TOITOIN Trojan: A New Multi-Stage Attack Targeting LATAM

By Niraj Shivtarkar, Preet Kamal

Published: 2023-07-07 · Archived: 2026-04-05 21:17:58 UTC

Analysis Of The Multi-Staged TOITOIN Infection Chain

Stage-1: Downloader module

Examination of the TOITOIN downloader module reveals its intricate operations, including string decryption routines, path retrieval, log file creation, and the selection of random file names. Understanding the string decryption process employed by malware is vital for defenders as it enables them to detect encrypted or obfuscated strings, analyze the attack, attribute it to specific threat actors, respond effectively, and develop mitigation strategies. The findings in this section shed light on the downloader module's functionalities and provide valuable insights into the overall execution flow of the TOITOIN malware.

During the analysis of the malware, specific attention was given to the downloader module. The path to the module's Program Database (PDB) file was identified as "**F:\Trabalho_2023\OFF_2023\LOAD_APP_CONSOLE_C_PLUS\LOAD\x64\Release\NAME.pdb**."

Upon execution, the downloader module initiates a String Decryption routine. Initially, the encrypted hex strings are concatenated in reverse order, employing multiple heap allocations. The resulting concatenated encrypted hex string is then passed as an argument to the decryption routine, as depicted in Figure 5 below.

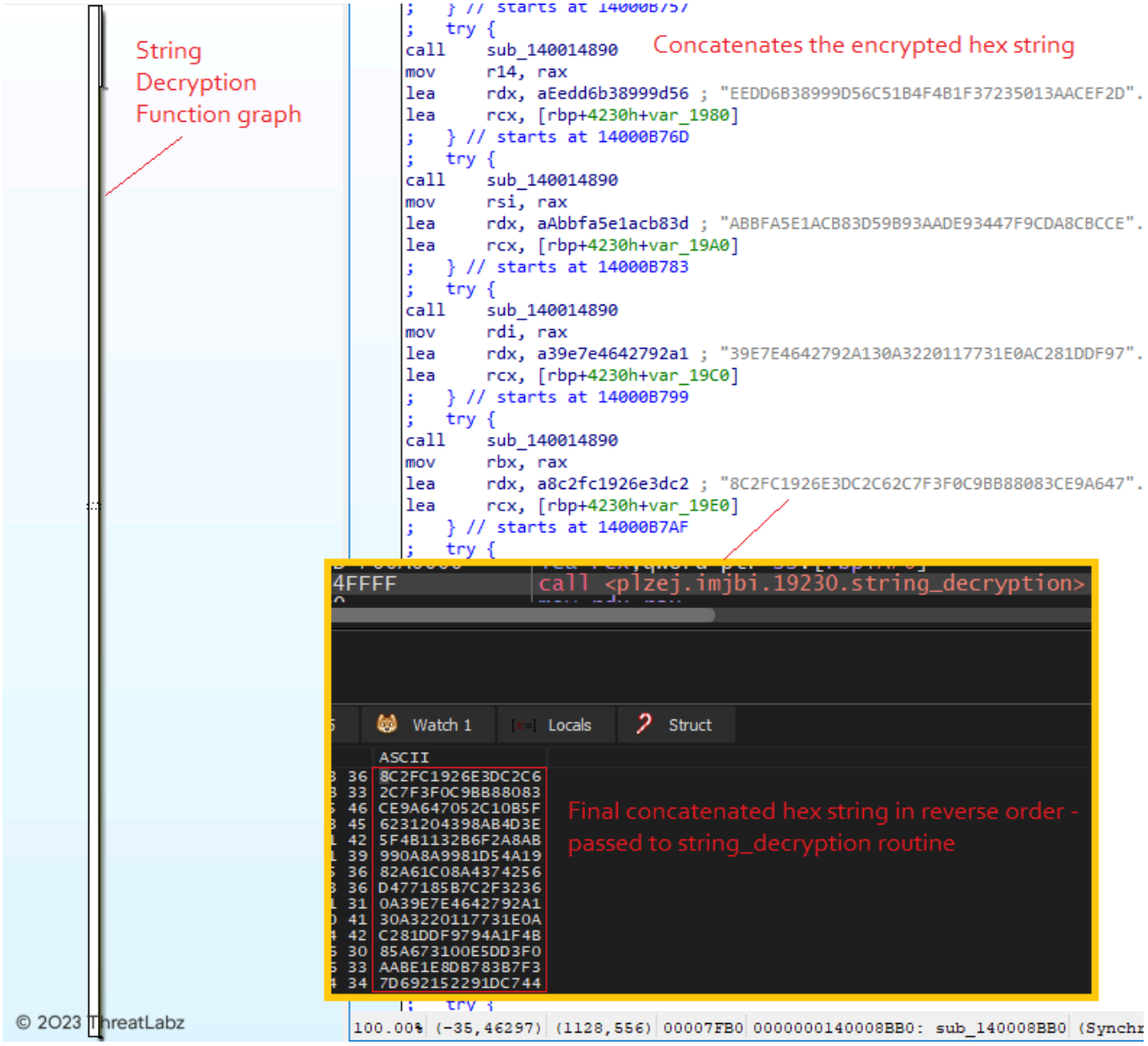

Figure 5 - Illustrates the String decryption routine, showcasing the concatenation process.

In the decryption routine, the encrypted hex string undergoes a series of operations. Firstly, the string is reversed, and subsequently, an XOR operation is performed between the N and N+1 byte, where N is incremented by 2 for each operation. To facilitate this process, a string decryptor was developed (Code: Appendix A) specifically for the string decryption routine. Utilizing this string decryptor, the final concatenated encrypted hex string can be decrypted, revealing a decrypted string in the pattern of "**@1-55:** ." Each of these encrypted hex strings is then individually decrypted using the same string decryption function, based on the specific index value passed to the function according to the requirements. Figure 6 shows the decryption of the encrypted hex strings using the string decryptor.

Figure 6 - Overview of the string decryption routine, focusing on the decryption of the downloader URL.

Once decrypted, the downloader module retrieves the paths to the '**Appdata**', '**HomeDrive**', and '**Username**' of the infected system by calling the **getenv()** function, with the decrypted strings "**appdata, homedrive, username**" as arguments. The module then proceeds to create a log file named "**.log**" within the "**AppData/Roaming**" directory. The computer name is obtained by invoking the **GetComputerNameA()** function.

Additionally, the downloader module selects a random file name from a collection of encrypted hex strings, shown below in Figure 7. These file names are decrypted dynamically using the same string decryption routine. The chosen file name is assigned to a signed executable responsible for sideloading the Krita Loader DLL. Further analysis of this process is presented in the subsequent sections.

icolover	HDDExpert	ksolaunch	vpncmgr	DocumentCollector
LaunchWallpaper	FreeFileSync	SyncBackFree	Twake	OpenDrive_Tray
Typograf	RealTimeSync	TwinkiePaste	SysInfo-EDB-to-PST-Converter(Demo)v22.0	OpenDrive
Type3	PrivaZer	DRSZohoMailBackupTool	wingetui	NetFrameCheck
CrossFnt	NTLite	MTPDFEditor	kdeconnect-app	esc
FontViewer	Passliss	FoxitPDFReader	ImageUploader	MSIPackageBuilder
nexusfont	SystemReport	iCUE	Acoustica	Beeftext
CLIPStudio	SteelSeriesGG	DVDFabPasskey	FileMove	soffice
Start11	HWiNFO64	BabelEdit	TextPad	pdfstudioviewer2022
EpicPen	SzArchiver	Scrivener	SideSlide	muCommander
ElevenClock	Syncovery	TextMaker	ScreamingFrogLogFileAnalyser	pdfConverterOverseas
Fences	picopdf	PlanMaker	AdobeDNGConverter	Kyklops
EarthView	RegCool	Presentations	PilotEdit	SmartSwitchPC
HopToDesk	Integrator	TypeButler	bdcam	ReNamer
CleverNote	Ighub	PDFKeeper	AutoHotkeyUX	DRSMSGConverter
Envelopep	UCheck64	LogiTune	AutoHotkey	PDFShaper
sticker	icepdfeditor	TrueBurner	SophiApp	WinX_DVD_Ripper_Platinum
CapCut	DropIt	Wrike	SysInfoPSTConverterTool	BurnAware
StorYBook	XYplorer	pdfReducer	StartupManager	BurnAwareFree
EFSUM	cherrytree	GlassWire	LostFiles	
prusa-slicer	dupsct	Obsidian	sanity	
phraseexpress	vuescan	balenaEtcher	SanityCheck	
TweakPower	Write-a-Document	MasterPDFEditor	rpi-imager	
id_win	csvedit	SysInfoMSGConverterTool	redbutton	© 2023 ThreatLabz

Figure 7 - Showcases a list of randomly generated file names.

Once a file name is selected, the downloader module proceeds to create a batch script in the temp directory with a dynamically generated name. The necessary information for the batch script, including the path to the temp directory, extensions, and content, is decrypted using the string decryption routine.

Upon execution, the batch script writes and executes a **VBScript** within the temp directory. The VBScript, in turn, creates a shortcut **(.LNK)** file in the startup folder. The name of the shortcut file, "**icepdfeditor.lnk**," is dynamically set to the previously selected random file name from the list. The **TargetPath** of the shortcut file is assigned as "**C:\Users\Public\Documents\knight\icepdfeditor.exe**," with the file name again set to the random selection from the list. The VBScript, identified as "**rnTiucm.vbs**," is subsequently deleted towards the end of this process.

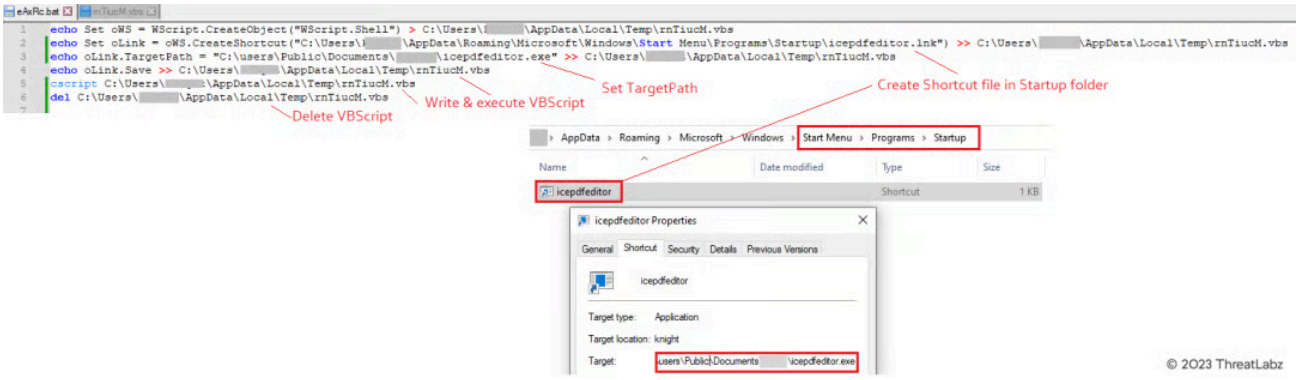

Figure 8 - Batch script creating LNK file in the StartUp folder for persistence.

The above figure illustrates the batch script's creation of an LNK file in the StartUp folder, ensuring persistence on the compromised machine. By placing the "**icepdfeditor.lnk**" shortcut in the StartUp folder, it executes every time the system restarts, subsequently launching "**icepdfeditor.exe**" in the Public Documents folder.

Following this, the downloader module initiates the downloading routine, decrypting URLs dynamically using the string decryption process, as shown in Figure 9 below.

Figure 9 - String decryption routine (downloader URLs).

Then in Figure 10 demonstrates the use of **InternetOpenUrlA()** and **InternetReadFile()** functions to retrieve encrypted data containing multiple payloads for this complex attack, disguised here as mp3 files from the URL: **http[:]//cartolabrasil[.]com/Homicidio[.]mp3/1-6.mp3**.

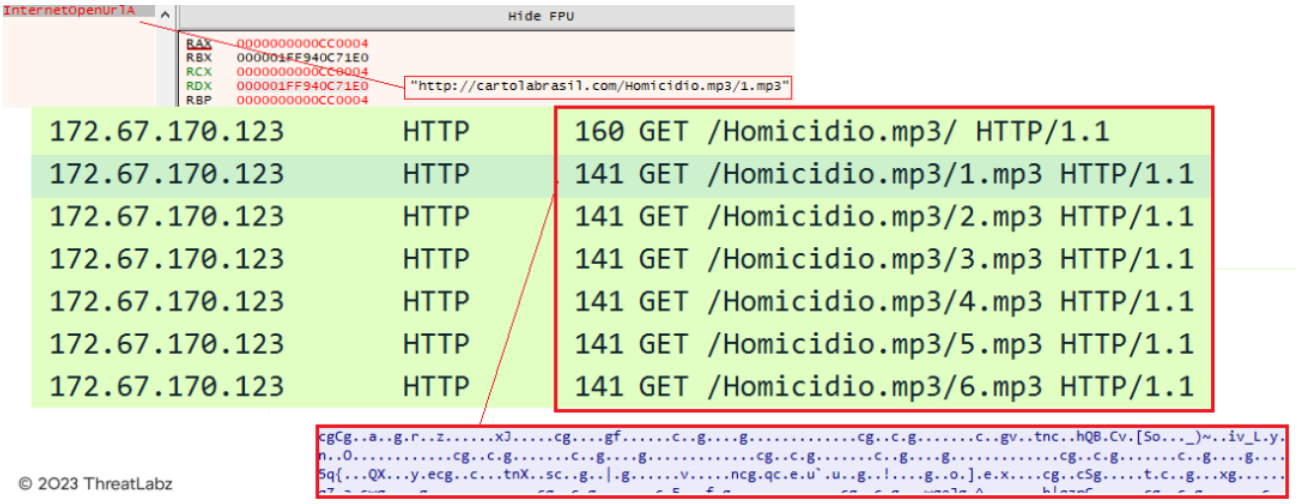

Figure 10 - Downloading multiple payloads from http[:]cartolabrasil.com.

The encrypted data is decrypted and reversed, and the resulting payloads are written to a newly created folder within the Public Documents directory, as depicted in Figure 11.

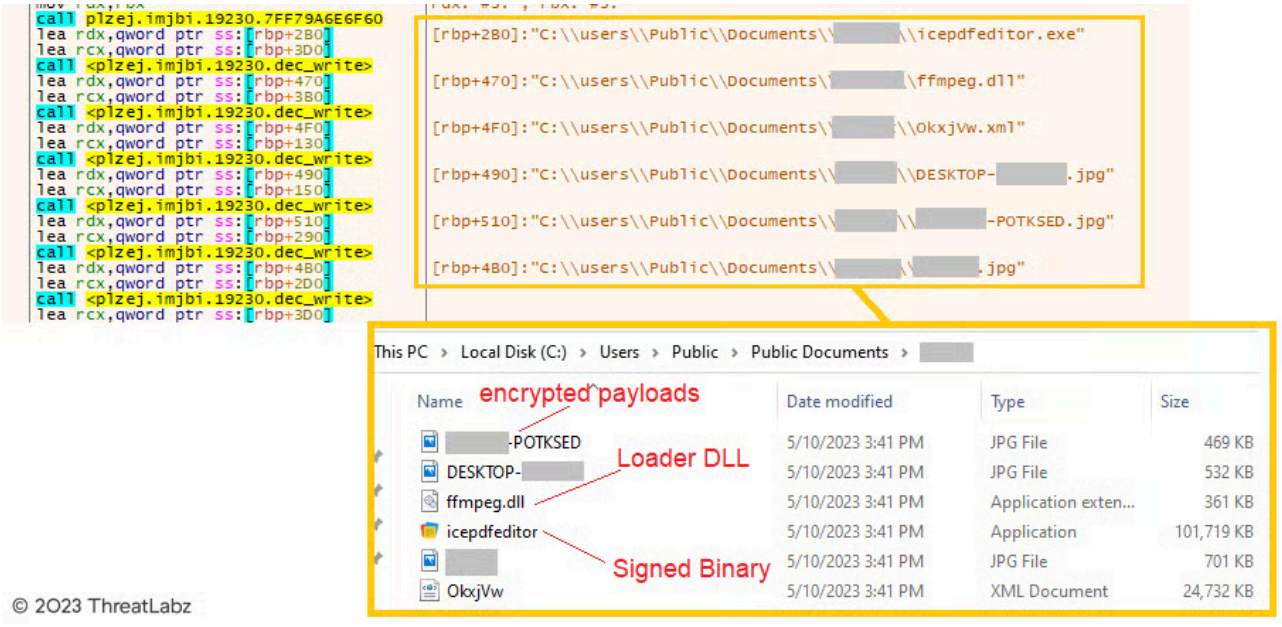

Figure 11 - Multiple payloads downloaded in the public documents folder.

In Figure 11, it can also be observed that the encrypted payloads have dynamically generated filenames based on the computer name, username, etc.. The Loader DLL, "**ffmpeg.dll**," has its filename decrypted using the string decryption process. The signed binary, "**icepdfeditor**," is randomly selected from the list of file names dynamically, and the extensions are decrypted accordingly. Additionally, the downloader creates a configuration file with the **ini** extension named after the computer name in the Public Documents folder, containing details about the encrypted payloads.

Towards the end of this process, the downloader generates a batch script in the "**AppData\Roaming**" directory, named after the computer name. Upon execution, this script restarts the system after a 10-second timeout, as depicted in Figure 12 below. The content of the batch script is decrypted using the string decryption function.

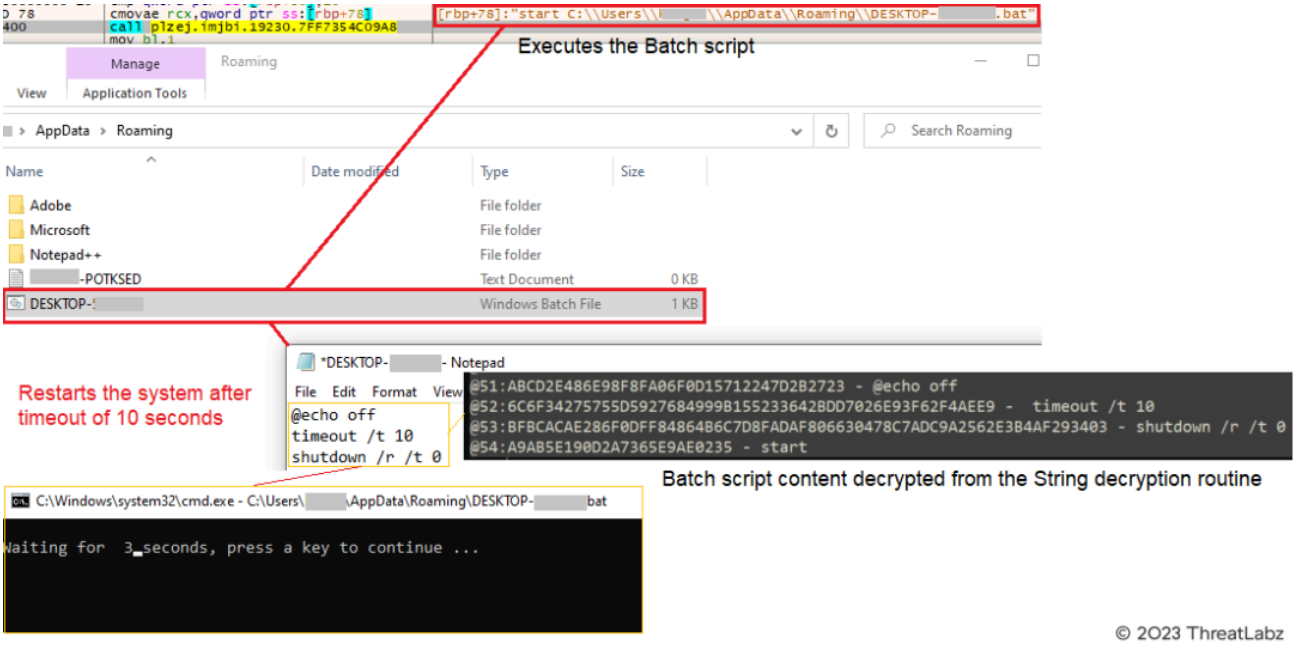

Figure 12 - Evades sandbox & executes the LNK file in the startup folder by restarting the system.

The system reboot serves to evade sandbox detection since the malicious actions occur only after the reboot. Upon restarting, the shortcut (.LNK) file, "**icepdfeditor.lnk**," in the startup folder is automatically executed, triggering the execution of "**icepdfeditor.exe**" from the Public Documents folder. "**icepdfeditor.exe**" is a valid signed executable by "ZOHO Corporation Private Limited," downloaded alongside the other payloads. Figure 13 shows the execution of "**icepdfeditor.exe**."

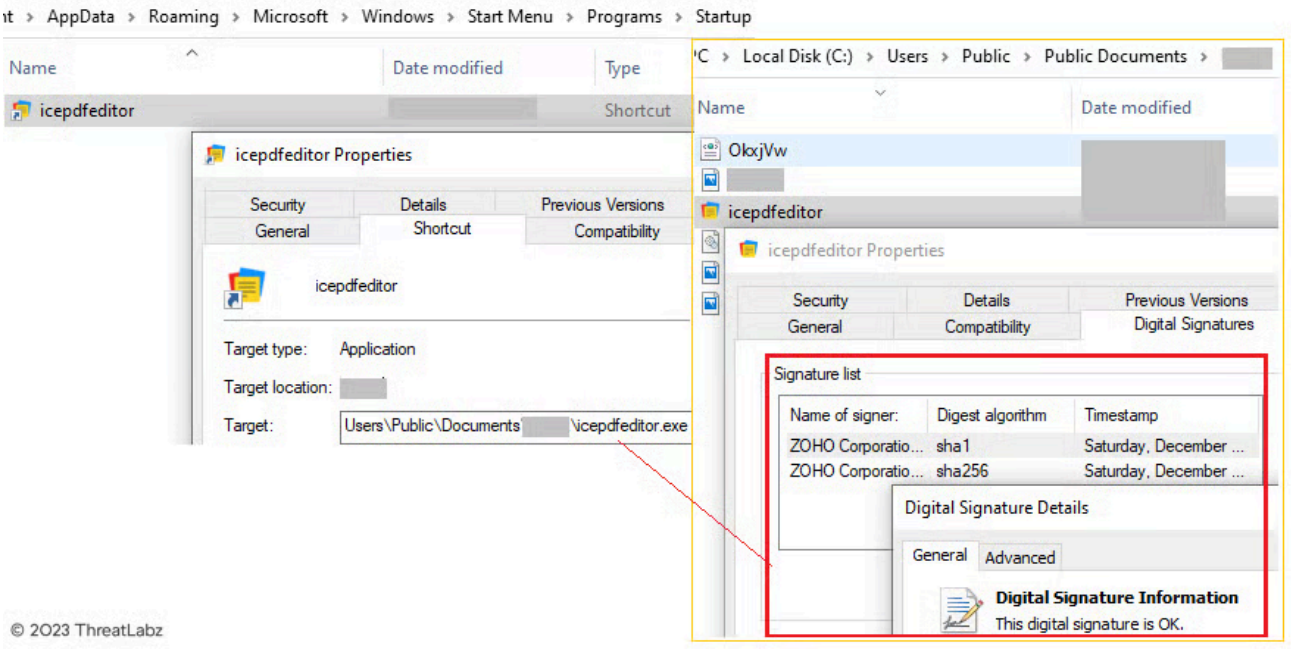

Figure 13 - Signed binary by ZOHO Corporation downloaded alongside malicious payloads.

Upon final execution, the signed binary, "**icepdfeditor.exe**," sideloads the malicious Krita Loader DLL, "**ffmpeg.dll**," from the current directory "**C:\Users\Public\Documents**" taking advantage of the Windows Search and Load order to load the malicious Loader DLL instead of the legitimate DLL, as illustrated in Figure 14.

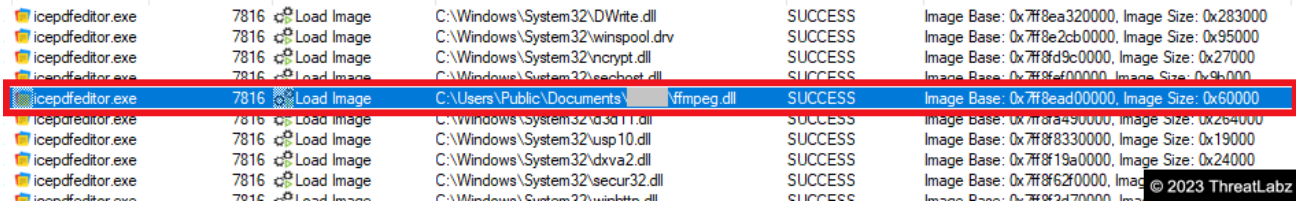

Figure 14 - Signed Binary "icepdfeditor.exe" sideloads the malicious Krita Loader DLL "ffmpeg.dll".

Stage-2: Krita Loader DLL (ffmpeg.dll)

PDB Path: F:\Trabalho_2023\OFF_2023\DLL_Start_OK\x64\Release\DLL_Start_OK.pdb

In the analysis of the **Krita Loader DLL** (**ffmpeg.dll**), it is observed that the DLL reads encoded data from the **.jpg** file. This encoded data is then dynamically reversed and decoded using a replacement routine. The replacement routine replaces special characters with specific characters based on an algorithm, for example, replacing "!" with "A".

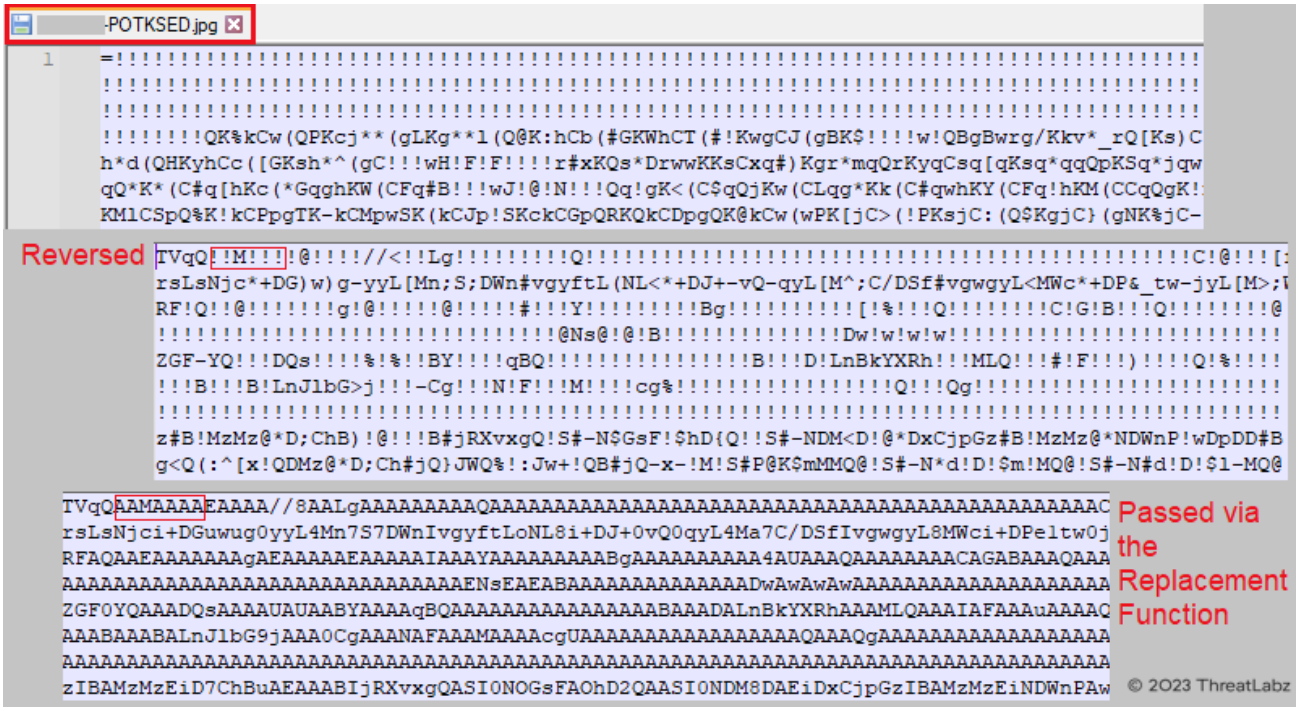

Figure 15 - Showcases the decoding process of the DLL, involving reverse and replace functions.

As depicted in the preceding screenshot, the data is subsequently subjected to base64 decoding, resulting in the formation of a PE file. This PE file is then written to the temporary directory, utilizing a randomly generated file name, as illustrated in Figure 16.

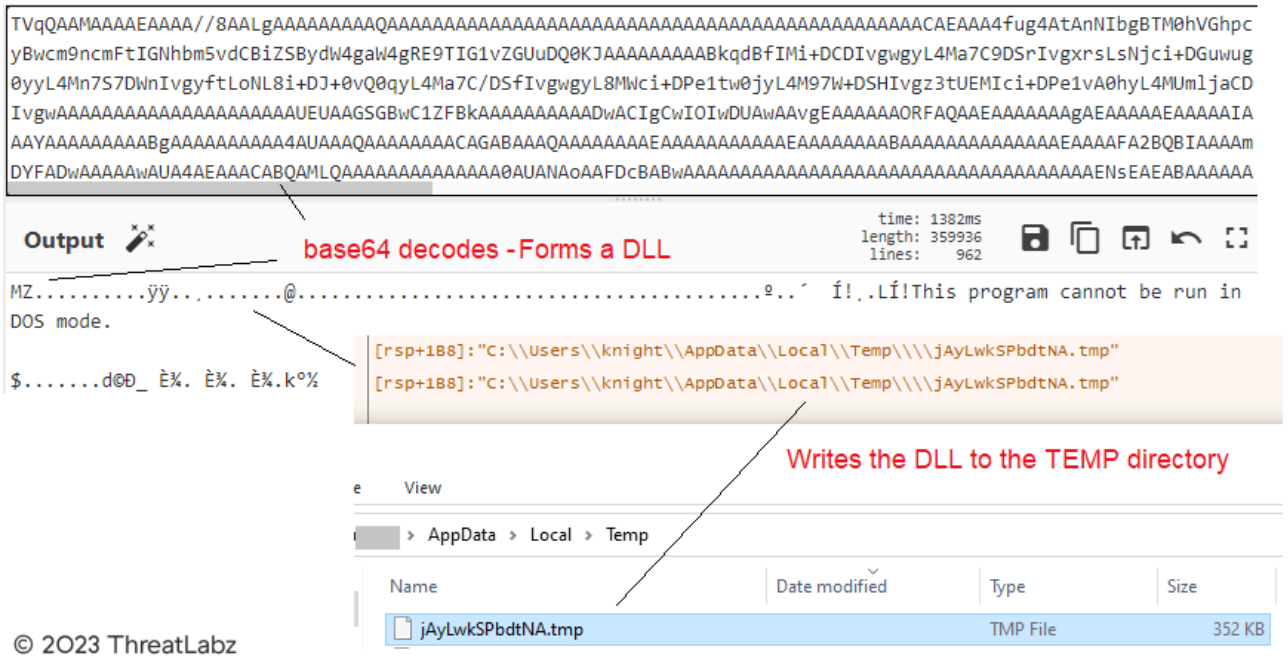

Figure 16 - Demonstrates the process of decoding the DLL through Base64 decoding.

Subsequently, the decoded **InjectorDLL** is loaded into memory by the **Krita Loader DLL** using the **LoadLibraryA()** function. Control is then transferred by retrieving the address of the export function "TEMP" through the **GetProcAddress()** function.

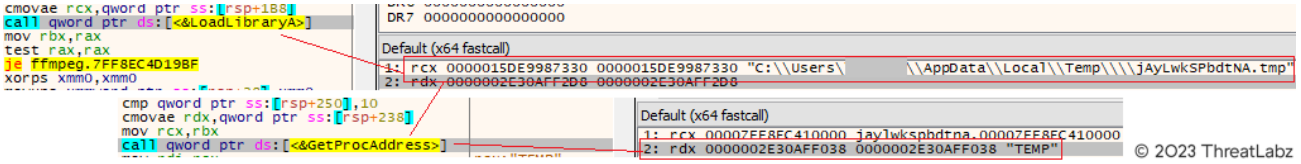

Figure 17 - Illustrates the loading of the InjectorDLL via the LoadLibraryA() function.

Stage-3: InjectorDLL Module

PDB Path: F:\Trabalho_2023\OFF_2023\DLL_Start_IN\x64\Release\DLL_START_IN.pdb

Once the **InjectorDLL** is loaded, it proceeds to read encoded data from another **.jpg** file. Similar to the **Krita Loader DLL**, the **InjectorDLL** dynamically reverses and decodes the data using a replacement routine that replaces special characters with specific characters based on a predefined algorithm. Subsequently, the data undergoes base64 decoding, resulting in the formation of the **ElevateInjectorDLL** module. This module is then injected into the remote process "**explorer.exe**" using a sequence of functions: **OpenProcess**, **VirtualAllocEx**, **WriteProcessMemory**, and **CreateRemoteThread**. The screenshot shown in Figure 18 below illustrates this injection process.

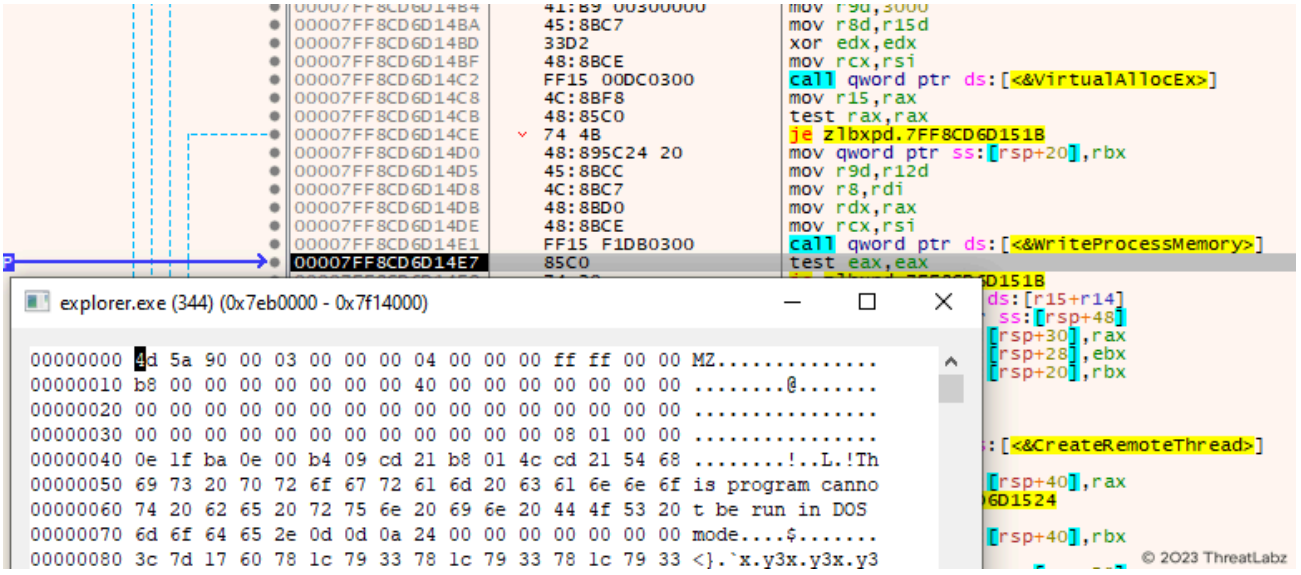

Figure 18 - Demonstrates the injection of the ElevateInjectorDLL module into the remote process "explorer.exe."

Stage-4: ElevateInjectorDLL Module

PDB Path: F:\Trabalho_2023\OFF_2023\DLL_Start_UP\x64\Release\DLL_Start_UP.pdb

Once injected into the **explorer.exe** process, the **ElevateInjectorDLL** module performs initial checks. It verifies whether the parent process is either "**explorer.exe**" or "**winlogon.exe**" and checks if the mutex "**explorer**" or "**winlogon**" has already been created using the **OpenMutexA()** function. If the conditions are met, the module creates the mutex "**explorer**" or "**winlogon**" based on the parent process as shown in Figure 19 below. Subsequently, it executes the main routine to carry out further actions.

```
if ( (unsigned __int8)check_proc_explorer() && !OpenMutexA(0x1F0001u, 0, "explorer") )
{
  CreateMutexA(0i64, 0, "explorer");
  sub_180016000();
}
if ( (unsigned __int8)check_proc_winlogon() && !OpenMutexA(0x1F0001u, 0, "winlogon") )
{
  CreateMutexA(0i64, 0, "winlogon");
  sub_180016000();
}
return 0i64;
```


Figure 19 - Showcases the process of checking the parent process, specifically verifying if it is either "explorer" or "winlogon".

This technique ensures that the module evades sandboxes by verifying the parent process. If the parent process does not match the expected value, the malicious code remains dormant and is not executed.

In this particular scenario, as the parent process is "**explorer.exe**," the main routine is executed. Within this routine, specific strings are base64 decoded. These strings contain the server address **(191[.]252[.]203[.]222/Up/indexW.php)** and the paths of the target processes (**explorer.exe** and **svchost.exe**) where the subsequent injection stages will take place.

Additionally, the **ElevateInjectorDLL** checks whether the process is elevated. In this case, as the process is not elevated, the DLL reads and decrypts another JPG file from the Public Documents folder. This decryption process forms the next stage module called "**BypassUAC**." Subsequently, the module performs process hollowing to inject the **BypassUAC** module into another **explorer.exe** process that was previously spawned in a suspended state.

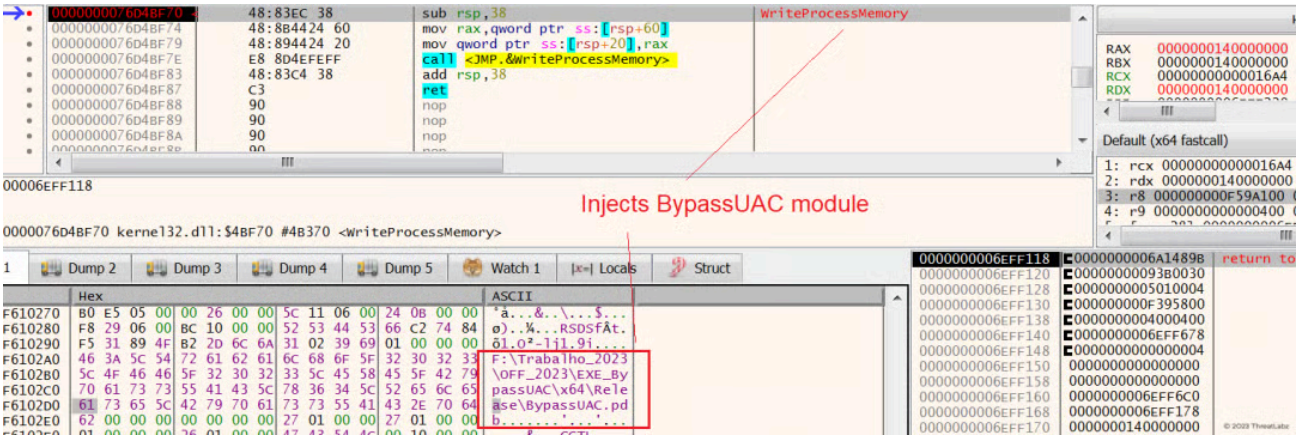

Figure 20 - Demonstrates the injection of the BypassUAC module into the explorer.exe process when the process is not elevated.

Stage-5: BypassUAC Module

PDB Path: F:\\Trabalho_2023\\OFF_2023\\EXE_BypassUAC\\x64\\Release\\BypassUAC.pdb

The **BypassUAC** Module is responsible for performing User Account Control (UAC) bypass, enabling the execution of the Downloader module with administrator privileges.

When the previously injected **BypassUAC** Module is executed within the remote process **explorer.exe**, it exits without executing the main routine under two conditions. Firstly, if the mutex "**explorer**" is not created, and secondly, if the

mutex "**bypass**" is already created. However, if these conditions are not met, the module proceeds to create the "**bypass**" mutex before continuing its execution.

```
if ( !OpenMutexA(0x1F0001u, 0, "explorer") || OpenMutexA(0x1F0001u, 0, "bypass") )
  exit(0);
CreateMutexA(0i64, 0, "bypass");
```


Figure 21 - Depicts the process of opening and creating mutexes.

In the context of UAC bypass, the malware leverages the COM Elevation Moniker "**Elevation:Administrator!new:**" along with specific elevated COM Objects. The purpose is to bypass the User Account Control (UAC) restrictions and gain elevated privileges on the system. To achieve this, the malware utilizes the CLSID {**3AD05575-8857-4850-9277-11B85BDB8E09**}, which provides functionalities related to copy, move, rename, delete, and link operations. Additionally, the CLSID {**BDB57FF2-79B9-4205-9447-F5FE85F37312**} is employed, specifically designed for the installation of Internet Explorer add-ons. By exploiting these elevated COM Objects, the malware aims to elevate its privileges and carry out malicious activities without being hindered by UAC restrictions.

```
        CoGetObject(
          L"Elevation:Administrator!new:{3AD05575-8857-4850-9277-11B85BDB8E09}",
          (BIND_OPTS *)pBindOptions,
          &riid,
          &ppv) < 0) )
  {
    CoUninitialize();
LABEL_267:
    invalid_parameter_noinfo_noreturn();
  }
  v19 = CoGetObject(
          L"Elevation:Administrator!new:{BDB57FF2-79B9-4205-9447-F5FE85F37312}",
          (BIND_OPTS *)pBindOptions,
          &stru_140073D58,
          &v172) < 0;
```


Figure 22 - Illustrates the UAC bypass technique achieved through the use of the COM Elevation Moniker.

In the process of UAC bypass, the malware utilizes the Copy/Move/Rename/Delete/Link COM Object. This COM Object serves the purpose of copying the "cmd.exe" file from the System32 Folder to the Temp directory with administrator privileges. The copied file is then renamed as [1]bdeunlock.exe. This technique allows the malware to manipulate system files and execute commands with elevated privileges, facilitating further malicious activities.

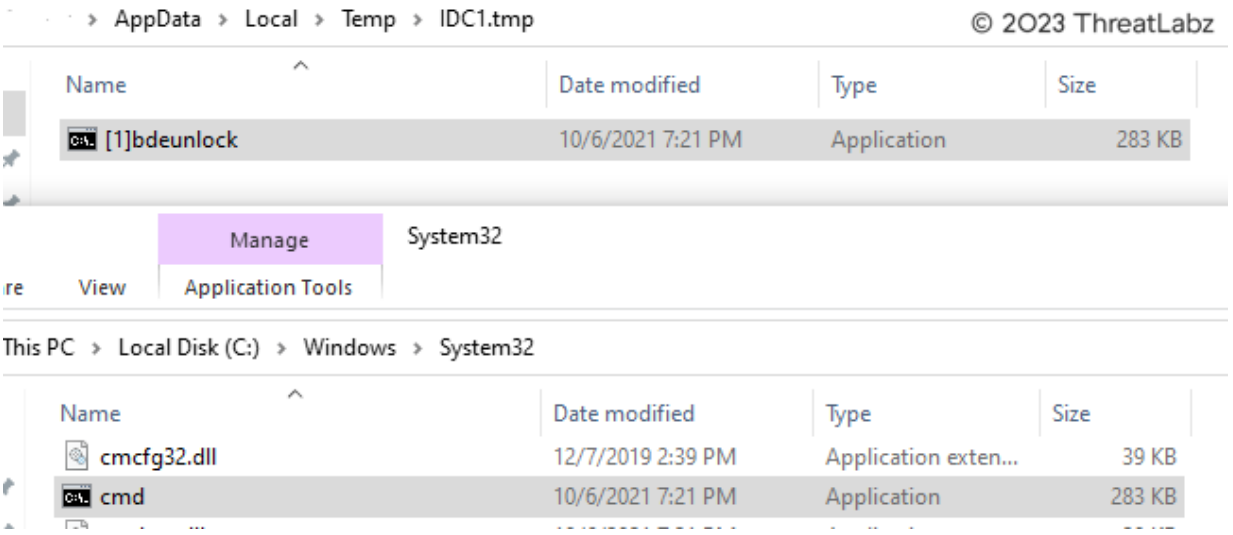

Figure 23 - Depicts the operation of copying the "cmd.exe" file into the Temp directory with administrator privileges using a COM Object.

Moreover, the auto-elevating Internet Explorer Add-on Installer, known as "**IEInstal.exe**," is triggered through the COM Object. This action aims to execute the signed binary "**icepdfeditor.exe**" with elevated privileges by spawning a new process named **[1]bdeunlock.exe**. The process is launched with specific arguments, namely "**/C start** ," as indicated in Figure 24. This technique allows the malware to execute the signed binary with elevated permissions, enabling it to carry out malicious activities on the system.

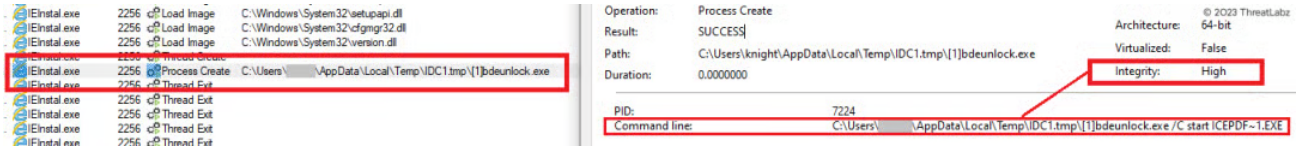

Figure 24 - Demonstrates the UAC bypass executed by the Internet Explorer Add-On Installer. This bypass enables the execution of the Krita Loader DLL with elevated privileges.

Consequently, the signed binary is executed with elevated privileges, facilitating the sideloading of the Krita Loader DLL onto the machine with administrative privileges.

Once the Krita Loader DLL is sideloaded with elevated privileges, it follows the same routine as previously discussed. However, in this instance, the ElevateInjectorDLL module, which previously injected the BypassUAC module, verifies whether it has elevated privileges. If elevated privileges are present, the module decrypts the final TOITOIN Trojan and injects it into the remote process "**svchost.exe**," as depicted in the screenshot provided in Figure 25 below.

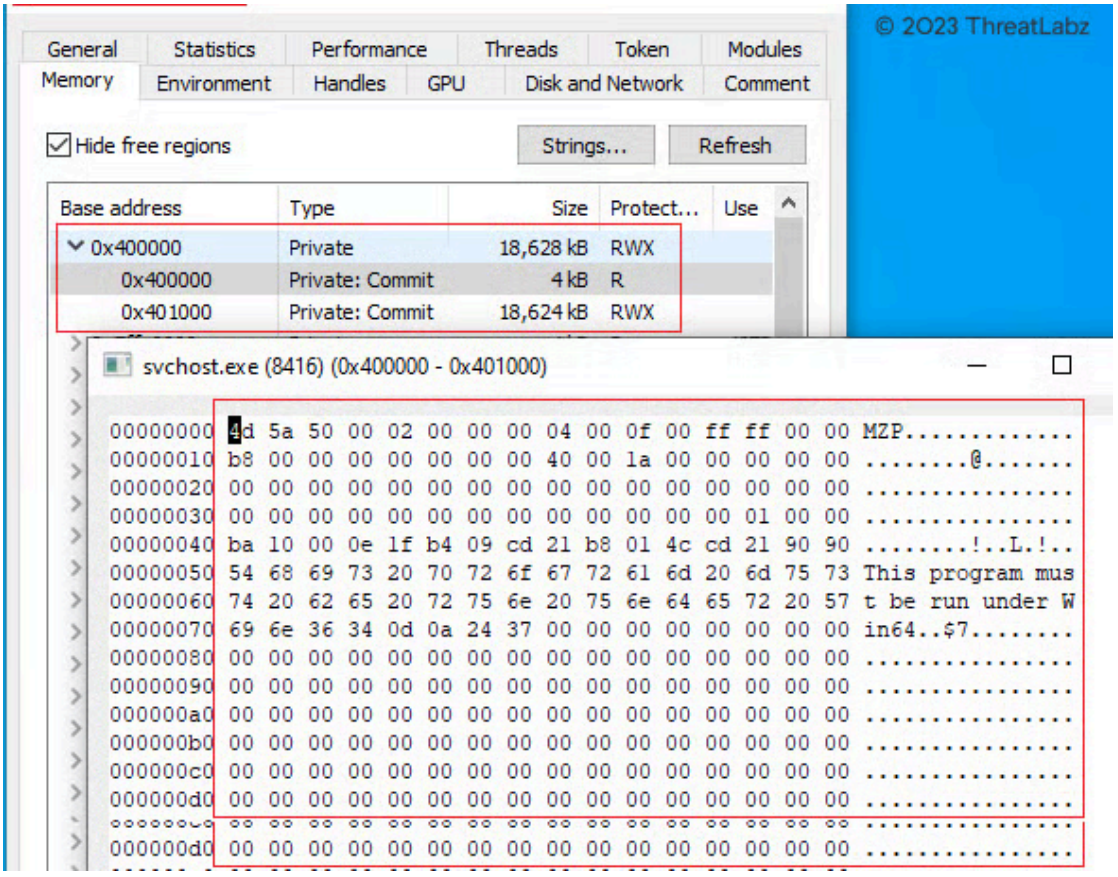

Figure 25 - Displays the injection of the TOITOIN Trojan into the svchost.exe process.

Stage-6: TOITOIN Trojan

The **ElevateInjectorDLL** injects the new Latin American Trojan, **TOITOIN**, into the remote process "**svchost.exe**." Upon execution, the Trojan first reads the encoded **.ini** configuration file that was previously written in the Public Documents folder by the Downloader module, as the captured screenshot in Figure 26 below shows.

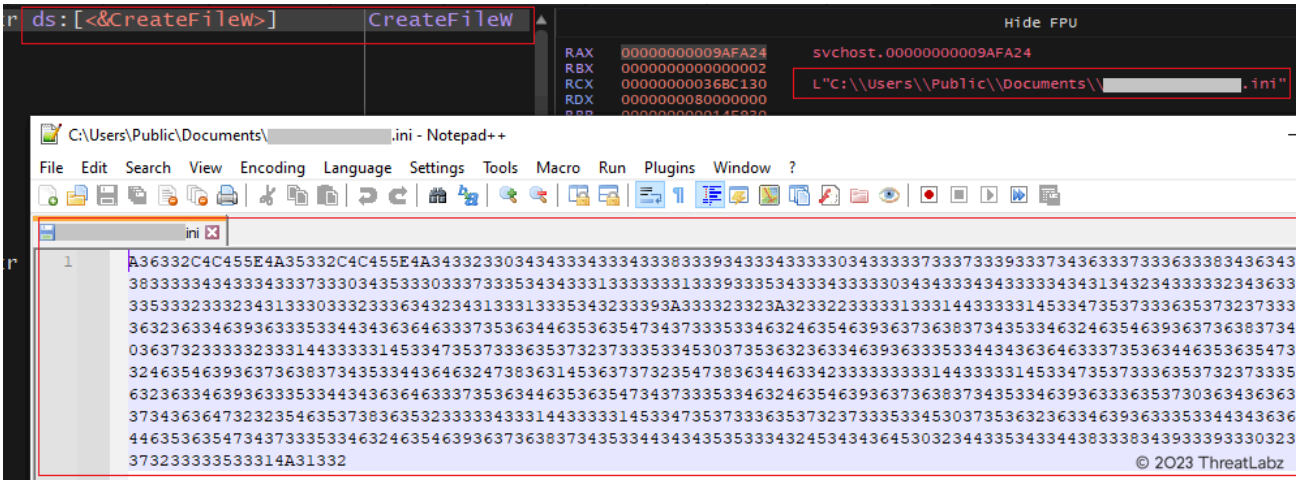

Figure 26 - Demonstrates the process of reading the INI configuration file.

Below, Figure 27 reveals the decoding process of the hex blob within the INI Configuration file. The hex blob is reversed and converted to ASCII format, unveiling its original content.

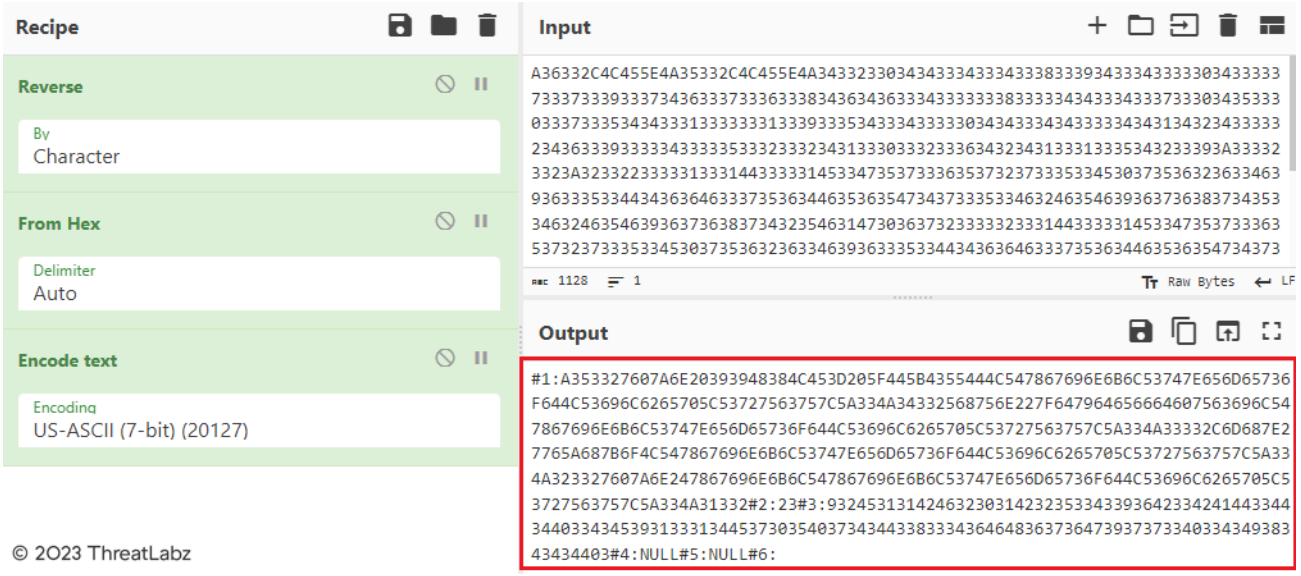

Figure 27 - Showcases the decoding process of the INI Configuration File.

Each of the # hex strings undergoes further decoding using the same logic. In the Figure 28 screenshot below, the decoded #1 hex string reveals the complete path and file name of the multiple payloads that were downloaded by the Downloader from **http[:]//cartolabrasil[.]com**.

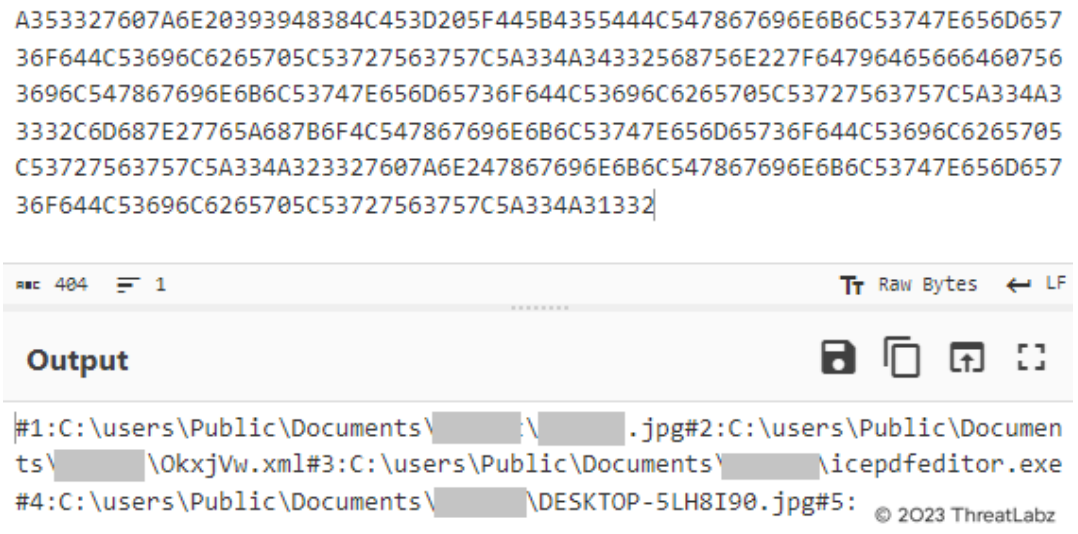

Figure 28 - Depicts the continued decoding process of the INI Configuration File.

Additionally, the #3 hex blob undergoes decoding using the aforementioned logic, resulting in another hex blob. This hex blob is then decrypted using a custom XOR logic, such as applying the XOR operation with the first two bytes (**0D44 -> (0x44 ^ 0x31) - 0x0D = "h"**). ThreatLabz researchers developed a decryptor specifically for the INI Configuration file, as shown below in Figure 29.

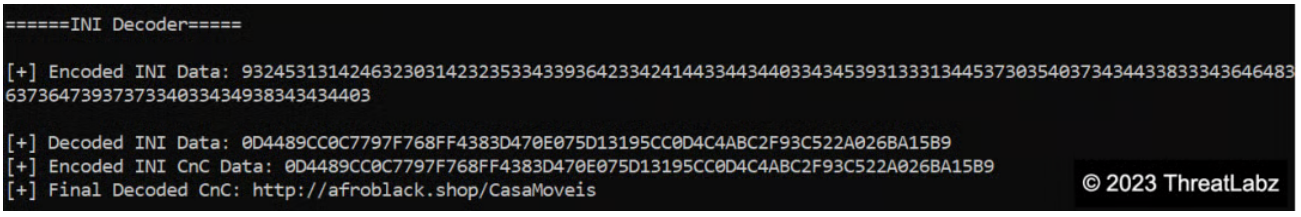

Figure 29 - Showcases the INI Configuration Decryptor, which reveals the Command & Control (C&C) URL.

In Figure 29, the decrypted value extracted from the INI Configuration file reveals the Command & Control (C&C) URL: **http[:]//afroblack[.]shop/CasaMoveis**. However, during the analysis of a different corrected sample, a distinct C&C server was discovered: **http[:]//contabilidademaio[.]servebeer[.]com/Robs/counter**.

The final backdoor decrypts various strings, including C&C URLs and other crucial information, using a custom XOR decryption logic based on the INI configuration values of "1" or "2":

- 5EBCDD2160A3E060F95B, 4644454647484786DF61 - /0202.php
- B9D91B5B9FC2C1035AFB, 07076684E60A094881C0 - /POST.php
- 4980C50B4AB5D534A72BBD144483D7084A9D21B3164E8B88DA7BD9, D40CB117B6C1C1C115B629A13291C516B82AAD3E80D87493C70642 - http[:]//bragancasbrasil[.]com
- 37AE12B71962A0FE01097390F01861BEC0C9C6CC - http://179[.]188[.]38[.]7
- 1D7E85858489898A8B8C8E87EA0B6588E6017E96, 95F61C7A9BFE1F60A1E2277EE3027AFF7EF674AC - 26/04/2023(TOITOIN)

Some of the decrypted strings include paths to payloads, browser installations, and relevant dates.

Additionally, a substantial encrypted hex blob is decrypted using the same custom XOR decryption logic. This hex blob consists of strings related to file paths, browser types, and timestamps. The TOITOIN Trojan employs these decrypted strings in its operation:

- @36:C91756F9588E30A03F6E85DF7DD376E3094988D862B33249BC284E9E20A2359DC81DBD0157B42B92C6 - \Program Files\Topaz OFD\Warsaw\core.exe
- @37:86D025A529BF2C - [Core]
- @38:5F8BF014BD2A - [64x]
- @39:204A4C4FF665 - [32x]
- @45:679F3E9C3590C41062FB5FFF5CF07C - Google\Chrome\
- @46:314F8F3B9533AE217087C1055BF051F864FB54F76CE362EB2BA12BA1 - Mozilla Firefox\firefox.exe
- @47:748CCB71E76BE87FC2D3289133AE2F90C406538DC30D4C89C90D4340943C90 - Internet Explorer\iexplore.exe
- @48:1051F2538BCC1FBD006D8DCC034494C11FB116B528A13699F66DD40B - Programs\Opera\launcher.exe
- @49:5C9D21A1389BCD0A4C99D97BD175E775D577ED518380D47CD0 - Programs\Opera\opera.exe
- @50:47A53E9032903595CA0E5B91C41DB32153F2528FC91C4CF16AE866F350F269FC558B88DC64F8 - Microsoft\Edge\Application\msedge.exe
- @51:AB364E82CB0D4886C91EA1CF - [Iexplorer]
- @52:1541B52F9031AF24B0 - [Chrome]
- @53:2EB8379723AC27A235A3 - [Mozzila]
- @54:5F8BEB6DE367F966 - [Opera]
- @55:5882F66AE077E5 - [Edge]

The Trojan fetches the Windows version by querying the **ProductName** registry key value, retrieves the environment variable **%homedrive%** and the path to the Program Files directory, and determines whether the system is 32-bit or 64-bit.

Based on the installed browsers, including Chrome, Edge, Opera, Mozilla Firefox, and Internet Explorer, the Trojan assigns specific values to each browser. It also checks for the presence of the **Topaz OFD** - Protection Module at the specified path and sets the value "[Core]" accordingly.

Furthermore, another encrypted hex blob is decrypted, containing strings related to certain variables, such as **ClienteD.php?1=, - (hyphen), Versao_DLL(, Data(, dd/mm/yyyy, and hh:mm:ss**:

- @36:F5639735AF24A329BF3552F36DEC1D7F8D - \ClienteD.php?1=
- @37:77A6E232 - -
- @38:D7C6C2D31BB115B92AA8394CA9C4DD - Versao_DLL(
- @39:2170ACFD73E56BFD17 - Data(
- @40:F266FB1AB61575DF68D07B - dd/mm/yyyy
- @41:EB65FC06429EE96CEE - hh:mm:ss

Leveraging these decrypted strings, the Trojan assigns values to variables like AA1, AA2, AA3 & AA4, AA5, and AA10, using the previously decrypted encoded format. For example, AA1 represents the computer name, AA2 represents the Windows version, AA3 & AA4 represent the installed browsers, AA5 represents the bit value (32x or 64x), and AA10 represents the date (26/04/2023, in this case).

- AA1 - 0393948384C453D205F445B4355444 --> DESKTOP-******* (Computer Name)
- AA2 - E6F6964716365746540203130237777F646E69675 --> Windows 10 Education (Windows Version)
- AA3 & AA4 - D556764654B5D5275627F6C607875694B5 --> [Iexplorer][Edge] (Installed browsers & protection module)
- AA5 - D5874363B5 --> [64x] (Bit)
- AA10 - 92E494F44594F4458233230323F24303F26323 --> 26/04/2023(TOITOIN)

Analyzing and understanding these decrypted strings allows for a better understanding of the TOITOIN Trojan's configuration, the system it operates on, and the communication channels it utilizes for command and control.

TOITOIN utilizes the decrypted strings in the following manner in order to gather the system & browser information:

1. It fetches the Windows version by querying the **ProductName** value from the registry key: **SOFTWARE\Microsoft\Windows NT\CurrentVersion.**

2. It retrieves the environment variable **%homedrive%** using **GetEnvironmentVariableW**, which usually corresponds to the **C:\ drive**.

3. It determines whether the system is 32-bit or 64-bit and sets the value to [64x] or [32x] accordingly.

4. It checks if specific web browsers are installed on the system by verifying the existence of corresponding folders and files. The checked browsers include Chrome, Edge, Opera, Mozilla Firefox, and Internet Explorer.

5. Based on the installed browsers, it assigns specific values for each browser:

- [**Iexplorer**] for Internet Explorer
- [**Chrome**] for Chrome
- [**Mozzila**] for Mozilla Firefox
- [**Opera**] for Opera
- [**Edge**] for Microsoft Edge

6. It checks whether the **Topaz OFD** - Protection Module is installed at the path **\Program Files\Topaz OFD\Warsaw\core.exe**. If the module exists, it sets the value "[**Core**]".

By leveraging these decrypted strings and performing these checks, the TOITOIN Trojan adapts its behavior based on the system's Windows version, installed browsers, and the presence of the **Topaz OFD** - Protection Module.
