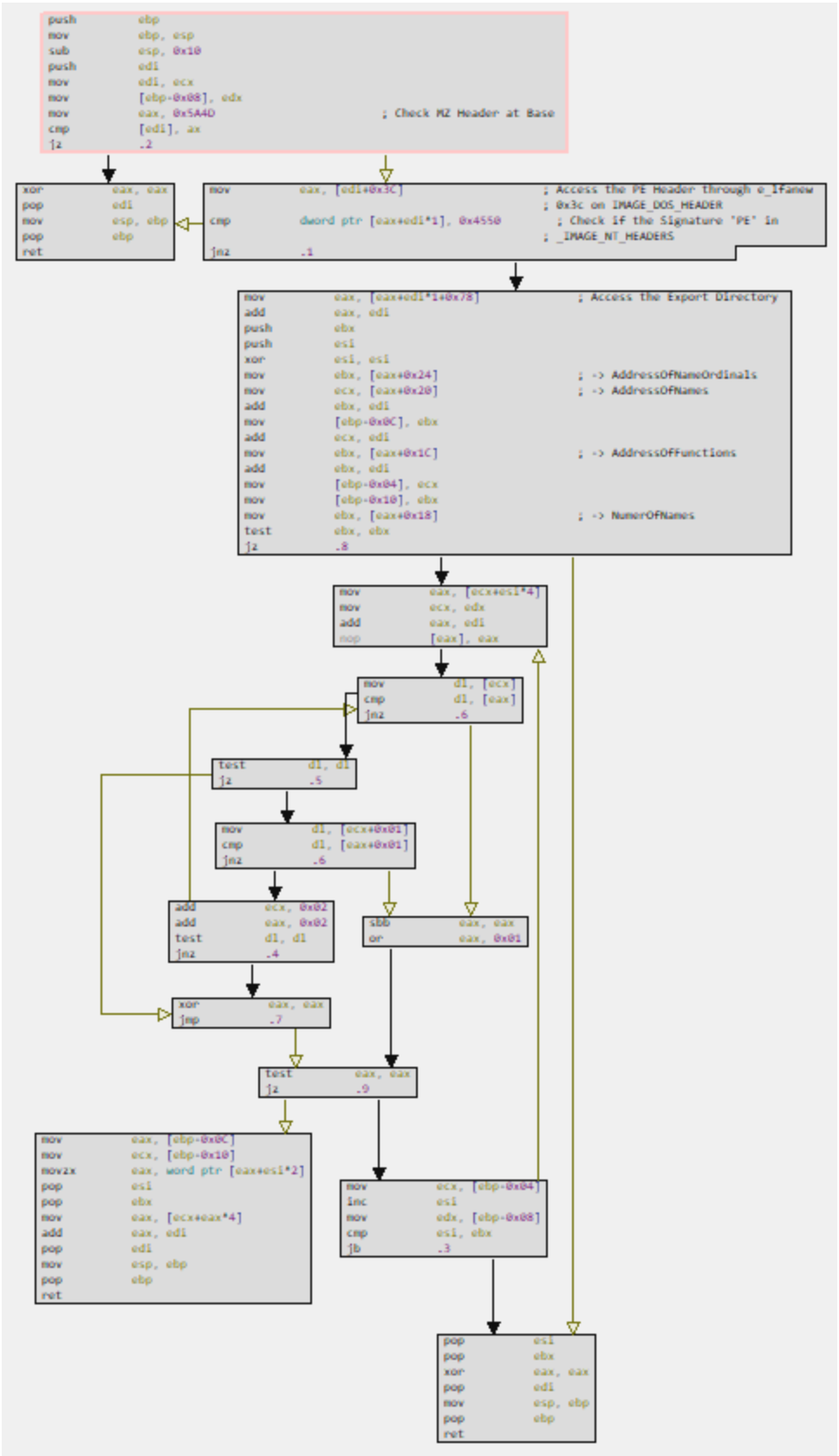


Also when analyzing the **pe_parsing_dynamically_load_api** function, we validate our hypothesis, which establishes that this function is responsible for parsing the DLL that will have a certain API called dynamically.



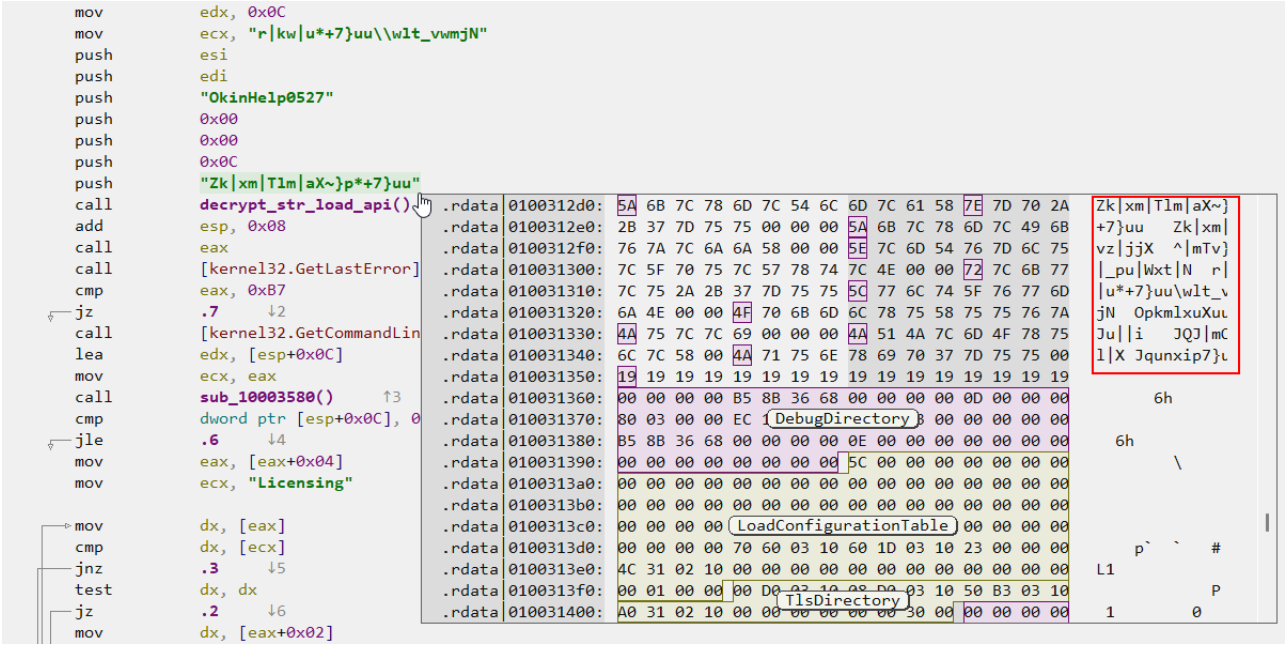
```
pe_parsing_dynamically_load_api() {
    55                          push        ebp
    8BEC                        mov         ebp, esp
    83EC10                      sub         esp, 0x10
    57                          push        edi
    8BF9                        mov         edi, ecx
    8955F8                      mov         [ebp-0x08], edx
    B84D5A0000                  mov         eax, 0x5A4D                 ; Check MZ Header at Base
    663907                      cmp         [edi], ax
    7407                        jz          .2      ↓1
.1:
    33C0                        xor         eax, eax
    5F                          pop         edi
    8BE5                        mov         esp, ebp
    5D                          pop         ebp
    C3                          ret
.2:
    8B473C                      mov         eax, [edi+0x3C]             ; Access the PE Header through e_lfanew
                                                                        ; 0x3c on IMAGE_DOS_HEADER
    813C3850450000              cmp         dword ptr [eax+edi*1], 0x4550 ; Check if the Signature 'PE' in
                                                                        ; _IMAGE_NT_HEADERS
    75ED                        jnz         .1      ↑2
    8B443878                    mov         eax, [eax+edi*1+0x78]       ; Access the Export Directory
    03C7                        add         eax, edi
    53                          push        ebx
    56                          push        esi
    33F6                        xor         esi, esi
    8B5824                      mov         ebx, [eax+0x24]             ; -> AddressOfNameOrdinals
    8B4820                      mov         ecx, [eax+0x20]             ; -> AddressOfNames
    03DF                        add         ebx, edi
    895DF4                      mov         [ebp-0x0C], ebx
    03CF                        add         ecx, edi
    8B581C                      mov         ebx, [eax+0x1C]             ; -> AddressOfFunctions
    03DF                        add         ebx, edi
    894DFC                      mov         [ebp-0x04], ecx
    895DF0                      mov         [ebp-0x10], ebx
    8B5818                      mov         ebx, [eax+0x18]             ; -> NumerOfNames
    85DB                        test        ebx, ebx
    7442                        jz          .8      ↓3
```

In the image below, you can observe the entire flow of the **pe_parsing_dynamically_load_api** function.

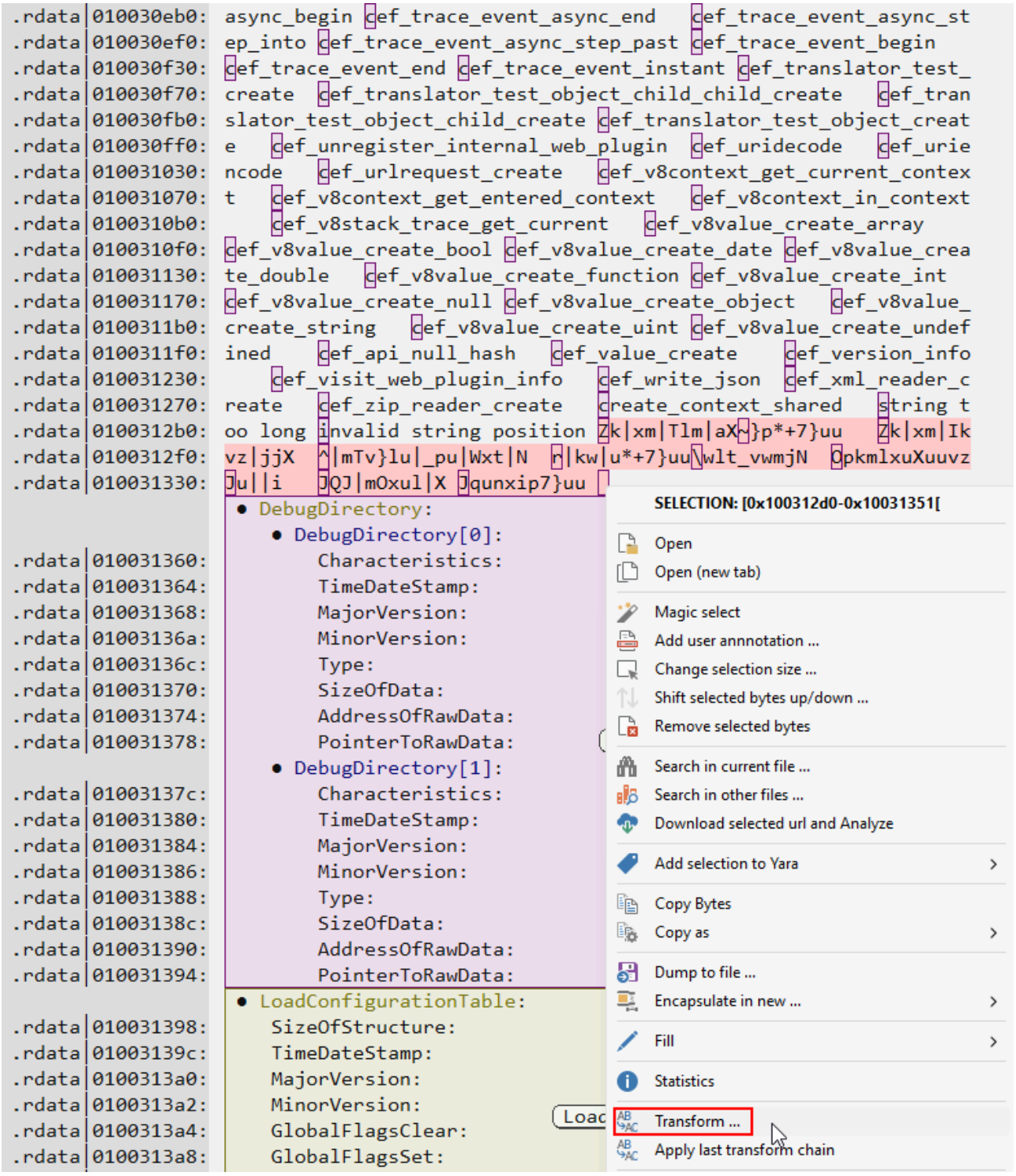


Malcat allows us to analyze the binary from several aspects, through *Disassembly*, *Decompiler*, *Hexadecimal Editor*, and also the way in which the data is structured in such a way that it allows us to

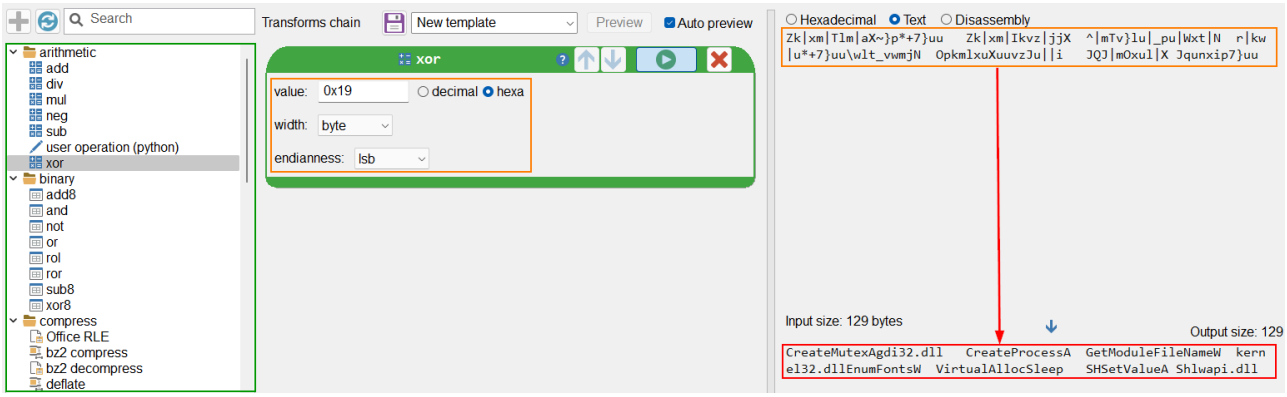
identify code, sections, data, etc. Below, we are able to observe that the offset of the selected encrypted string is found together with all other encrypted strings.



In the following image, we can validate this hypothesis by observing all the encrypted strings in an orderly fashion. Malcat has a feature called **Transform**, which allows us to transform selected data. Knowing that these strings are encrypted using a simple XOR algorithm, with byte **0x19** as the key, we can decrypt them within the Malcat instance itself.



As you can see below, the visuals are reminiscent of *CyberChef*, with the transformation algorithms on the left, the algorithm currently being used with your current configuration in the middle, the original data on the top right, and the transformed data on the bottom. In our case, you can see the names of the APIs/DLLs that will be loaded by the functions described above.



When we perform encryption in Malcat's Transformer, we can apply the decrypted data, which will be available in the *Disassembly*/*Decompiler*, as we can see below.

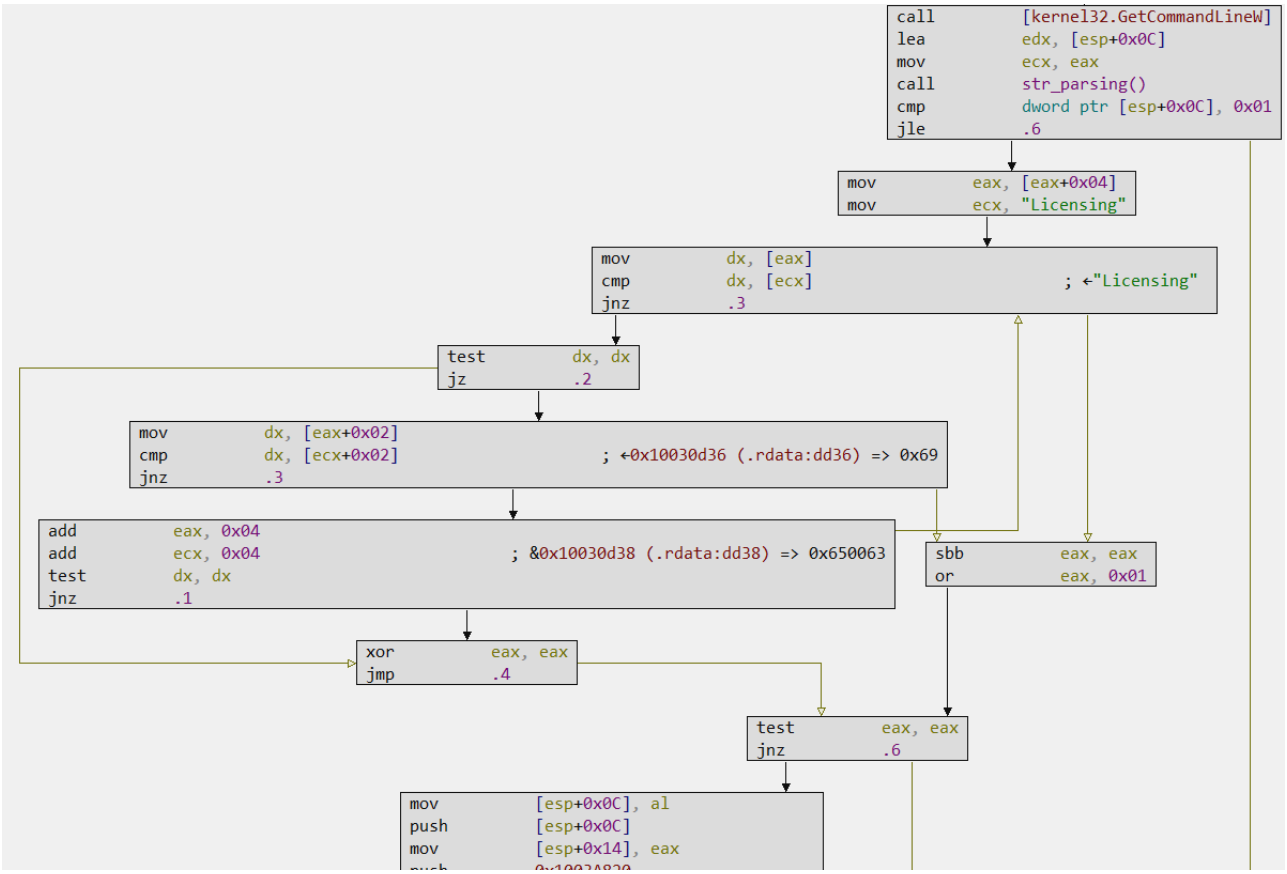
```
push        ebp
mov         ebp, esp
and         esp, 0xFFFFFFF8
sub         esp, 0x20
mov         edx, 0x0C
mov         ecx, "kernel32.dllEnumFontsW"
push        esi
push        edi
push        "OkinHelp0527"
push        0x00
push        0x00
push        0x0C
push        "CreateMutexAgdi32.dll"
call        decrypt_str_load_api()
add         esp, 0x08
call        eax                                 ; Call to CreateMutex
call        [kernel32.GetLastError]
cmp         eax, 0xB7                           ; 0xB7 -> ERROR_ALREADY_EXISTS
jz          .7
```

Argument Checking

Claimloader expects to receive an argument, which will be identified and parsed in the code block below by collecting the command line using GetCommandLineW, followed by parsing the collected string, which refers to the current process's command line. Depending on whether the current process's command line is identified as having an argument (**cmp dword ptr [esp+0x0C], 0x01**) or not, *Claimloader's* code will take two completely opposite directions.

```
call        [kernel32.GetCommandLineW]
lea         edx, [esp+0x0C]
mov         ecx, eax
call        str_parsing()
cmp         dword ptr [esp+0x0C], 0x01
jle         .6
```

In the following image, we can see that *Claimloader* expects that, if an argument is identified, it will be "**Licensing**".



If the correct argument is detected, *Claimloader* will follow its code flow for decryption, injection, and *Publoader* execution. If the argument is not detected, *Claimloader* will create two types of persistence, one of which is through registry keys. This is the flow we will analyze next.

Creating Multiple Persistencies in the Infected System

If no argument (or an incorrect argument) is identified when executing the decoy executable, *Claimloader* will execute the code block below.

```
call    cef_get_extensions_for_mime_type_display()      ; Copies the decoy executable and Claimloader DLL to a masked Adobe directory
push    0x4C = 'L'
push    "C:\\ProgramData\\AdobeLicensingPlu..e_licensing_helper.exe Licensing"
push    0x01
push    "EpsonPrint"
push    autorun                         ; "Software\\Microsoft\\Windows\\CurrentVersion\\Run" [T1547.001]
push    0x80000001 = HKEY_CURRENT_USER
push    0x0B
push    "SHSetValueA"
mov     edx, 0x0B
mov     ecx, "Shlwapi.dll"
call    decrypt_str_load_api()
add     esp, 0x08
call    eax
call    cef_stream_writer_create_for_Error()      ; Create another persistence to execute the Claimloader
pop     edi
xor     eax, eax
pop     esi
mov     esp, ebp
pop     ebp
ret
```

As you can see in the macro view, *Claimloader* will copy the decoy and *Claimloader* to a fake *Adobe* directory, followed by implementing the T1547.001 persistence technique, which consists of adding a program to the **Software\Microsoft\Windows\CurrentVersion\Run** registry key, allowing the decoy to run correctly (with the '**Licensing**' argument) whenever the system restarts, followed by creating another persistence, which we will analyze in detail later.

First, let's look at how Claimloader implements the copy of Decoy and Claimloader, in the **cef_get_extensions_for_mime_type_display** function.

In the image below, we can see the creation of the **C:\ProgramData\AdobeLicensingPlugin** directory through the CreateDirectoryW API.

```
push        ebp
mov         ebp, esp
and         esp, 0xFFFFFFF8
sub         esp, 0x24C
mov         eax, [#SecurityCookie]
xor         eax, esp
mov         [esp+0x248], eax
push        esi
push        0x23 = '#'
xor         eax, eax
mov         dword ptr [esp+0x34], 0x07
push        "C:\\ProgramData\\AdobeLicensingPlugin"        ; Create this masked directory, for persistence
lea         ecx, [esp+0x24]
mov         dword ptr [esp+0x34], 0x00
mov         [esp+0x24], ax
call        sub_10004890()
push        0x104
lea         eax, [esp+0x44]
mov         edx, 0x0C
push        eax
push        0x00
push        0x12
push        "GetModuleFileNameW"
mov         ecx, "kernel32.dllEnumFontsW"
call        decrypt_str_load_api()
add         esp, 0x08
call        eax
cmp         dword ptr [esp+0x30], 0x08
lea         eax, [esp+0x1C]
push        0x00
cmovnb      eax, [esp+0x20]
push        eax
call        [kernel32.CreateDirectoryW]
test        eax, eax
jnz         .2

call        [kernel32.GetLastError]
cmp         eax, 0xB7
jz          .2
```

Next, the Claimloader dynamically loads the MoveFileW API using stack strings, which will be moved to the **ECX** register to be used as the argument (**lpProcName**) of the GetProcAddress API. After loading, the API is called indirectly through the offset pointer stored in the **EAX** register. At the end of this block, the decoy will be moved and renamed to the full path "**C:\ProgramData\AdobeLicensingPlugin\WF_Adobe_licensing_helper.exe**".

The same technique is implemented to move the Claimloader to the same directory, with its full path as "**C:\ProgramData\AdobeLicensingPlugin\NewUI.dll**".

After these code blocks are executed correctly, the function will return to the main function, where *Claimloader* will create the persistence for the **\Run** registry key, explained previously, with the following command as an argument:

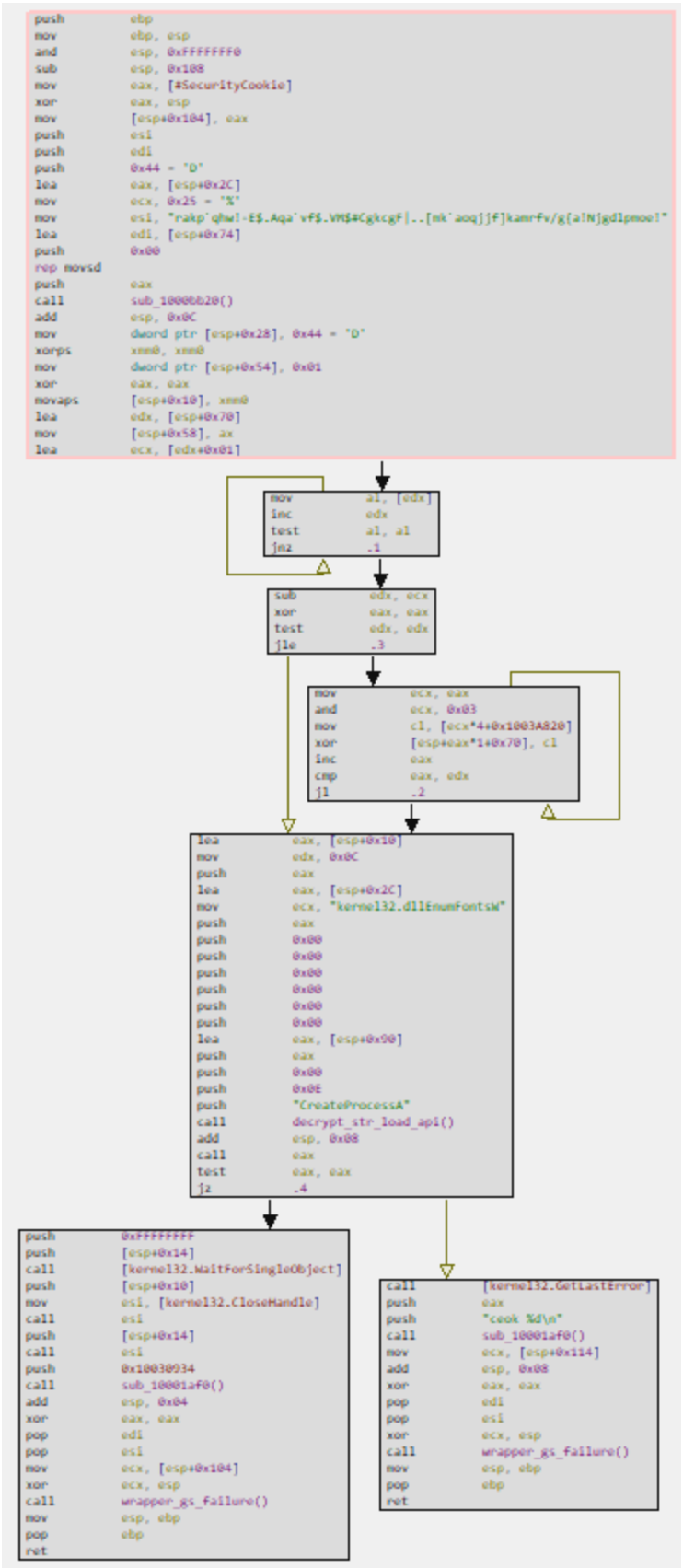
- **C:\ProgramData\AdobeLicensingPlugin\WF_Adobe_licensing_helper.exe Licensing**

Therefore, upon device restart, Claimloader will execute correctly with the expected argument.

Now let's analyze the **cef_stream_writer_create_for_Error** function, which runs after the T1547.001 persistence implementation. This function creates another persistence so that *Claimloader* can run effectively. It also increases the level of investigation for DFIR analysts, who can stop looking for possible persistence techniques if they find just one implemented by *Claimloader*.

Below, we can see a simple and direct code flow in this function, in which the decryption of a large string will be performed, using a different algorithm than the one we analyzed previously, followed by the creation of a

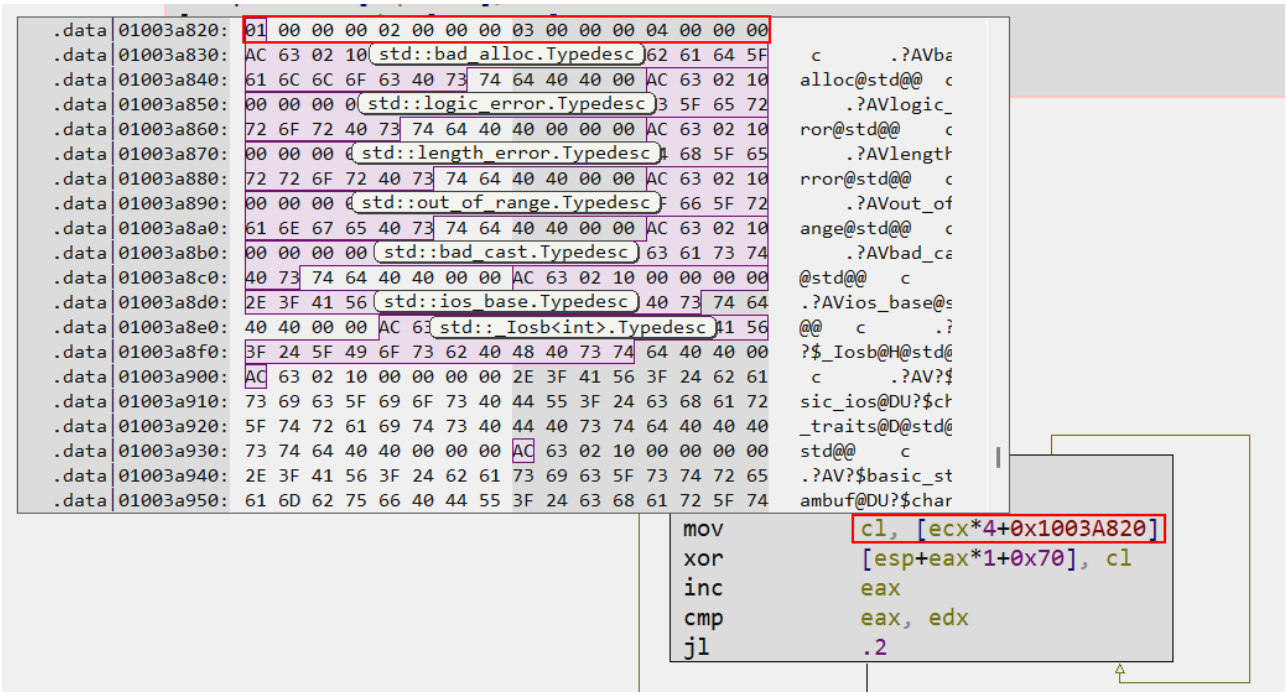
process at the end of this function.



First, let's analyze another custom string decryption algorithm. As you can see in the image below, the string is much longer than the other strings we identified previously. Furthermore, the decryption key is not explicit and is not a single-byte XOR key, as we identified in the previous algorithm.

If we take a closer look at the algorithm in the image below, we'll notice that the instruction and **ECX**, **0x03** is equivalent to **ECX & 3**, meaning it keeps only the two least significant bits of **ECX**, which contain the contents of **EAX**. In the following instruction, the *Claimloader* finds the exact address of the *XOR key*, which is different for each XORed byte in each loop.

The XOR key is composed of a 4-byte array, respectively *0x01*, *0x02*, *0x03*, *0x04*, as we can see in the image below. Therefore, the loop goes through each encrypted byte of this array, returning to the beginning of the array when it reaches byte *0x04*.



I developed a Python script to demonstrate what I explained above in a practical way. As you can see in the sequence of images below, each encrypted byte has a key that corresponds to a byte in the *4-byte* array illustrated in the image above.

```
PS C:\Users\0x0d4y\Desktop\Research\Malware-Research\China-Nexus\Mustang Panda\Voice for the Voiceless photos> python.exe .\custom_dec_str.py
Key: 0x1
Encrypted String: 0x72
String Block Decrypted: s
--------------------------------------------------------
Key: 0x2
Encrypted String: 0x61
String Block Decrypted: sc
--------------------------------------------------------
Key: 0x3
Encrypted String: 0x6b
String Block Decrypted: sch
--------------------------------------------------------
Key: 0x4
Encrypted String: 0x70
String Block Decrypted: scht
--------------------------------------------------------
Key: 0x1
Encrypted String: 0x60
String Block Decrypted: schta
--------------------------------------------------------
Key: 0x2
Encrypted String: 0x71
String Block Decrypted: schtas
--------------------------------------------------------
Key: 0x3
Encrypted String: 0x68
String Block Decrypted: schtask
--------------------------------------------------------
Key: 0x4
Encrypted String: 0x77
String Block Decrypted: schtasks
--------------------------------------------------------
Key: 0x1
Encrypted String: 0x21
String Block Decrypted: schtasks
```

```
Key: 0x2
Encrypted String: 0x2d
String Block Decrypted: schtasks /
-------------------------------------------------------
Key: 0x3
Encrypted String: 0x45
String Block Decrypted: schtasks /F
-------------------------------------------------------
Key: 0x4
Encrypted String: 0x24
String Block Decrypted: schtasks /F
-------------------------------------------------------
Key: 0x1
Encrypted String: 0x2e
String Block Decrypted: schtasks /F /
-------------------------------------------------------
Key: 0x2
Encrypted String: 0x41
String Block Decrypted: schtasks /F /C
-------------------------------------------------------
Key: 0x3
Encrypted String: 0x71
String Block Decrypted: schtasks /F /Cr
-------------------------------------------------------
Key: 0x4
Encrypted String: 0x61
String Block Decrypted: schtasks /F /Cre
-------------------------------------------------------
Key: 0x1
Encrypted String: 0x60
String Block Decrypted: schtasks /F /Crea
-------------------------------------------------------
Key: 0x2
Encrypted String: 0x76
String Block Decrypted: schtasks /F /Creat
-------------------------------------------------------
```

At the end of the loop, we have our decrypted string, revealing itself as a command to be executed, implementing another persistence technique described by *MITRE ATT&CK* as T1053.005, through the **Schtasks.exe** binary.

```
PS C:\Users\0x0d4y\Desktop\Research\Malware-Research\China-Nexus\Mustang Panda\Voice for the Voiceless photos> python.exe .\custom_dec_str.py
---------- Encrypted String ----------
b'rakp`qhw!-E$.Aqa`vf$.VM$#CgkcgF|qgqmdl`aLcmefgq&!-PG!ojjtvf$.OL$3",PS"!G;^SvneqelFbp`^B`n`fHhafjrkmcQnvchl_SG]B`n`f[mk`aoqjjf]kamrfv/g{a!Njgdlpmoe!'
---------- Decrypted String ----------
schtasks /F /Create /TN "AdobeExperienceManager" /SC minute /MO 2 /TR "C:\ProgramData\AdobeLicensingPlugin\WF_Adobe_licensing_helper.exe Licensing"
```

Below is the structural explanation of the persistence configuration.

schtasks → Windows binary to create, delete or manage scheduled tasks.

/F → Forces the creation of the task, overwriting if it already exists.

/Create → Indicates that a new task will be created.

/TN "AdobeExperienceManager" → Name of the scheduled task (in this case, "**AdobeExperienceManager**").

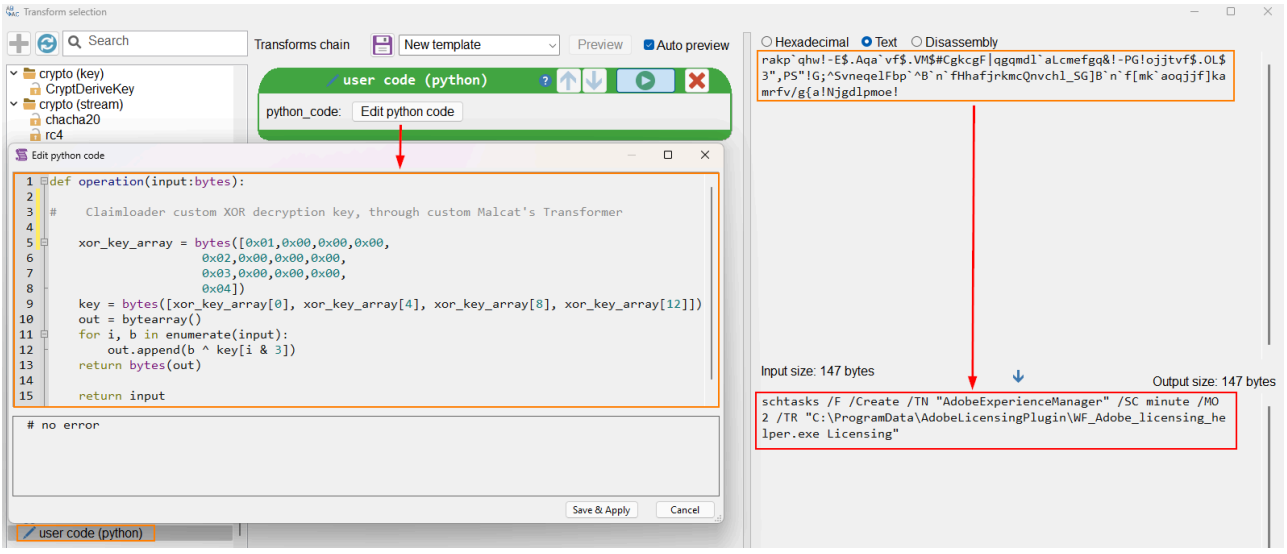
/SC minute → Sets the execution frequency: every minute.

/MO 2 → Schedule modifier: Run every *2 minutes*.

/TR "C:\ProgramData\AdobeLicensingPlugin\WF_Adobe_licensing_helper.exe Licensing" → Path to the executable that will be triggered by the task, with the "*Licensing*" argument.

As we saw in the analysis of the first string decryption algorithm earlier, Malcat allows us to perform such actions through the built-in Transform, with many similarities to CyberChef. However, the XOR algorithm described above is an algorithm with custom logic, despite being just a simple XOR.

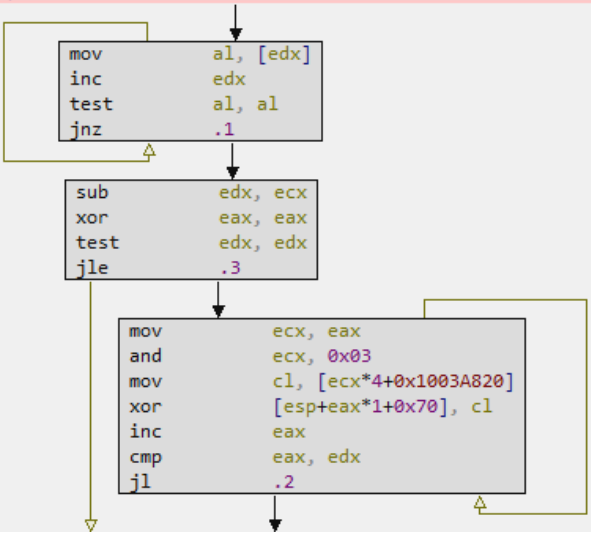
Despite this customization of the XOR algorithm for the persistence implementation command encryption, Malcat allows us to implement a custom Transformer in Python. This custom Transformer allows us to perform the same actions as the built-in Transformers, such as modifying the encrypted string in the code for better analysis.



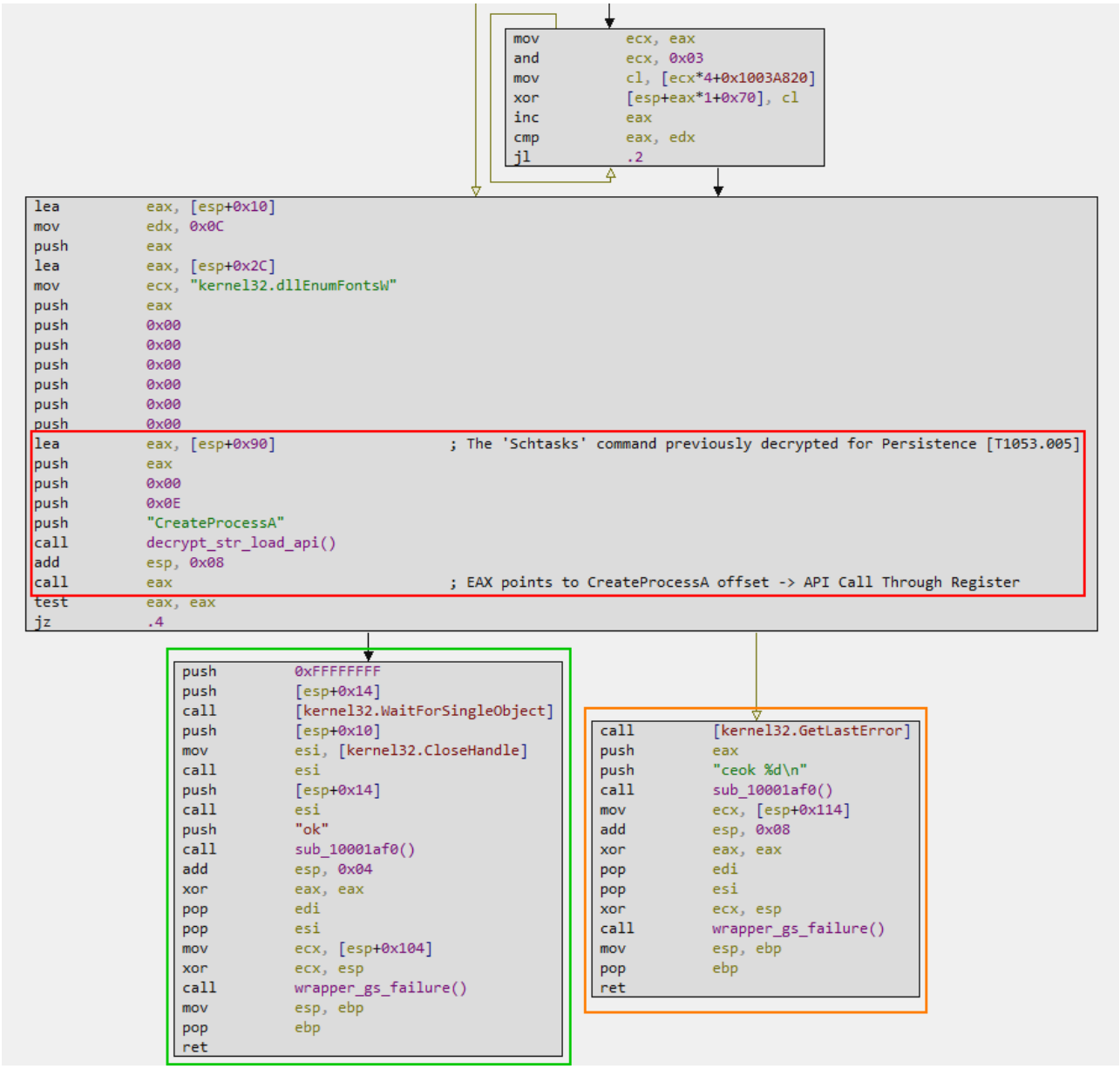
In the following image, we can see the command in plain text and decrypted, making the final functionality of this function clearer.

```
push        ebp
mov         ebp, esp
and         esp, 0xFFFFFFF0
sub         esp, 0x108
mov         eax, [#SecurityCookie]
xor         eax, esp
mov         [esp+0x104], eax
push        esi
push        edi
push        0x44 = 'D'
lea         eax, [esp+0x2C]
mov         ecx, 0x25 = '%'
mov         esi, "schtasks /F /Create /TN \"AdobeEx..._licensing_helper.exe Licensing\""
lea         edi, [esp+0x74]
push        0x00
rep movsd
push        eax
call        string_formatting()
add         esp, 0x0C
mov         dword ptr [esp+0x28], 0x44 = 'D'
xorps       xmm0, xmm0
mov         dword ptr [esp+0x54], 0x01
xor         eax, eax
movaps      [esp+0x10], xmm0
lea         edx, [esp+0x70]
mov         [esp+0x58], ax
lea         ecx, [edx+0x01]
```

String Decrypted and Transformed in Malcat's Proximity View



When decrypting the command to be executed, for the implementation of another persistence, the command is passed as an argument to the CreateProcessA API, which will be responsible for finally executing the command.

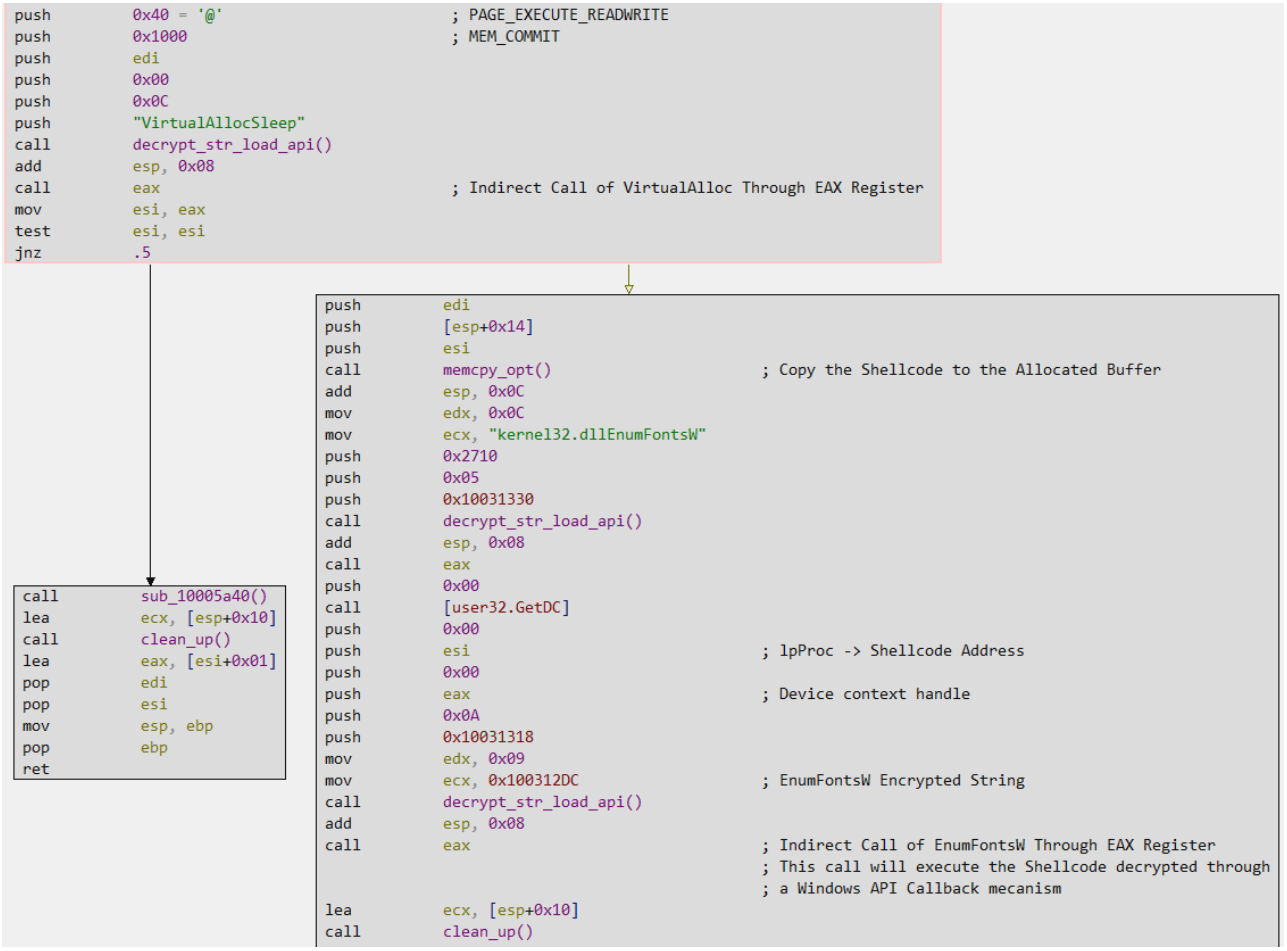


Therefore, we can see that this entire *Claimloader* execution flow is the victim's first execution, and the victim clearly doesn't know they're running malware, much less that an argument is required for it to function properly. Thus, when the user executes the decoy normally, *Claimloader* will execute this entire flow, aiming to allow *Claimloader* to execute correctly within the next two minutes.

Shellcode Extraction

To extract the Shellcode that will be decrypted, we can use the dynamic analysis methodology through *x32dbg*, by understanding where we should be to extract it correctly.

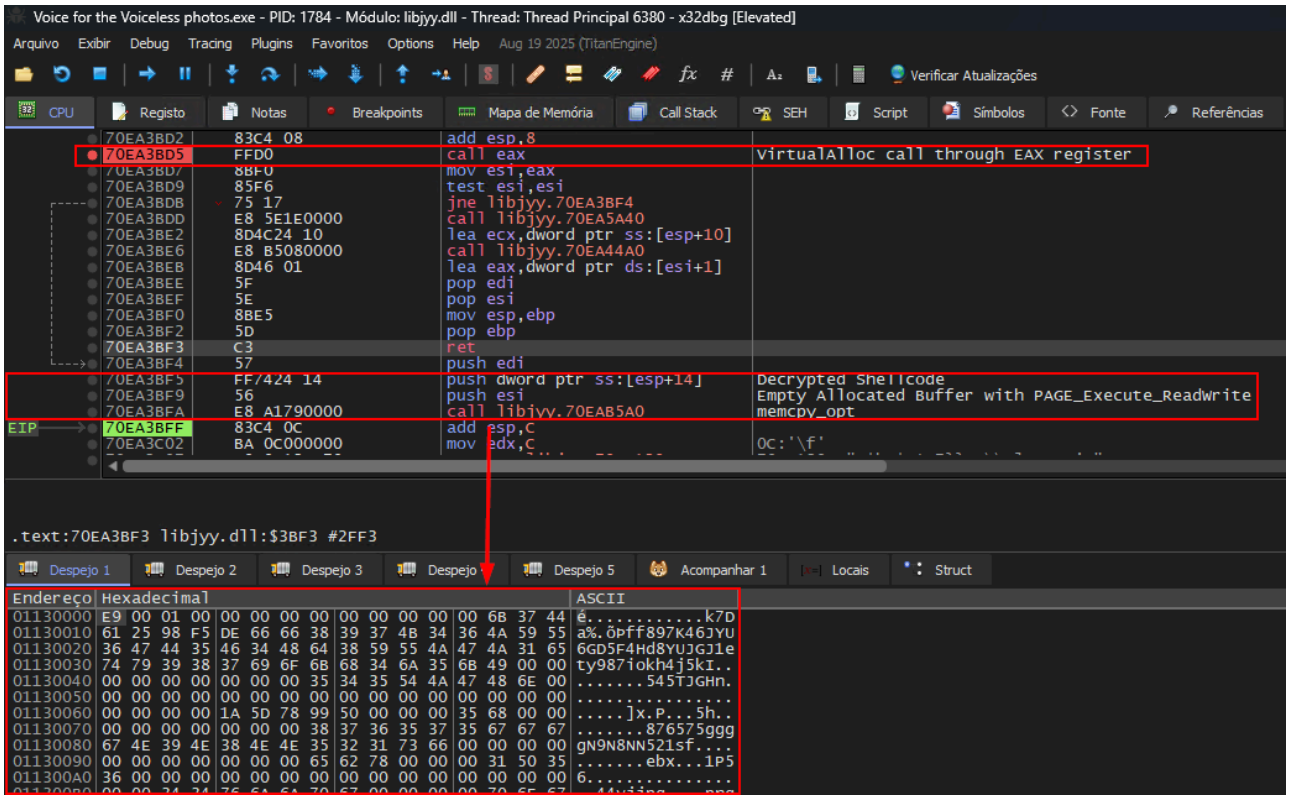
Below, we can see the other flow that will only be executed if the decoy is executed containing the '*Licensing*' argument.



In the code block above, we can see that two strings referring to APIs are decrypted and dynamically loaded by the previously analyzed function (*decrypt_str_load_api*). VirtualAlloc is an extremely common function for allocating buffers for shellcode, and as can be seen in the flow above, the offset of the buffer allocated with execution permission (**PAGE_EXECUTE_READWRITE**) is copied to the *ESI* register, which is used as an argument to a custom function that replicates memcpy in an optimized way, also having as an argument the offset to the buffer with the already decrypted shellcode, but in a memory space without execution permission.

After copying the shellcode to the previously allocated buffer, in the image above we can also see that the offset of this buffer will be used as an argument for the EnumFontsW function call, which, when called, will execute the shellcode by abusing the API's Callback mechanism. The **lpProc** argument of *EnumFontsW* expects to receive a pointer to the application-defined callback function. Therefore, by passing the offset of a shellcode to this argument, *EnumFontsW* will execute the shellcode.

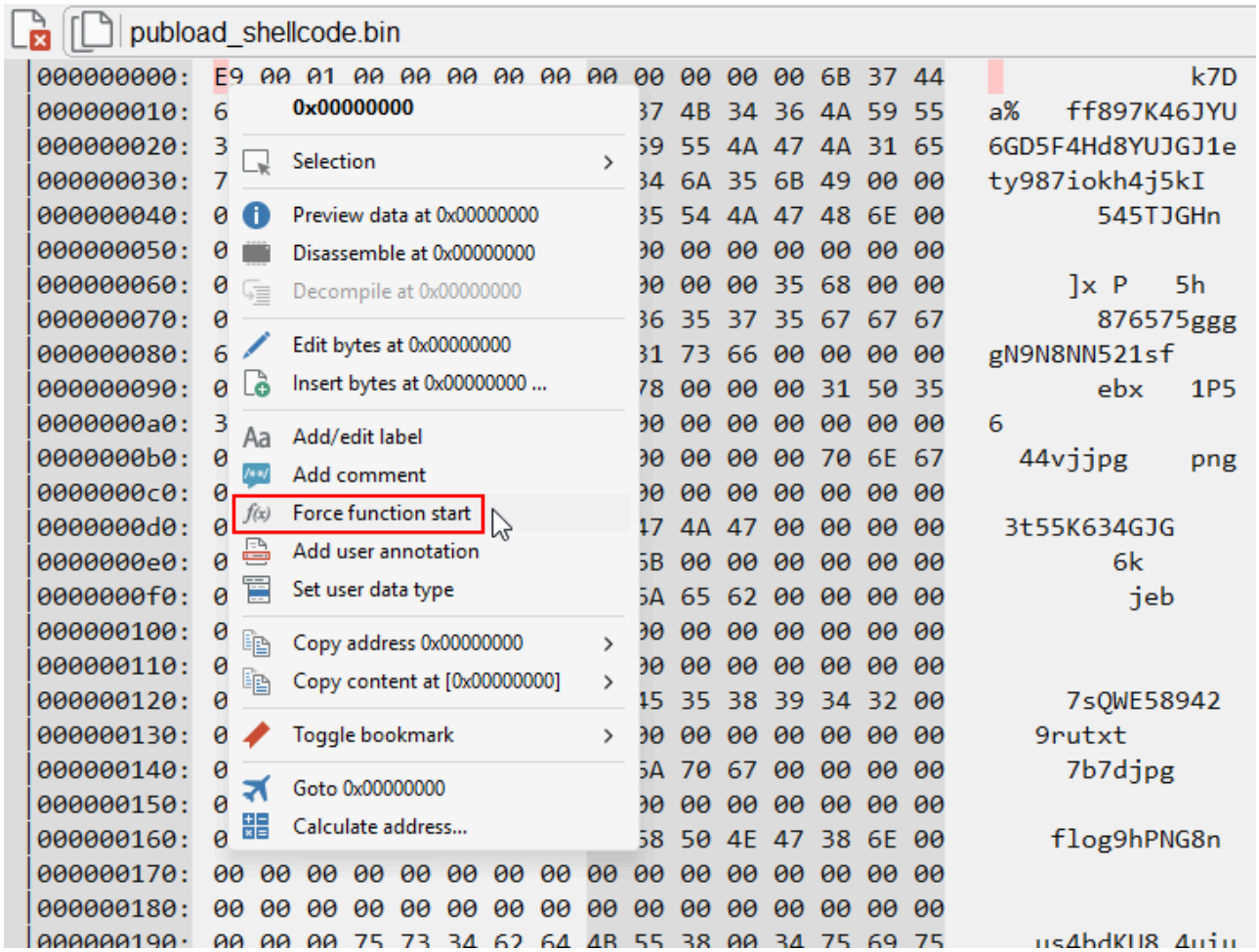
In the image below, we can see the shellcode being copied to the buffer with execution permission. This allows us to extract the complete shellcode code for static analysis.



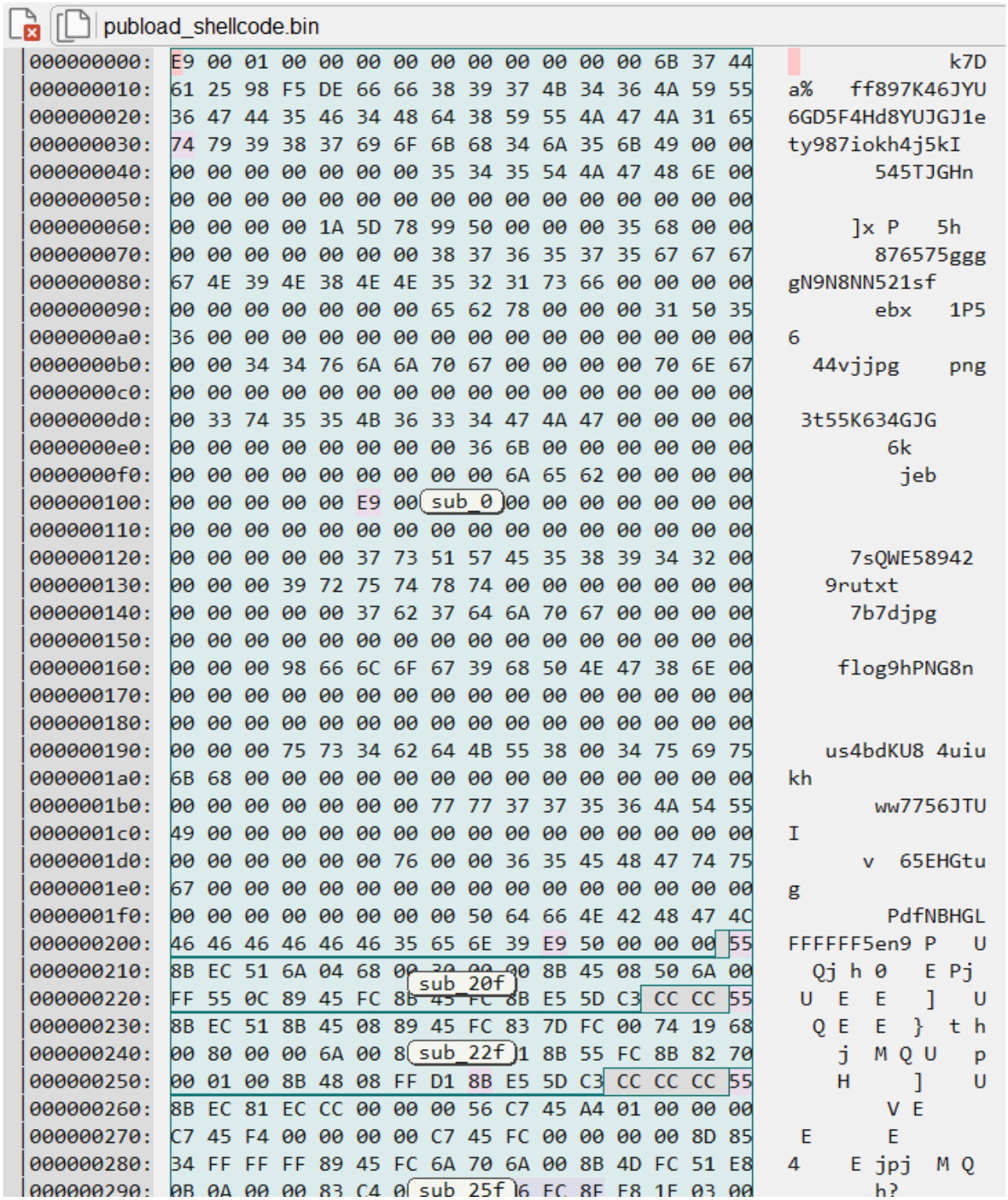
Extracted Shellcode Functionality Identification

Once the shellcode has been correctly extracted from memory, we can use Malcat to perform further triage and analysis. When we first upload the *shellcode*, Malcat doesn't know how to process it, just like other software (such as *Binary Ninja*, *IDA Pro*, etc.).

But, to force Malcat to structure it, we just need to go to the Hexadecimal editor tab, and force Malcat to start a function in the first byte of the file (**0xE9** is a classic opcode for **JMP**).



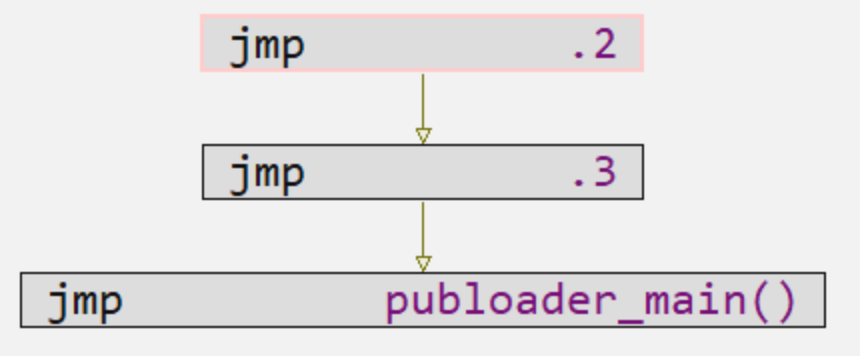
By doing this, we can notice the difference in the way Malcat understands the file, now in a fully structured way.



If we run CAPA to perform a Triage on the file, we'll observe the detection of a function that resolves hashes. This is crucial because, since this is shellcode, the developers likely implemented *API Hashing* to dynamically load APIs in an obfuscated manner. Therefore, this finding makes perfect sense given what we have at hand.

```
pubload_shellcode.bin
                                        CAPA script

CAPA framework by mandiant (https://github.com/mandiant/capa) using malcat for analysis.
Everything except the malcat comes from the github repository.
Mandiant rules can be found in <analysis DATA DIR>/scripts/capa/rules.zip.
You can add your own as .yml files in <USER DATA DIR>/scripts/capa/all_rules/*.yml.


                                      ATT&CK information
+------------------------+-------------------------------------------------------------------------------+
| ATT&CK Tactic          | ATT&CK Technique                                                              |
|------------------------+-------------------------------------------------------------------------------|
| DEFENSE EVASION        | Obfuscated Files or Information::Indicator Removal from Tools [T1027.005]     |
+------------------------+-------------------------------------------------------------------------------+



                                    11 capabilities found
+-------------------------------------------------------+-------------------------------------------------+
| CAPABILITY                                            | NAMESPACE                                       |
|-------------------------------------------------------+-------------------------------------------------|
| resolve function by hash                              | linking/runtime-linking                         |
+-------------------------------------------------------+-------------------------------------------------+


                            resolve function by hash
namespace:  linking/runtime-linking
att&ck:     Defense Evasion::Obfuscated Files or Information::Indicator Removal from Tools [T1027.005]
rule scope: function
author:
function @ 0000025f
  or:
    number: 0x7C0DFCAA = ROR13(GetProcAddress) @ 000002c8
```

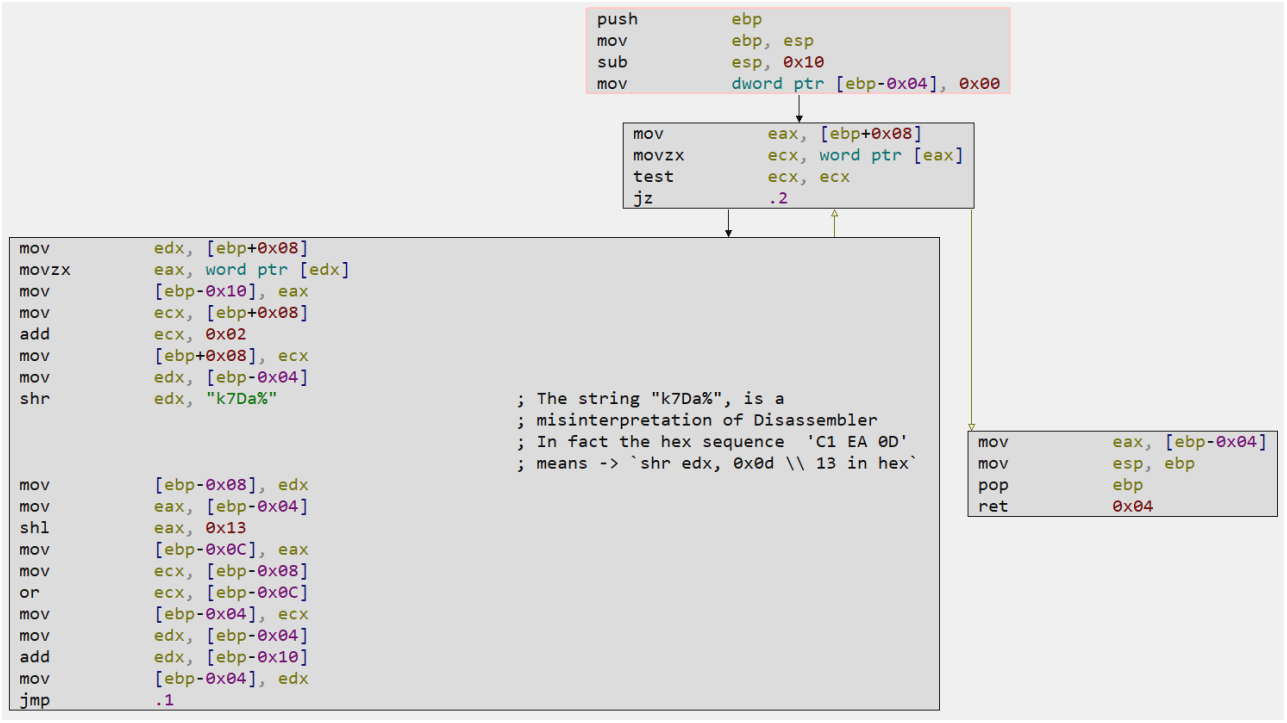
The first function consists of two sequences of **JMPs** followed by a last one that actually points to a function, which consists of the function with the main functionalities of the *Publoader Shellcode*.



The first action of Publoader's Main function is to resolve kernel32.dll in two ways. To ensure the DLL is resolved and finally loaded, it attempts to resolve it using two hashes (**6E2BCA17** and **8FECD63F**), a critical step for the rest of the code. When resolving kernel32.dll through the **publoader_dll_hash_ror13** function, the *GetProcAddress* and LoadLibraryA APIs are also resolved through the **publoader_api_hash_ror13** function.

When analyzing the *publoader_dll_hash_ror13* function, we can observe that Publoader uses the *PEB Walking* technique, where the *PEB* structure is accessed in a loop to collect the name of each module, where each name will be submitted to the *ROR13* hash algorithm, and finally compared if the Hash resulting from the function will match one of the hashes previously identified, which were placed as arguments for this function.

In the image below, we are able to validate that the algorithm used is in fact **ROR13**, through the identification of logical operations, mainly referring to **shr edx, 0x0d**.



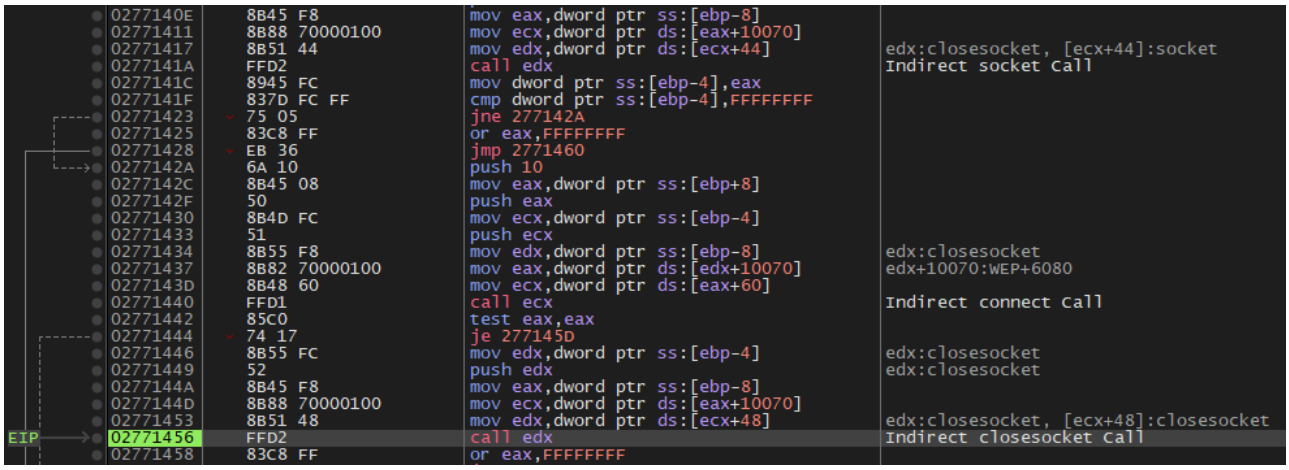
Publoader also implements a long API loading function through API Hashing, as we can see in the image below.

Function: publoader_main_api_hashing() at 0x0000074f

```
    module_str = 0x4b /* 'K' */;
    uStack_4b = 0x65 /* 'e' */;
    uStack_4a = 0x72 /* 'r' */;
    uStack_49 = 0x6e /* 'n' */;
    uStack_48 = 0x65 /* 'e' */;
    uStack_47 = 0x6c /* 'l' */;
    uStack_46 = 0x33 /* '3' */;
    uStack_45 = 0x32 /* '2' */;
    uStack_44 = 0x2e /* '.' */;
    uStack_43 = 100;
    uStack_42 = 0x6c /* 'l' */;
    uStack_41 = 0x6c /* 'l' */;
    uStack_40 = sub_0;
    ptr_module_name = (*(code *)param_1[1])(&module_str);
    if (ptr_module_name != sub_0) {
        uVar1 = publoader_api_hash_ror13(*param_1, ptr_module_name, 0x91afca54 /* hash(VirtualAlloc) */);
        param_1[4] = uVar1;
        uVar1 = publoader_api_hash_ror13(*param_1, ptr_module_name, 0x7946c61b /* hash(VirtualProtect) */);
        param_1[3] = uVar1;
        uVar1 = publoader_api_hash_ror13(*param_1, ptr_module_name, 0x30633ac /* hash(VirtualFree) */);
        param_1[2] = uVar1;
        uVar1 = publoader_api_hash_ror13(*param_1, ptr_module_name, 0xdb2d49b0 /* hash(Sleep) */);
        param_1[6] = uVar1;
        uVar1 = publoader_api_hash_ror13(*param_1, ptr_module_name, 0x96a4228f /* hash(GetComputerNameA) */);
        param_1[7] = uVar1;
        uVar1 = publoader_api_hash_ror13(*param_1, ptr_module_name, 0x8ab241a0 /* hash(GetVolumeInformationA) */);
        param_1[9] = uVar1;
        uVar1 = publoader_api_hash_ror13(*param_1, ptr_module_name, 0xf791fb23 /* hash(GetTickCount) */);
        param_1[10] = uVar1;
    }
    module_str = 0x75 /* 'u' */;
    uStack_4b = 0x73 /* 's' */;
    uStack_4a = 0x65 /* 'e' */;
    uStack_49 = 0x72 /* 'r' */;
    uStack_48 = 0x33 /* '3' */;
    uStack_47 = 0x32 /* '2' */;
    uStack_46 = 0x2e /* '.' */;
    uStack_45 = 100;
    uStack_44 = 0x6c /* 'l' */;
    uStack_43 = 0x6c /* 'l' */;
    uStack_42 = sub_0;
    ptr_module_name = (*(code *)param_1[1])(&module_str);
    if (ptr_module_name != sub_0) {
        uVar1 = publoader_api_hash_ror13(*param_1, ptr_module_name, 0xbc4da2a8 /* hash(MessageBoxA) */);
```

And finally, after loading all the necessary DLLs, the Shellcode executes them indirectly, with the aim of collecting information from the device and sending it to the command and control server, as we can see in the following sequence of images in *x32dbg*.

```
          02771270    8BEC            mov ebp,esp
          02771272    83EC 0C         sub esp,C
          02771275    56              push esi
          02771276    894D F4         mov dword ptr ss:[ebp-C],ecx        ecx:GetComputerNameA
          02771279    C745 F8 FF010000 mov dword ptr ss:[ebp-8],1FF
          02771280    8D45 F8         lea eax,dword ptr ss:[ebp-8]
          02771283    50              push eax
          02771284    8B4D 08         mov ecx,dword ptr ss:[ebp+8]        ecx:GetComputerNameA
          02771287    51              push ecx                            ecx:GetComputerNameA
          02771288    8B55 F4         mov edx,dword ptr ss:[ebp-C]
          0277128B    8B82 70000100   mov eax,dword ptr ds:[edx+10070]
          02771291    8B48 1C         mov ecx,dword ptr ds:[eax+1C]       ecx:GetComputerNameA, [eax+1C]:GetComputerNameA
EIP ----> 02771294    FFD1            call ecx                            ecx:GetComputerNameA
```



Conclusion

This research was a lot of fun to produce, as it's very interesting to see that the China-Nexus APTs share the same TTPs, allowing us to identify operational and tactical similarities in their campaigns, as observed in the Veletrix Loader campaign I analyzed. I'm considering writing a post focused on forensics and threat hunting, regarding the execution of this campaign.

Below you can find Yara rules and Python script for String Decryption analyzed in this post, in my Github repository.

Until next time, I hope you, reader, enjoyed this long read!