

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.

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

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

Mode                LastWriteTime         Length Name
----                -
-a-rhs-            27/05/2025   17:43         244224 libjyy.dll
-a-----            27/05/2025   17:43        175328 Voice for the Voiceless photos.exe
```

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 libjyy.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:

79bf3258e03fd1acb395dc184f5496dfa4b3d6a3f9f4598c5df13422cc600d

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:

fhhbjyy

LegalCopyright:

合肥诺拉网络科技有限公司, 保留所有权利。

OriginalFilename:

fhhbjyy.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\fhbjyy.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.

622 symbols

Expand all

libjyy.dll

Name	Address	XRefs
LABELS (0 total)		
IMPORTS (0 total)		
EXPORTS (0 total)		
ENTRIES (0 total)		
FUNCTIONS (0 total)		
DATA (0 total)		
TYPEDEFS (0 total)		
INTERNAL (0 total)		

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**).

Σ Functions (110 hits)				
Address	Level	Confidence	F	Identification
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.

Check hash of file against online threat intelligence services			Intelligence Lookup	☐ Hide unknown
Intelligence source	Level	Signature		
▼ Malshare [NOT FOUND]				
▼ MalwareBazaar [NOT FOUND]				
▼ OPSWAT MetaDefender [DEACTIVATED]				
▼ ⚠️ Triage [TIMEOUT]				
▼ VirusExchange [NOT FOUND]				
▼ ✓ VirusShare				
be4ed0d3aa0b2573927a046620106b13	CLEAN	Added on 2023-09-02T21:16:36+00:00		
▼ ✓ VirusTotal				
Avast-Mobile				
BitDefenderFalx				
SymantecMobileInsight				
Trustlook				
ALYac	CLEAN			
APEX	CLEAN			
AVG	CLEAN			
Acronis	CLEAN			
AhnLab-V3	CLEAN			
Alibaba	CLEAN			
Antiy-AVL	CLEAN			
Arcabit	CLEAN			
Avast	CLEAN			
Avira	CLEAN			
Baidu	CLEAN			
BitDefender	CLEAN			
Bkav	CLEAN			
CAT-QuickHeal	CLEAN			
CMC	CLEAN			
CTX	CLEAN			
ClamAV	CLEAN			
CrowdStrike	CLEAN			
Cylance	CLEAN			
Cynet	CLEAN			
DeeplInstinct	CLEAN			
DrWeb	CLEAN			
ESET-NOD32	CLEAN			
Elastic	CLEAN			
Emsisoft	CLEAN			
F-Secure	CLEAN			
Fortinet	CLEAN			
GDData	CLEAN			

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**.

"ProcessMain"

0x0040e1c4

☐ keep

☒ Little Endian
☐ Big Endian

☒ Hexadecimal
☐ Decimal

[-] Strings

Ascii character	P
UTF16-le character	隨
Ascii string	ProcessMain
Utf8 string	ProcessMain
Utf16 string	隨到徽贈榆n璫械柿瑛淨氣湯g潼慶褐璫械柿潔獯瑩n

[-] Numbers

Unsigned byte	0x50
Signed byte	80
Unsigned short	0x7250
Signed short	29264
Unsigned int	0x636F7250
Signed int	1668248144
Unsigned long long	0x4D737365636F7250
Signed long long	5580931242539315792
Float	4,41701e+21
Double	1,28025e+65

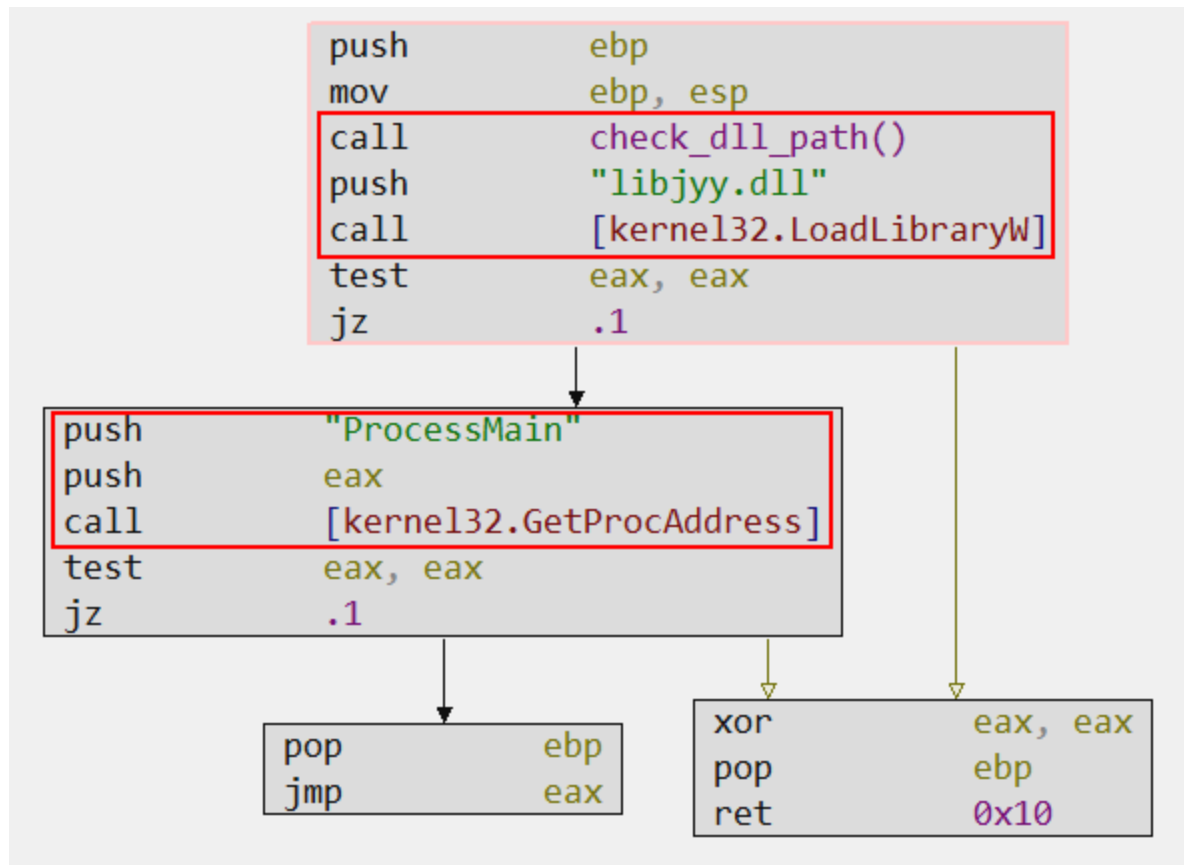
[+] Time

[+] Misc

[-] Incoming references

CODE	0x00401d27 (load_claimloader+17)
------	----------------------------------

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*

File information

File name:

libjyy.dll

File size:

244224 bytes (238.5 KiB)

MD5:

d2520edf1e9fce069e202b3b13b320fd

SHA1:

2d91a7d1ea04c0e8dd899a486123b853e19bd1d2

SHA256:

3e7384c5e7c5764258947721c7729f221fb47ef53d447a7af5db5426f1e7c13d

TLSH:

T10a348e017841c432e4bf163845ba96624abc75630b74ceefa7e84d3e0f765d06a326b7

Imphash:

481b9f28907ab2c13f12afa2767df9a7

Check online intelligence

Type



Program

PE

Metadata

Compile date:

2025-05-28 04:06:13

VersionInfo:

CompanyName:

Wargaming.net

FileDescription:

Wargaming.net Game Center

FileVersion:

24.04.00.6472

InternalName:

helper_process

LegalCopyright:

Copyright ? 2009-2024 Wargaming.net

OriginalFilename:

helper_process.Game Center

ProductName:

Wargaming.net Game Center

ProductVersion:

24.04.00.6472

Exports:

Module name:

Win32Project1.dll

Exports date:

2025-05-28 04:06:13

Debug:

Date.Debug.Pogo:

2025-05-28 04:06:13

Date.Debug.Iltcg:

2025-05-28 04:06:13

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.

CryptoShuffler	1.15 %
Ryuk	1.15 %
Dented	1.04 %
Aurora	0.94 %

Functions (279 hits)				
Address	Level	Confidence	Fuzzy?	Identification
0x10004350 (sub_10004350)	MALICIOUS		☐	Gophe (25%) + ...Pay2Key (25%)
0x10001db0 (sub_10001db0)	MALICIOUS		☐	Final1stSpy
0x10001c80 (sub_10001c80)	MALICIOUS		☐	Final1stSpy
0x10001ee0 (sub_10001ee0)	MALICIOUS		☐	Final1stSpy
0x10012869 (sub_10012869)	LIBRARY		☐	wdk-10.0.17763....0.17134 (33%)
0x1000b4c0 (SEH.1)	LIBRARY		☐	msvc-2013 (50...svc-2015 (50%)
0x1001d760 -> _GetPrimaryLen	LIBRARY		☐	wdk-10.0.17763....0.10240 (33%)
0x1000b057 -> __std_exception_destroy	LIBRARY		☐	openssl (25%) ...svc-2015 (25%)
0x10009ce0 -> __alloca_probe	LIBRARY		☐	msvc-2015
0x100068bc -> 'scalar deleting destructor'	LIBRARY		☐	msvc-2015 (33...svc-2016 (33%)
0x10010be2 -> __ischartype_l	LIBRARY		☐	wdk-10.0.17763....0.10240 (33%)

Strings (831 hits)			
String	Level	Confidence	Identification
Wargaming.net	MALICIOUS		CryptBot
unh investigation			
google Team			
cyberintelmatrix.com			
Error: File Corrupted			
The PDF file is corrupted. Pl...t your computer to try again.			
CrashForException			
DumpProcessWithoutCrash			
InjectDumpForHangDebugging			
InjectDumpProcessWithoutCrash			
TerminateProcessWithoutDump			

Constants (6 hits)		
Level	Confidence	Identification
MALICIOUS		CryptoShuffler
MALICIOUS		Ryuk
MALICIOUS		Dented
MALICIOUS		Aurora
MALICIOUS		Oni
MALICIOUS		Edam

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-jeww8asky5: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	tl!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 .yaml files in <USER DATA DIR>/scripts/capa/all_rules/*.yaml.

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]

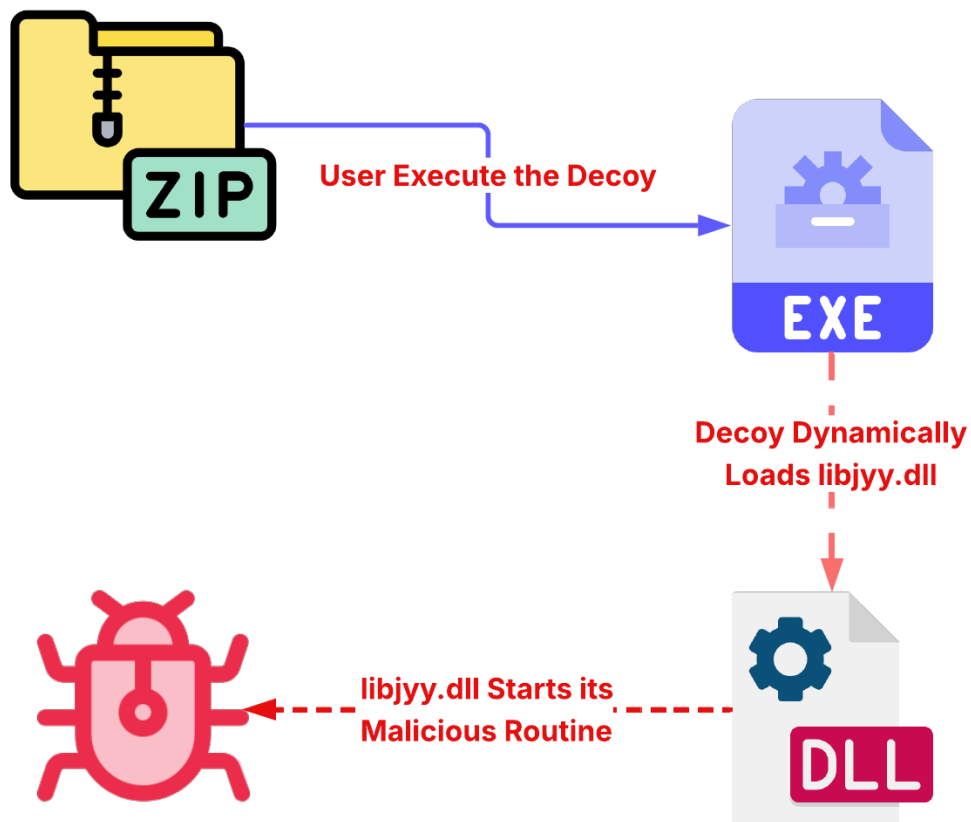
Malware Behavior Catalog

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

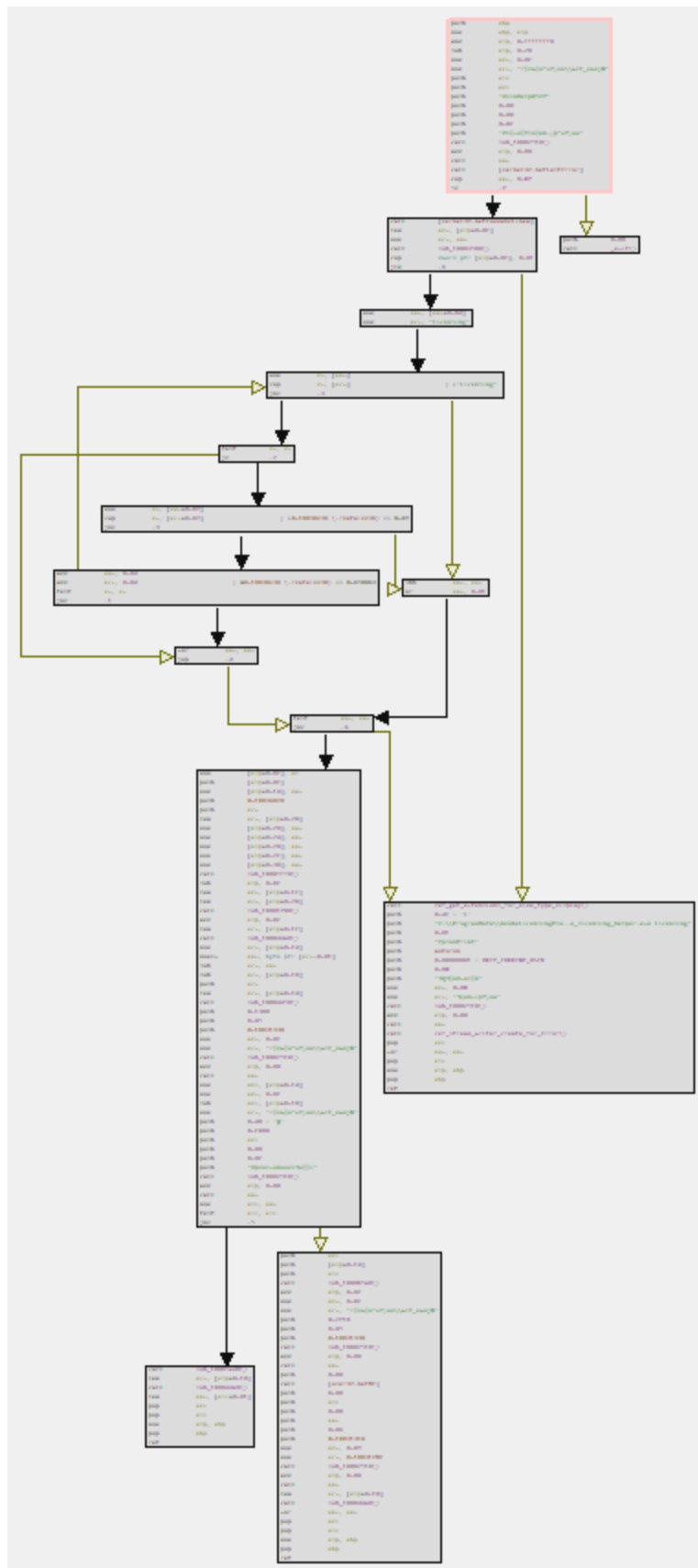
Now that we have enough information, let's start our in-depth analysis through the function in which the decoy loads and calls **libjyy.dll**, **ProgramMain**.

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  
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**.



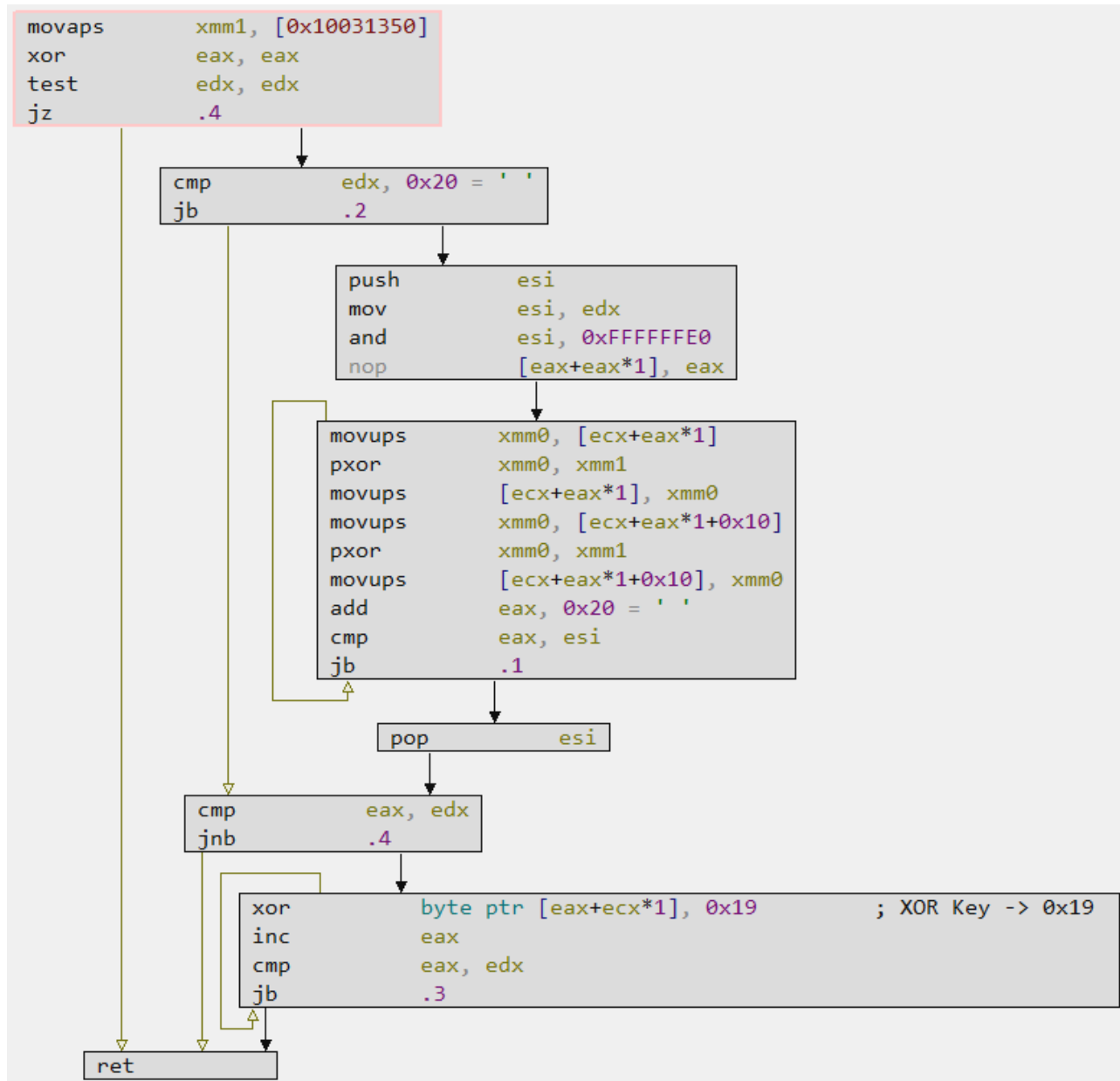
Within the **decrypt_str_load_api** function, we can observe its two main functionalities, implemented through the functions:

- **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.

8D8D78FFFFFF	lea	ecx, [ebp-0x88]	
8BD3	mov	edx, ebx	
E82AFFFFFF	call	decrypt_str_api()	↑3
8BD6	mov	edx, esi	
8D4DB8	lea	ecx, [ebp-0x48]	
E820FFFFFF	call	decrypt_str_api()	↑3
83C404	add	esp, 0x04	
682CFE0210	push	"ntdll.dll"	
FF1510300210	call	[kernel32.GetModuleHandleA]	
8BF0	mov	esi, eax	
BA38FE0210	mov	edx, "LdrLoadDll"	
8BCE	mov	ecx, esi	
E804FEFFFF	call	pe_parsing_dynamically_load_api()	
BA44FE0210	mov	edx, "LdrGetProcedureAddress"	
8BCE	mov	ecx, esi	
8BF8	mov	edi, eax	
E8F6FDFFFF	call	pe_parsing_dynamically_load_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.



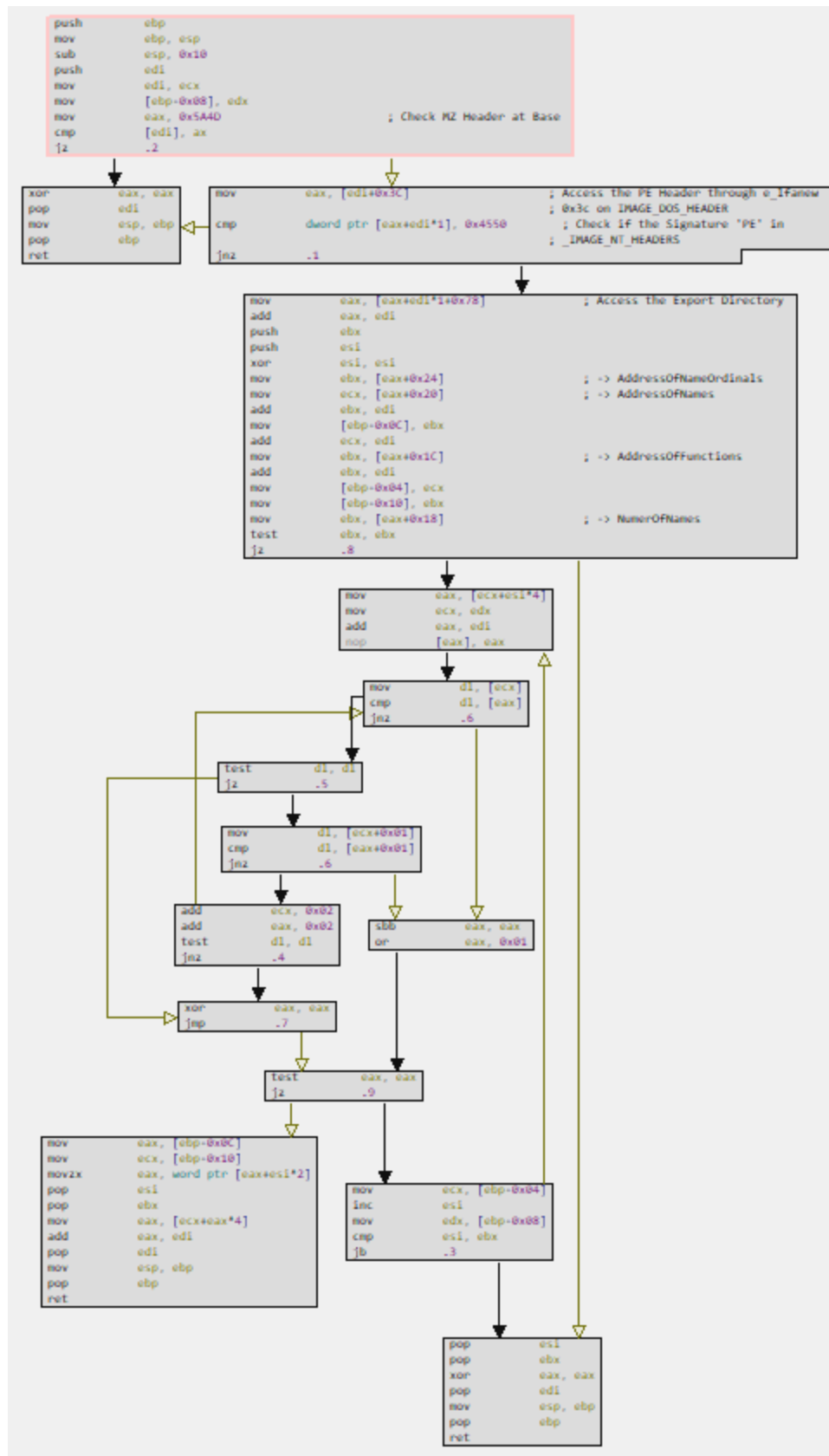
Also when analyzing the `pe_parsing_dynamically_load_api` function, we validate our hypothesis, which establishes that this function is responsible for parsing the DLL that will have a certain API called dynamically.

```

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:         xor      eax, eax
    33C0      pop      edi
    5F        mov      esp, ebp
    8BE5      pop      ebp
    5D        ret
    C3
.2:         mov      eax, [edi+0x3C]          ; Access the PE Header through e_lfanew
    8B473C      ; 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]          ; -> NumberOfNames
    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.

The image shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code includes instructions such as `mov edx, 0x0C`, `mov ecx, "r|kw|u*+7}uu\\wlt_vwmjN"`, `push esi`, `push edi`, `push "Ok in Help0527"`, `push 0x00`, `push 0x00`, `push 0x0C`, `call decrypt_str_load_api()`, `add esp, 0x08`, `call eax`, `call [kernel32.GetLastError]`, `cmp eax, 0xB7`, `jz .7`, `call [kernel32.GetCommandLine]`, `lea edx, [esp+0x0C]`, `mov ecx, eax`, `call sub_10003580()`, `cmp dword ptr [esp+0x0C], 0`, `jle .6`, `mov eax, [eax+0x04]`, `mov ecx, "Licensing"`, `mov dx, [eax]`, `cmp dx, [ecx]`, `jnz .3`, `test dx, dx`, `jz .2`, and `mov dx, [eax+0x02]`. The memory dump on the right shows hex values and their corresponding ASCII representations. A red box highlights a specific section of the memory dump containing encrypted strings like `Zk|xm|Tlm|aX~}p*+7}uu`.

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.

.rdata|010030eb0: async_begin def_trace_event_async_end def_trace_event_async_st
.rdata|010030ef0: ep_into def_trace_event_async_step_past def_trace_event_begin
.rdata|010030f30: def_trace_event_end def_trace_event_instant def_translator_test_
.rdata|010030f70: create def_translator_test_object_child_child_create def_tran
.rdata|010030fb0: slator_test_object_child_create def_translator_test_object_creat
.rdata|010030ff0: e def_unregister_internal_web_plugin def_uridecode def_urie
.rdata|010031030: ncode def_urlrequest_create def_v8context_get_current contex
.rdata|010031070: t def_v8context_get_entered_context def_v8context_in_context
.rdata|0100310b0: def_v8stack_trace_get_current def_v8value_create_array
.rdata|0100310f0: def_v8value_create_bool def_v8value_create_date def_v8value_crea
.rdata|010031130: te_double def_v8value_create_function def_v8value_create_int
.rdata|010031170: def_v8value_create_null def_v8value_create_object def_v8value_
.rdata|0100311b0: create_string def_v8value_create_uint def_v8value_create_undef
.rdata|0100311f0: ined def_api_null_hash def_value_create def_version_info
.rdata|010031230: def_visit_web_plugin_info def_write_json def_xml_reader_c
.rdata|010031270: reate def_zip_reader_create create_context_shared string t
.rdata|0100312b0: oo long invalid string position Zk|xm|Tlm|aX|}p*+7}uu Zk|xm|Ik
.rdata|0100312f0: vz|jjX ^|mTv}lu|_pu|Wxt|N n|kw|u*+7}uu\wlt_vwmjN OpkmlxuXuuvz
.rdata|010031330: Ju|i QJ|mOxul|X Jqunxip7}uu

- DebugDirectory:
 - DebugDirectory[0]:
Characteristics:
TimeDateStamp:
MajorVersion:
MinorVersion:
Type:
SizeOfData:
AddressOfRawData:
PointerToRawData:
 - DebugDirectory[1]:
Characteristics:
TimeDateStamp:
MajorVersion:
MinorVersion:
Type:
SizeOfData:
AddressOfRawData:
PointerToRawData:
- LoadConfigurationTable:
SizeOfStructure:
TimeDateStamp:
MajorVersion:
MinorVersion:
GlobalFlagsClear:
GlobalFlagsSet:

SELECTION: [0x100312d0-0x10031351]

Open

Open (new tab)

Magic select

Add user annotation ...

Change selection size ...

Shift selected bytes up/down ...

Remove selected bytes

Search in current file ...

Search in other files ...

Download selected url and Analyze

Add selection to Yara >

Copy Bytes

Copy as >

Dump to file ...

Encapsulate in new ... >

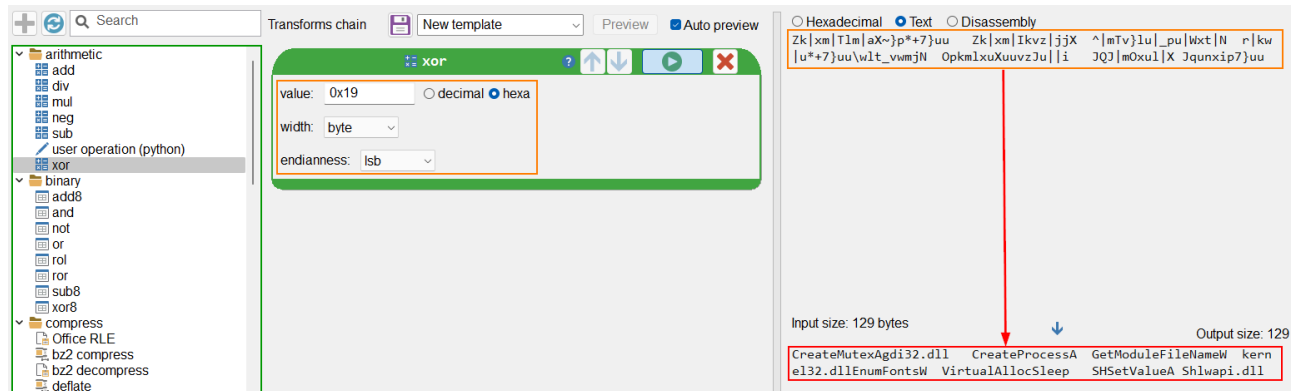
Fill >

Statistics

Transform ...

Apply last transform chain

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, 0xFFFFFFFF8
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

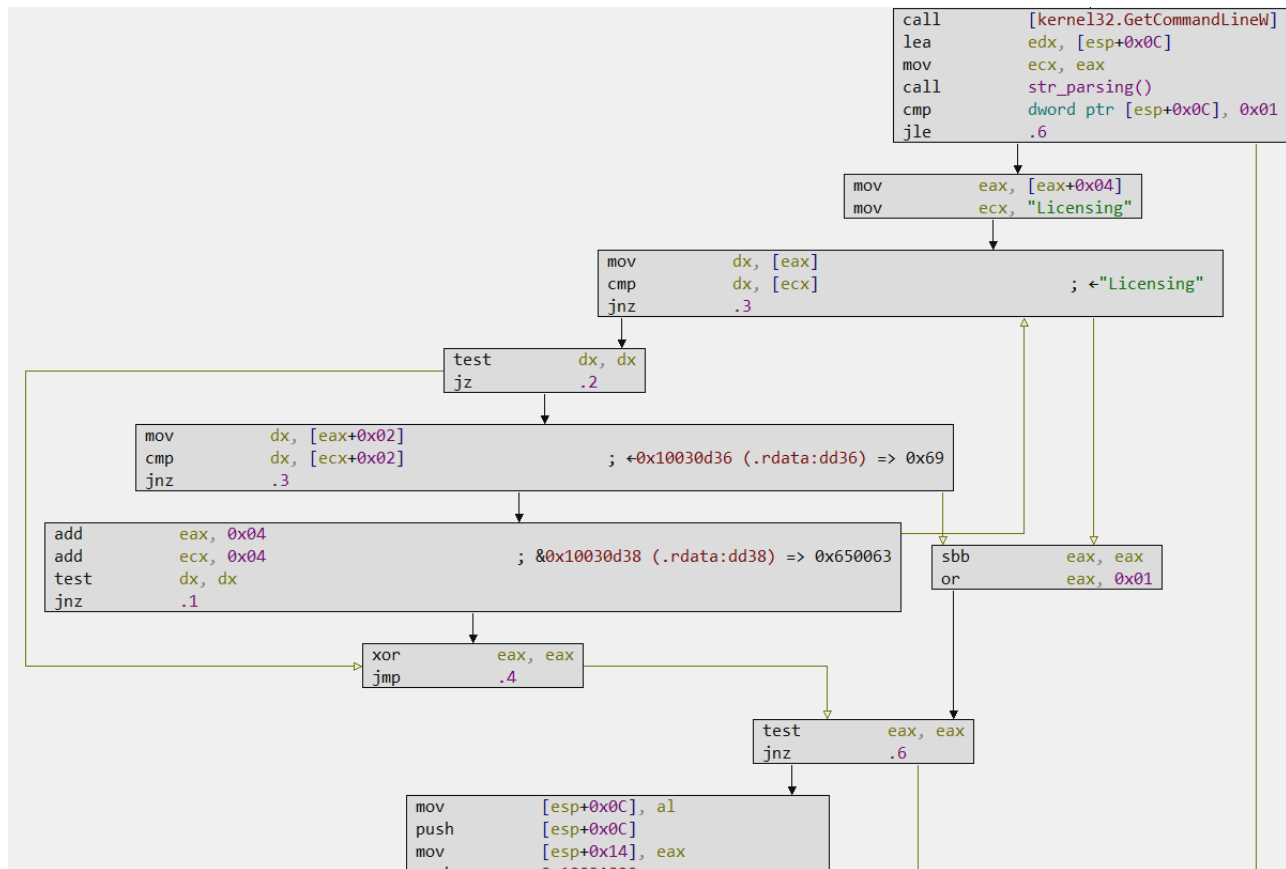
Claimloader expects to receive an argument, which will be identified and parsed in the code block below by collecting the command line using [GetCommandLineW](#), followed by parsing the collected string, which refers to the current process's command line. Depending on whether the current process's command 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

```

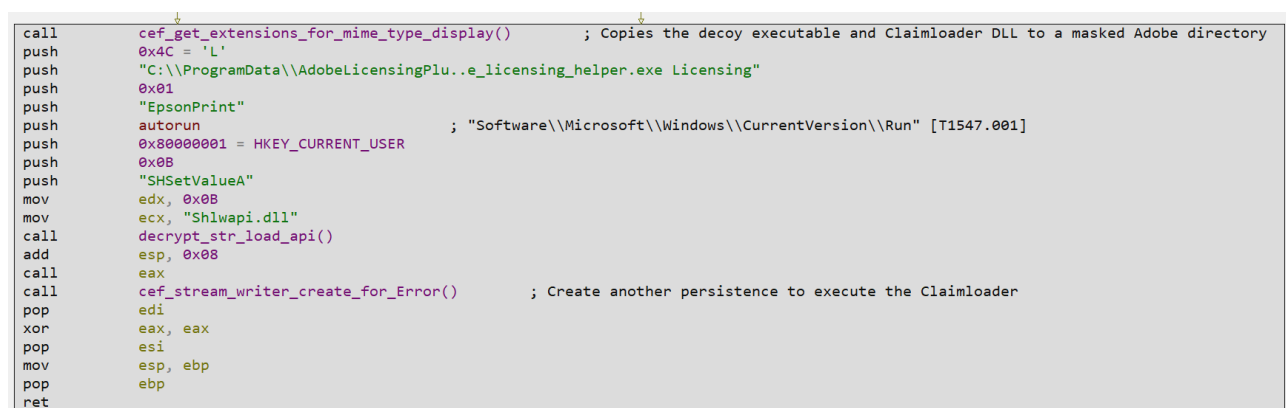
In the following image, we can see that *Claimloader* expects that, if an argument is identified, it will be “Licensing”.



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

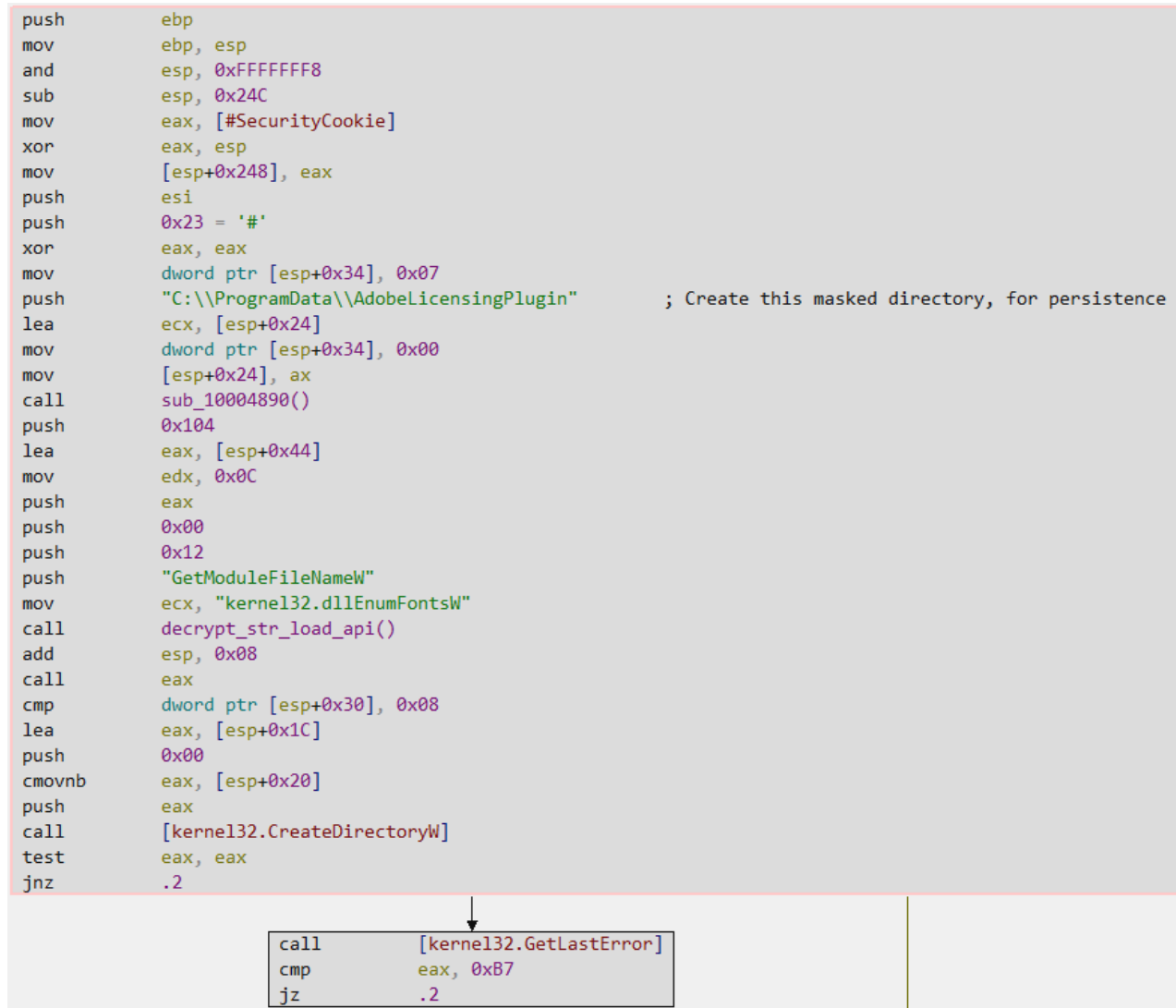
If no argument (or an incorrect argument) is identified when executing the decoy executable, *Claimloader* will execute the code block below.



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.



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:

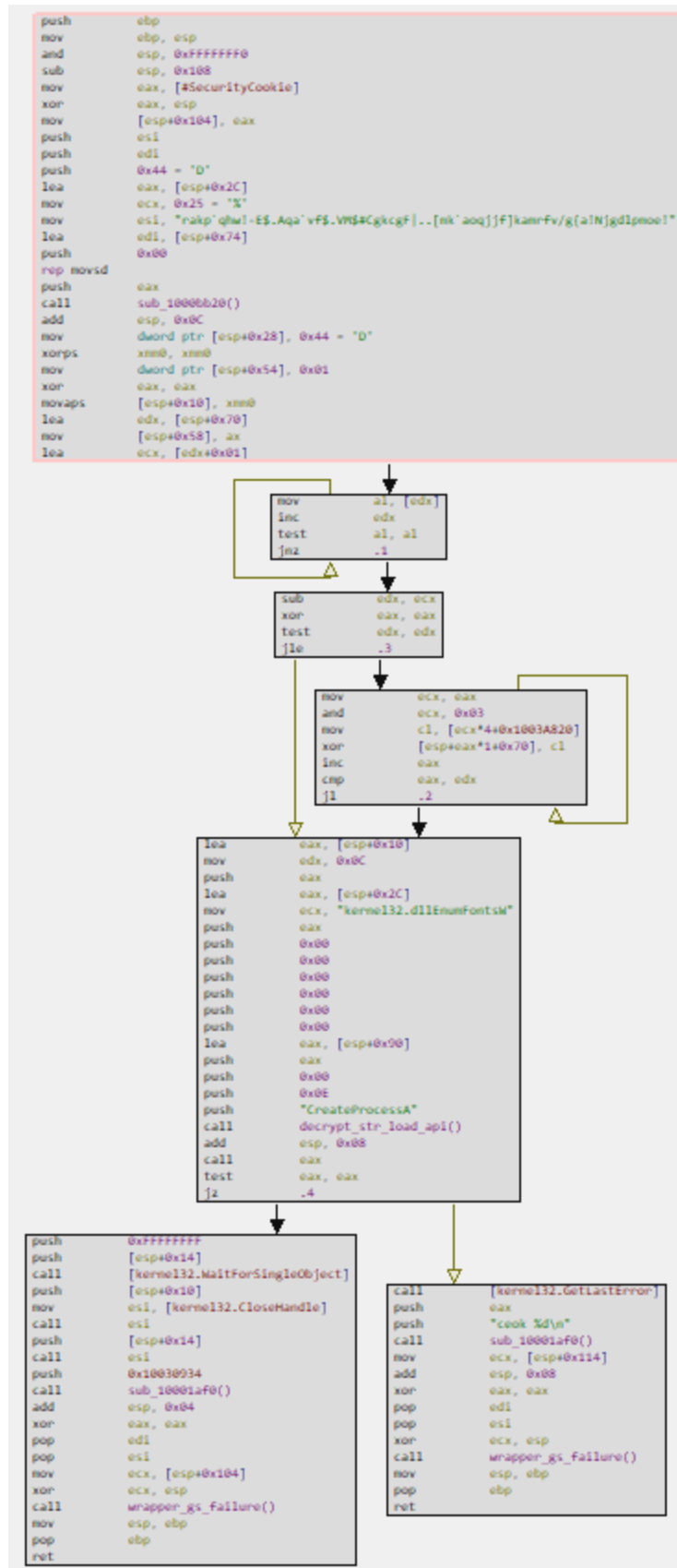
- **C:\ProgramData\AdobeLicensingPlugin\WF_Adobe_licensing_helper.exe Licensing**

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

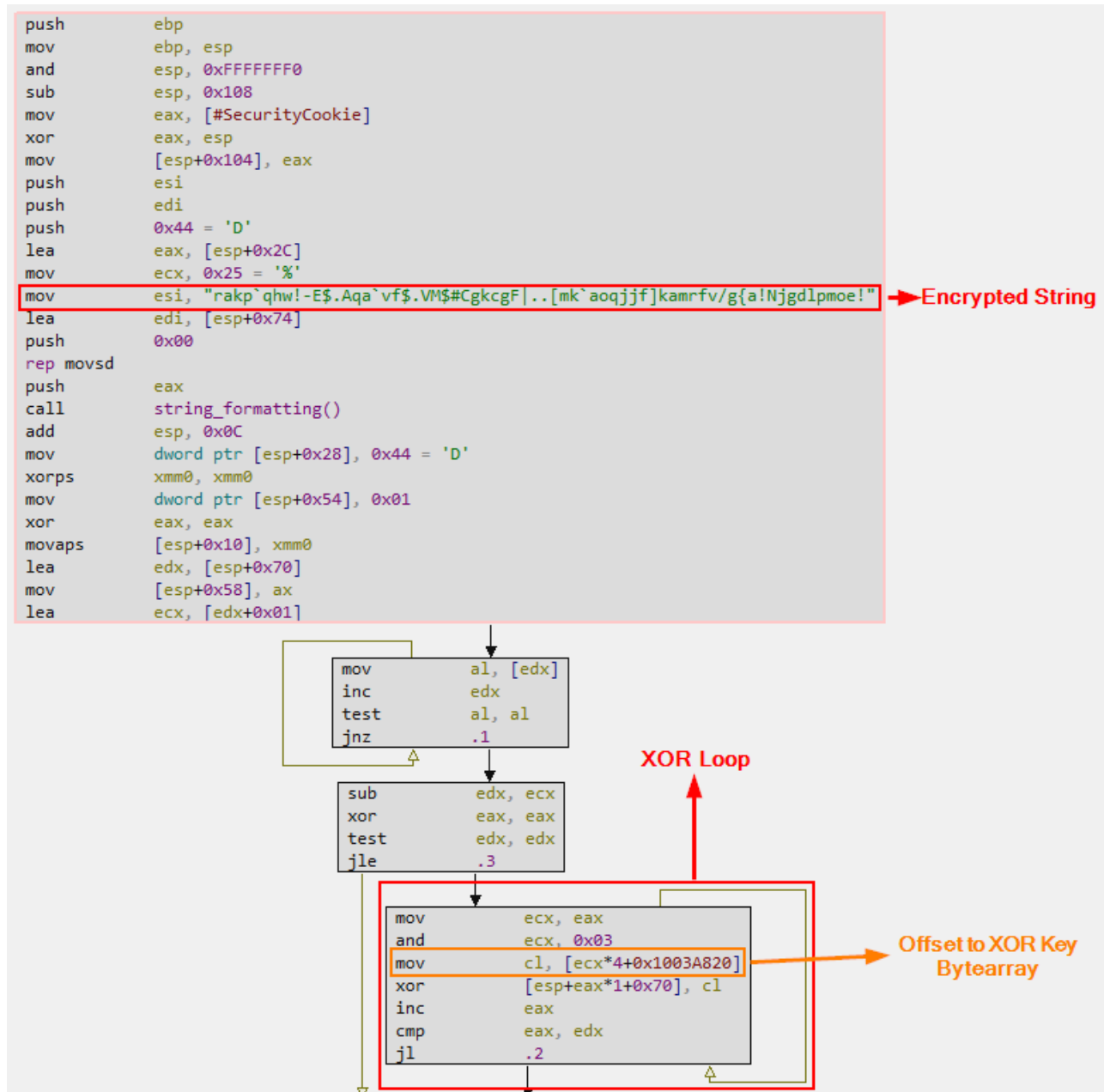
Now let's analyze the **cef_stream_writer_create_for_Error** function, which runs after the [T1547.001](#) persistence implementation. This function creates another persistence so that *Claimloader* can run effectively. It also increases the level of investigation for DFIR analysts, who can stop looking 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`.

.data	01003a820:	01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00	
.data	01003a830:	AC 63 02 10 (std::bad_alloc.Typeid) 62 61 64 5F	c .?AVbe
.data	01003a840:	61 6C 6C 6F 63 40 73 74 64 40 40 00 AC 63 02 10	alloc@std@@ c
.data	01003a850:	00 00 00 0 (std::logic_error.Typeid) 3 5F 65 72	.?AVlogic_
.data	01003a860:	72 6F 72 40 73 74 64 40 40 00 00 00 AC 63 02 10	ror@std@@ c
.data	01003a870:	00 00 00 (std::length_error.Typeid) 68 5F 65	.?AVlength
.data	01003a880:	72 72 6F 72 40 73 74 64 40 40 00 00 AC 63 02 10	rror@std@@ c
.data	01003a890:	00 00 00 (std::out_of_range.Typeid) 66 5F 72	.?AVout_of
.data	01003a8a0:	61 6E 67 65 40 73 74 64 40 40 00 00 AC 63 02 10	ange@std@@ c
.data	01003a8b0:	00 00 00 0 (std::bad_cast.Typeid) 63 61 73 74	.?AVbad_ca
.data	01003a8c0:	40 73 74 64 40 40 00 00 AC 63 02 10 00 00 00 00	@std@@ c
.data	01003a8d0:	2E 3F 41 56 (std::ios_base.Typeid) 40 73 74 64	.?AVios_base@s
.data	01003a8e0:	40 40 00 00 AC 63 (std::ios_base<int>.Typeid) 11 56	@@ c .?
.data	01003a8f0:	3F 24 5F 49 6F 73 62 40 48 40 73 74 64 40 40 00	?\$_Iosb@H@std@
.data	01003a900:	AC 63 02 10 00 00 00 00 2E 3F 41 56 3F 24 62 61	c .?AV?\$_
.data	01003a910:	73 69 63 5F 69 6F 73 40 44 55 3F 24 63 68 61 72	sic_ios@DU?\$_cl
.data	01003a920:	5F 74 72 61 69 74 73 40 44 40 73 74 64 40 40 40	_traits@D@std@
.data	01003a930:	73 74 64 40 40 00 00 00 AC 63 02 10 00 00 00 00	std@@ c
.data	01003a940:	2E 3F 41 56 3F 24 62 61 73 69 63 5F 73 74 72 65	.?AV?\$_basic_st
.data	01003a950:	61 6D 62 75 66 40 44 55 3F 24 63 68 61 72 5F 74	ambuf@DU?\$_char


```

mov     ecx, [ecx*4+0x1003A820]
xor     [esp+eax*1+0x70], ecx
inc     eax
cmp     eax, edx
jl      .2

```

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$.Aga`vf$.VM$#CgkcgF|qgqmdl`aLcmefgg&!-PG!ojjtvf$.OL$3",PS"IG;^Svneqe1Fbp`^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\Adobe\LicensingPlugin\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").

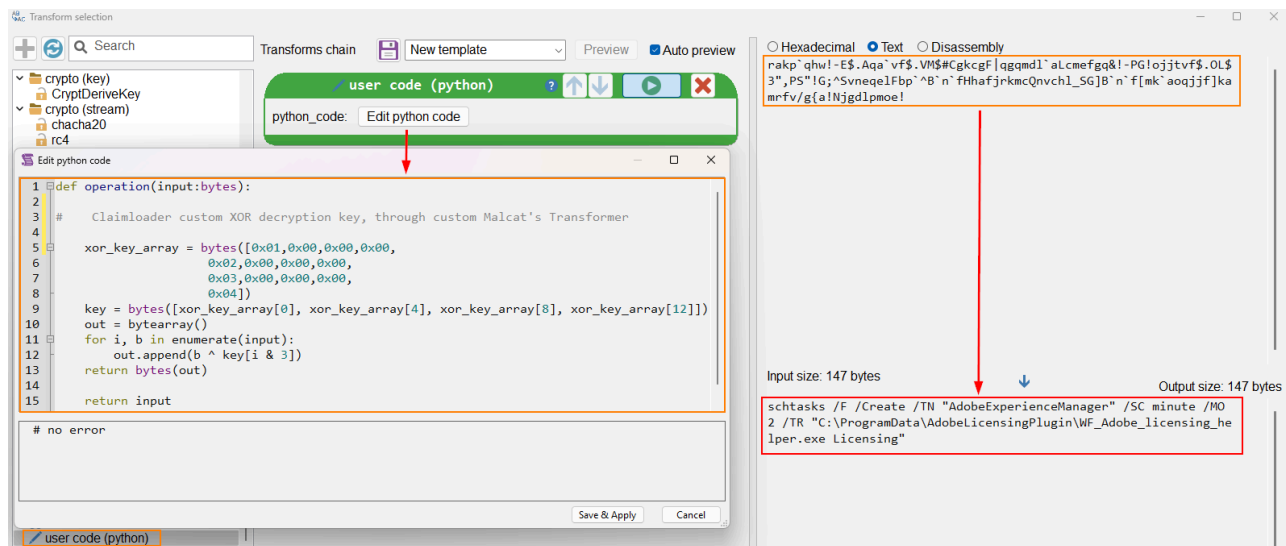
/SC minute → Sets the execution frequency: every minute.

/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.



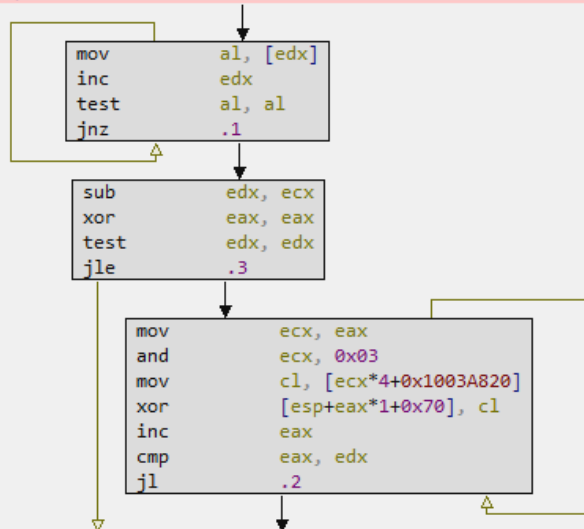
In the following image, we can see the command in plain text and decrypted, making the final functionality of this function clearer.

```

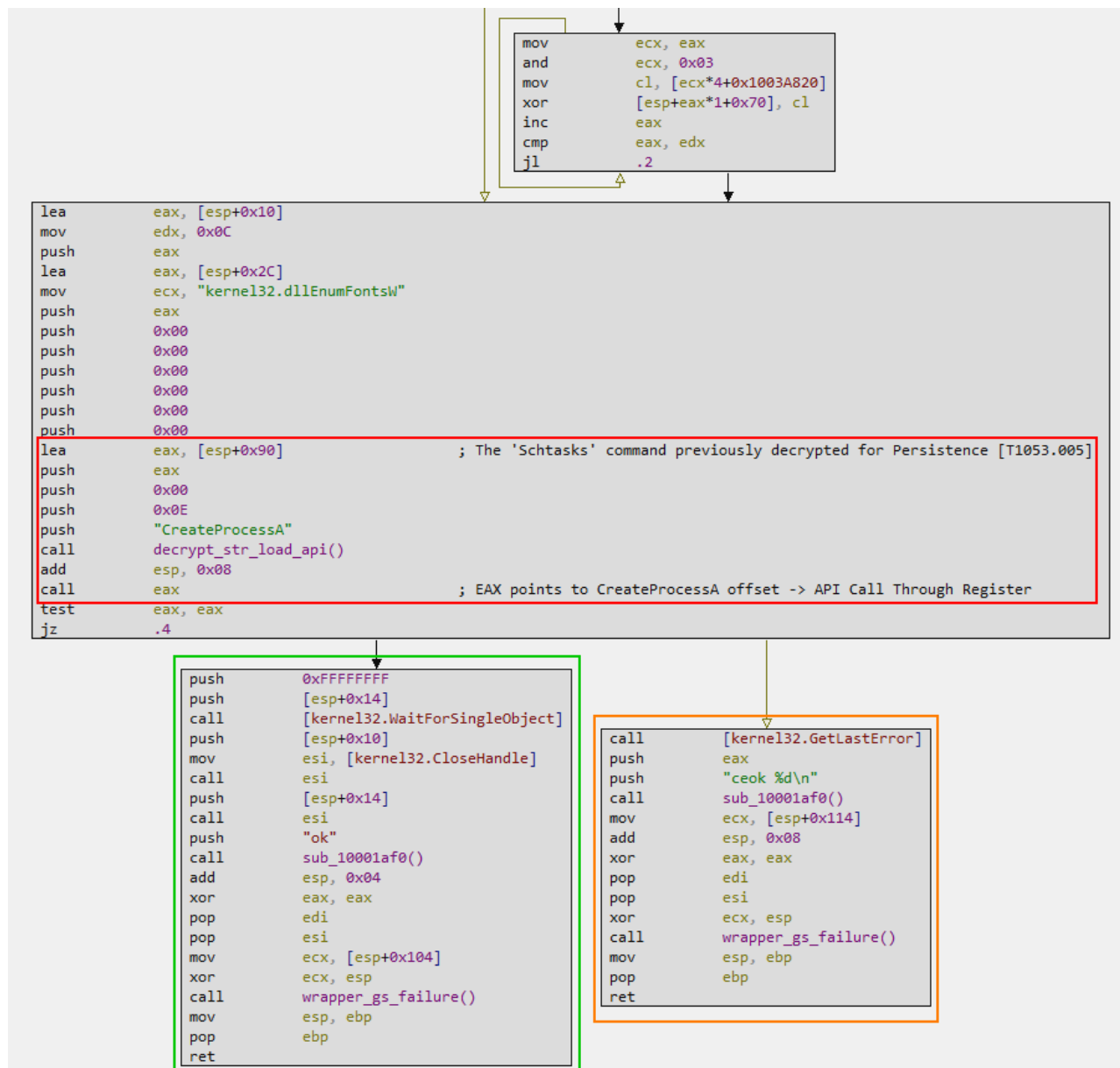
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0
sub     esp, 0x108
mov     eax, [#SecurityCookie]
xor     eax, esp
mov     [esp+0x104], eax
push    esi
push    edi
push    0x44 = 'D'
lea     eax, [esp+0x2C]
mov     ecx, 0x25 = '%'
mov     esi, "schtasks /F /Create /TN \"AdobeEx.._licensing_helper.exe 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.

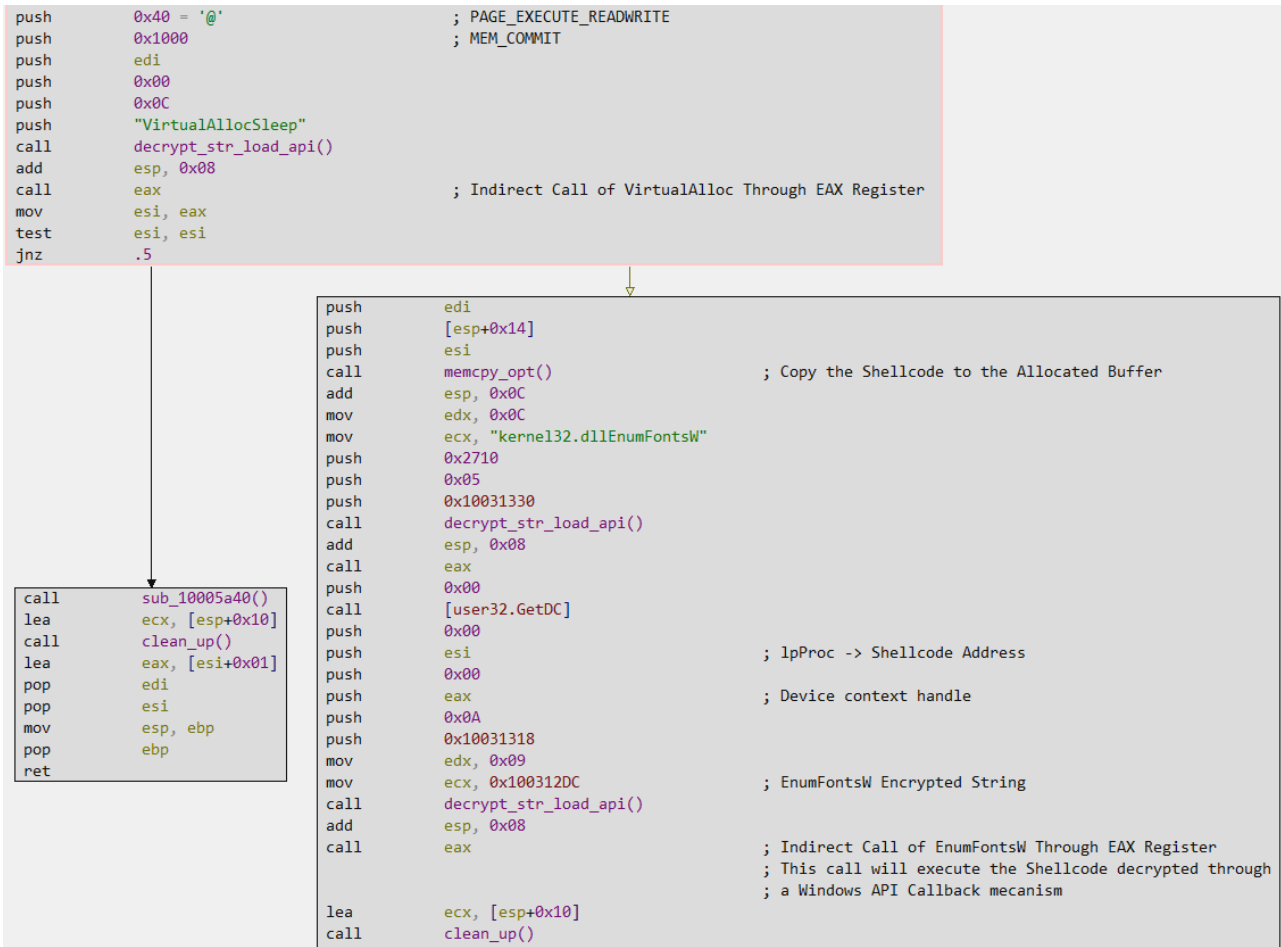


Therefore, we can see that this entire *Claimloader* execution flow is the victim's first execution, and the victim clearly doesn't know they're running malware, much less that an argument is required for it to function properly. Thus, when the user executes the decoy normally, *Claimloader* will execute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

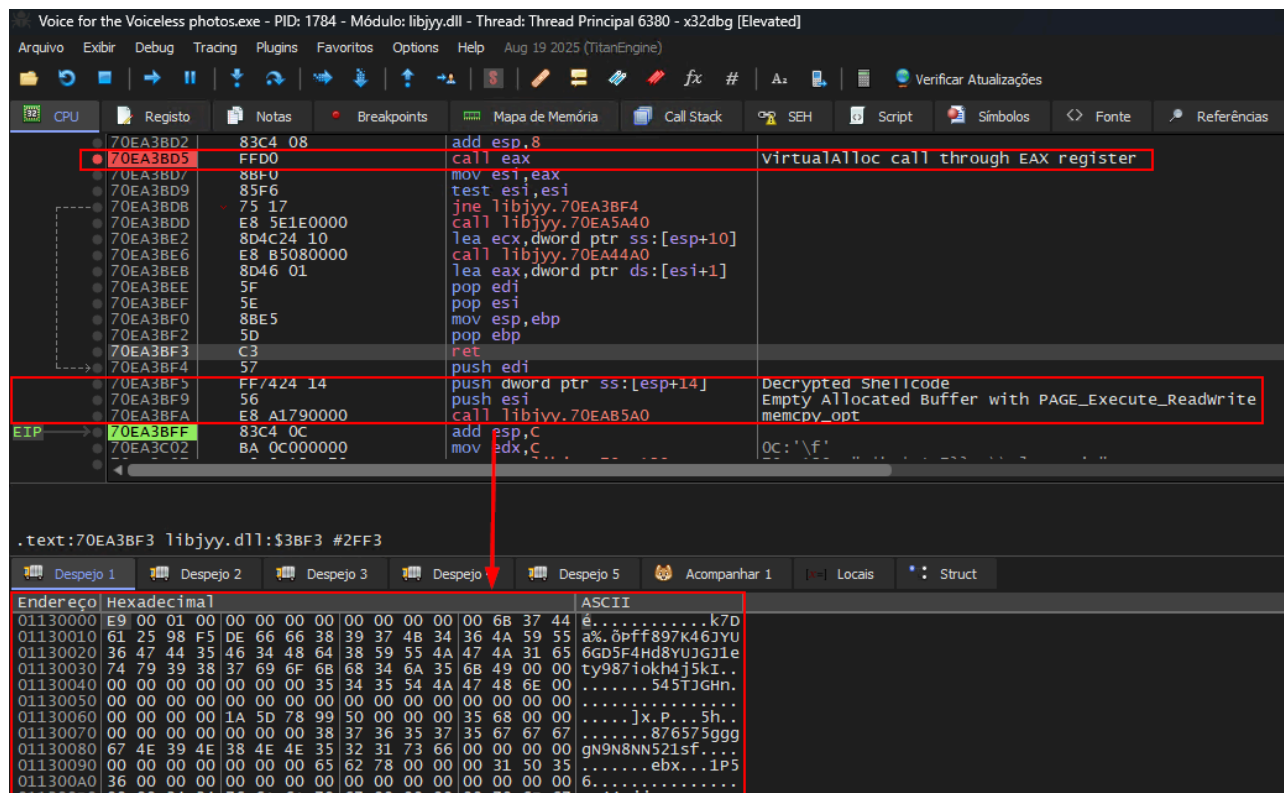
Below, we can see the other flow that will only be executed if the decoy is executed containing the 'Licensing' argument.



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.

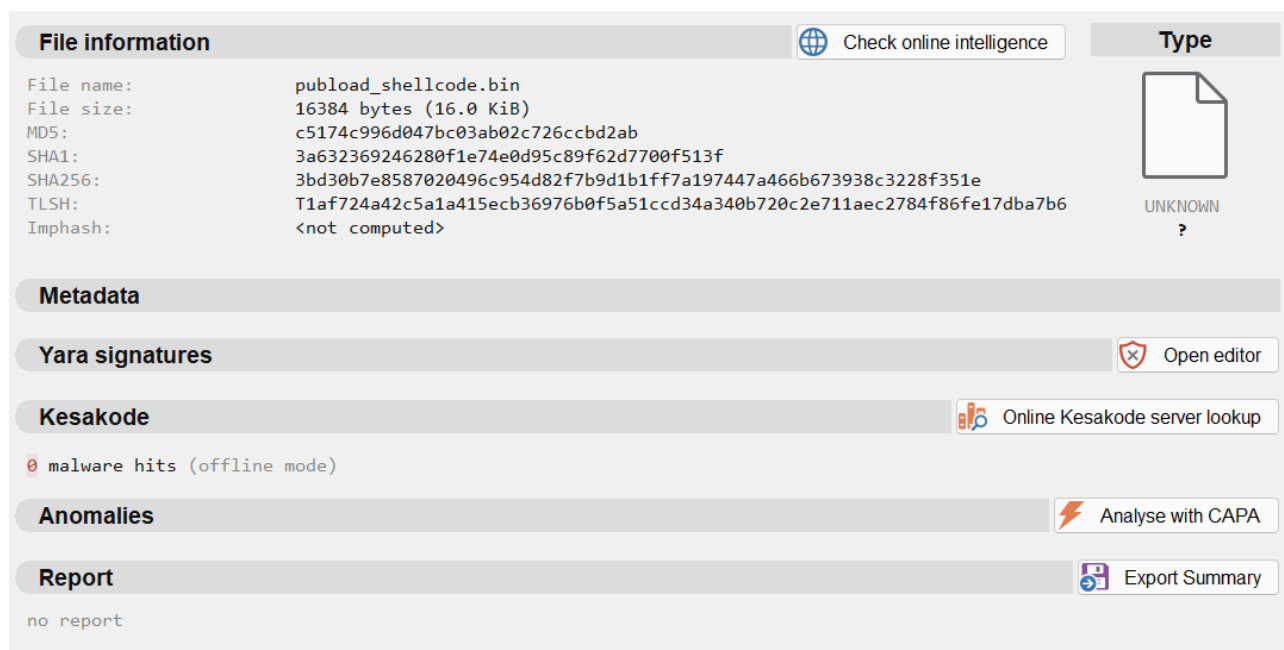
After copying the shellcode to the previously allocated buffer, in the image above we can also see that the offset of this buffer will be used as an argument for the *EnumFontsWith* function call, which, when called, will execute the shellcode by abusing the API's Callback mechanism. The *lpProc* argument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

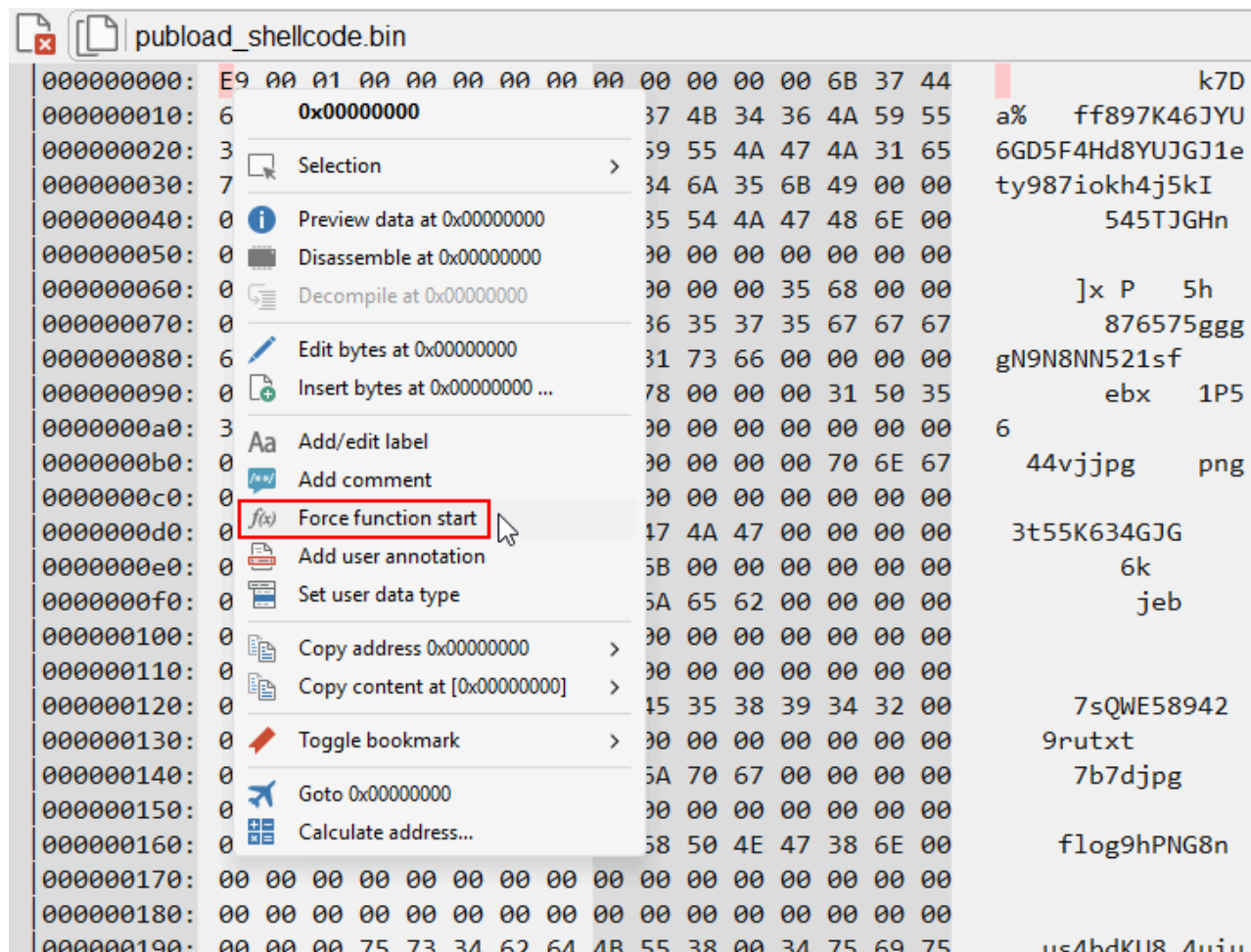


Extracted Shellcode Functionality Identification

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.

pubload_shellcode.bin		
00000000:	E9 00 01 00 00 00 00 00 00 00 00 00 6B 37 44	k7D
00000010:	61 25 98 F5 DE 66 66 38 39 37 4B 34 36 4A 59 55	a% ff897K46JYU
00000020:	36 47 44 35 46 34 48 64 38 59 55 4A 47 4A 31 65	6GD5F4Hd8YUJGJ1e
00000030:	74 79 39 38 37 69 6F 6B 68 34 6A 35 6B 49 00 00	ty987iokh4j5kI
00000040:	00 00 00 00 00 00 00 35 34 35 54 4A 47 48 6E 00	545TJGHn
00000050:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000060:	00 00 00 00 1A 5D 78 99 50 00 00 00 35 68 00 00	]x P 5h
00000070:	00 00 00 00 00 00 00 38 37 36 35 37 35 67 67 67	876575ggg
00000080:	67 4E 39 4E 38 4E 4E 35 32 31 73 66 00 00 00 00	gN9N8NN521sf
00000090:	00 00 00 00 00 00 00 65 62 78 00 00 00 31 50 35	ebx 1P5
000000a0:	36 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6
000000b0:	00 00 34 34 76 6A 6A 70 67 00 00 00 00 70 6E 67	44vjjpg png
000000c0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
000000d0:	00 33 74 35 35 4B 36 33 34 47 4A 47 00 00 00 00	3t55K634GJG
000000e0:	00 00 00 00 00 00 00 00 36 6B 00 00 00 00 00 00	6k
000000f0:	00 00 00 00 00 00 00 00 00 6A 65 62 00 00 00 00	jeb
00000100:	00 00 00 00 00 E9 00 sub_0 00 00 00 00 00 00 00	
00000110:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000120:	00 00 00 00 00 37 73 51 57 45 35 38 39 34 32 00	7sQWE58942
00000130:	00 00 00 39 72 75 74 78 74 00 00 00 00 00 00 00	9rutxt
00000140:	00 00 00 00 00 37 62 37 64 6A 70 67 00 00 00 00	7b7djjpg
00000150:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000160:	00 00 00 98 66 6C 6F 67 39 68 50 4E 47 38 6E 00	flog9hPNG8n
00000170:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000180:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00000190:	00 00 00 75 73 34 62 64 4B 55 38 00 34 75 69 75	us4bdKU8 4uiu
000001a0:	6B 68 00 00 00 00 00 00 00 00 00 00 00 00 00	kh
000001b0:	00 00 00 00 00 00 00 77 77 37 37 35 36 4A 54 55	ww7756JTU
000001c0:	49 00 00 00 00 00 00 00 00 00 00 00 00 00 00	I
000001d0:	00 00 00 00 00 00 76 00 00 36 35 45 48 47 74 75	v 65EHGtu
000001e0:	67 00 00 00 00 00 00 00 00 00 00 00 00 00 00	g
000001f0:	00 00 00 00 00 00 00 00 50 64 66 4E 42 48 47 4C	PdfNBHGL
00000200:	46 46 46 46 46 46 35 65 6E 39 E9 50 00 00 00 55	FFFFFF5en9 P U
00000210:	8B EC 51 6A 04 68 00 30 00 00 8B 45 08 50 6A 00	Qj h 0 E Pj
00000220:	FF 55 0C 89 45 FC 8B 45 FC 8B E5 5D C3 CC CC 55	U E E] U
00000230:	8B EC 51 8B 45 08 89 45 FC 83 7D FC 00 74 19 68	Q E E } t h
00000240:	00 80 00 00 6A 00 8 sub_22f 1 8B 55 FC 8B 82 70	j M Q U p
00000250:	00 01 00 8B 48 08 FF D1 8B E5 5D C3 CC CC CC 55	H] U
00000260:	8B EC 81 EC CC 00 00 00 56 C7 45 A4 01 00 00 00	V E
00000270:	C7 45 F4 00 00 00 00 C7 45 FC 00 00 00 00 8D 85	E E
00000280:	34 FF FF FF 89 45 FC 6A 70 6A 00 8B 4D FC 51 E8	4 E jpj M Q
00000290:	0B 0A 00 00 83 C4 0 sub_25f 16 FC 8F F8 1F 03 00	h?

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]

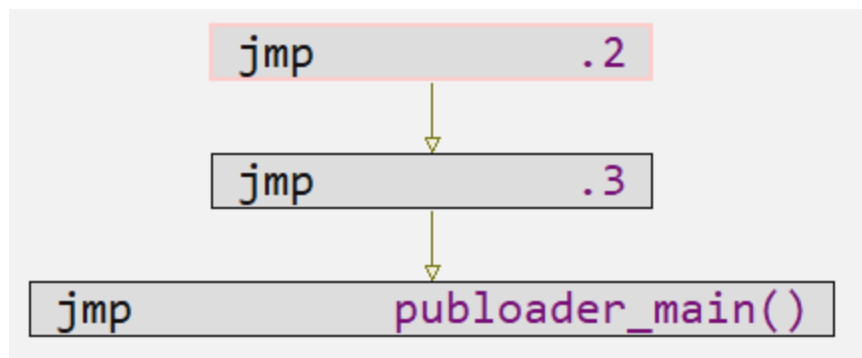
11 capabilities found	
CAPABILITY	NAMESPACE
resolve function by hash	linking/runtime-linking

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*.



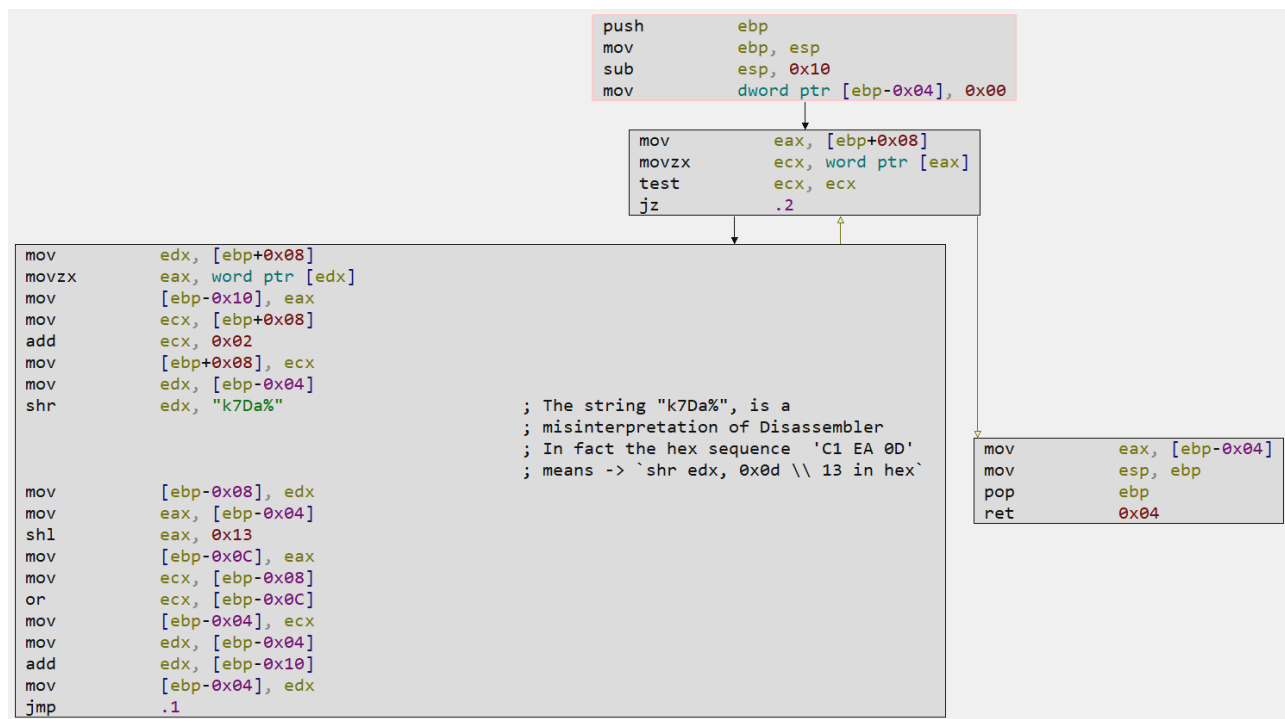
The first action of Publoader's Main function is to resolve kernel32.dll in two ways. To ensure the DLL is resolved and finally loaded, it attempts to resolve it using two hashes (**6E2BCA17** and **8FECD63F**), a critical step for the rest of the code. When resolving kernel32.dll through the **publoader_dll_hash_ror13**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

0277126F	55	push ebp	
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]	
02771288	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

0277140E	8B45 F8	mov eax,dword ptr ss:[ebp-8]	
02771411	8B88 70000100	mov ecx,dword ptr ds:[eax+10070]	
02771417	8B51 44	mov edx,dword ptr ds:[ecx+44]	edx:closesocket, [ecx+44]:socket
0277141A	FFD2	call edx	Indirect socket call
0277141C	8945 FC	mov dword ptr ss:[ebp-4],eax	
0277141F	837D FC FF	cmp dword ptr ss:[ebp-4],FFFFFFFF	
02771423	75 05	jne 277142A	
02771425	83C8 FF	or eax,FFFFFFFF	
02771428	EB 36	jmp 2771460	
0277142A	6A 10	push 10	
0277142C	8B45 08	mov eax,dword ptr ss:[ebp+8]	
0277142F	50	push eax	
02771430	8B4D FC	mov ecx,dword ptr ss:[ebp-4]	
02771433	51	push ecx	
02771434	8B55 F8	mov edx,dword ptr ss:[ebp-8]	edx:closesocket
02771437	8B82 70000100	mov eax,dword ptr ds:[edx+10070]	edx+10070:WEP+6080
0277143D	8B48 60	mov ecx,dword ptr ds:[eax+60]	
02771440	FFD1	call ecx	Indirect connect call
02771442	85C0	test eax,eax	
02771444	74 17	je 277145D	
02771446	8B55 FC	mov edx,dword ptr ss:[ebp-4]	edx:closesocket
02771449	52	push edx	edx:closesocket
0277144A	8B45 F8	mov eax,dword ptr ss:[ebp-8]	
0277144D	8B82 70000100	mov ecx,dword ptr ds:[eax+10070]	
02771453	8B51 48	mov edx,dword ptr ds:[ecx+48]	edx:closesocket, [ecx+48]:closesocket
EIP → 02771456	FFD2	call edx	Indirect closesocket call
02771458	83C8 FF	or eax,FFFFFFFF	

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!