A New Multi-Stage Attack Targeting LATAM

zscaler.com/blogs/security-research/toitoin-trojan-analyzing-new-multi-stage-attack-targeting-latam-region

Summary

Discover the intricate layers of a new sophisticated and persistent malware campaign targeting businesses in the LATAM region delivering the TOITOIN Trojan. Delve into the multi-stage attack methodology, from deceptive phishing emails to custom-built modules, as we dissect its techniques and shed light on its impact. Gain valuable insights into the evolving threat landscape and learn how organizations can fortify their defenses against this emerging Latin American cyber threat. Stay one step ahead with this in-depth analysis of TOITOIN and safeguard your business against advanced malware attacks.

Introduction

In the ever-evolving landscape of cyber threats, researchers from Zscaler ThreatLabz have recently uncovered a concerning development: a new targeted attack campaign striking businesses in the Latin American (LATAM) region. This sophisticated campaign employs a trojan that follows a multi-staged infection chain, utilizing specially crafted modules throughout each stage. These modules are custom designed to carry out malicious activities, such as injecting harmful code into remote processes, circumventing User Account Control via COM Elevation Moniker, and evading detection by Sandboxes through clever techniques like system reboots and parent process checks. The ultimate payload of this campaign is a new Latin American Trojan called TOITOIN, which incorporates a unique XOR decryption technique to decode its configuration file. Once decrypted, the trojan gathers crucial system information, as well as data pertaining to installed browsers and the Topaz OFD Protection Module, before sending it to the command and control server of the attackers in an encoded format. This blog post provides an in-depth analysis of this emerging malware campaign and its corresponding infection chain. Read on to learn more about this alarming threat.

Key Takeaways and Observations

1. The TOITOIN malware campaign targets businesses in the LATAM region, utilizing sophisticated techniques and multi-stage infection chains.

2. By leveraging Amazon EC2 instances, the threat actors evade domain-based detections, making it more challenging to detect and block their activities.

3. The analyzed campaign employs a series of custom-developed modules, including:

- Downloader Module: Downloads further stages, evades sandboxes through system reboots, and maintains persistence using LNK files.
- Krita Loader DLL: Sideloaded via a signed binary, it loads the InjectorDLL module.
- InjectorDLL Module: Injects the ElevateInjectorDLL into the remote process (explorer.exe).
- ElevateInjectorDLL Module: Evades sandboxes, performs process hollowing, and injects either the TOITOIN Trojan or BypassUAC module based on process privileges.
- BypassUAC Module: Utilizes COM Elevation Moniker to bypass User Account Control and execute the Krita Loader with administrative privileges.

4. The final payload, the TOITOIN Trojan, employs custom XOR decryption routines to decode the configuration file containing the Command & Control server's URL. It transmits encoded system information and details about installed browsers and the Topaz OFD Protection Module to the C&C server. In the absence of the configuration file, the information is sent via a POST request using curl.

5. Zscaler's Zero Trust Exchange provides strong protection against sophisticated malware campaigns like TOITOIN, leveraging its zero trust model, advanced threat intelligence, cloud-native architecture, and granular access controls to ensure the security and integrity of customer environments.

TOITOIN Infection Chain

In May 2023, diligent threat hunters within the Zscaler cloud, recognized as the world's largest security cloud, made a significant breakthrough. Their discovery involved the identification of numerous malware samples concealed within compressed ZIP archives. All of the identified archives were found to be hosted by Amazon EC2, as shown in Figure 1 below.

Upon closer examination and thorough analysis of related malware samples obtained from Zscaler cloud, it became evident that a novel campaign had emerged. This campaign, named TOITOIN introduced a series of custom-built malwares specifically designed to target businesses operating within LATAM. Commencing in May 2023, this malicious endeavor continues to pose an ongoing threat, demanding immediate attention and comprehensive understanding from defenders.

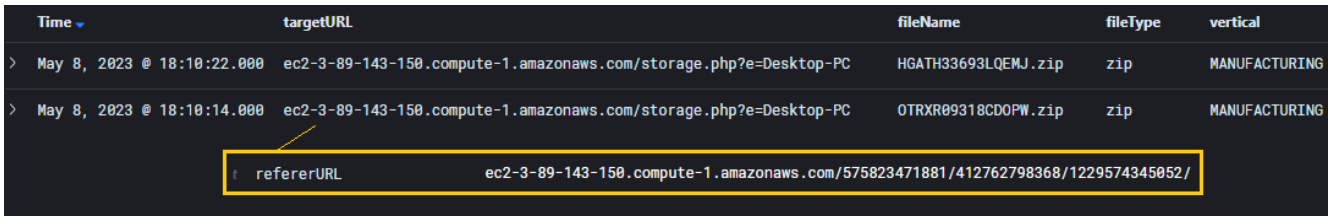

Figure 1 - Researchers discover suspicious ZIP archives hosted on Amazon EC2 during threat hunting activities in Zscaler cloud.

The TOITOIN malware infection chain, shown in Figure 2 below, employed in this targeted campaign follows a well-crafted sequence, starting with an initial compromise phishing email.

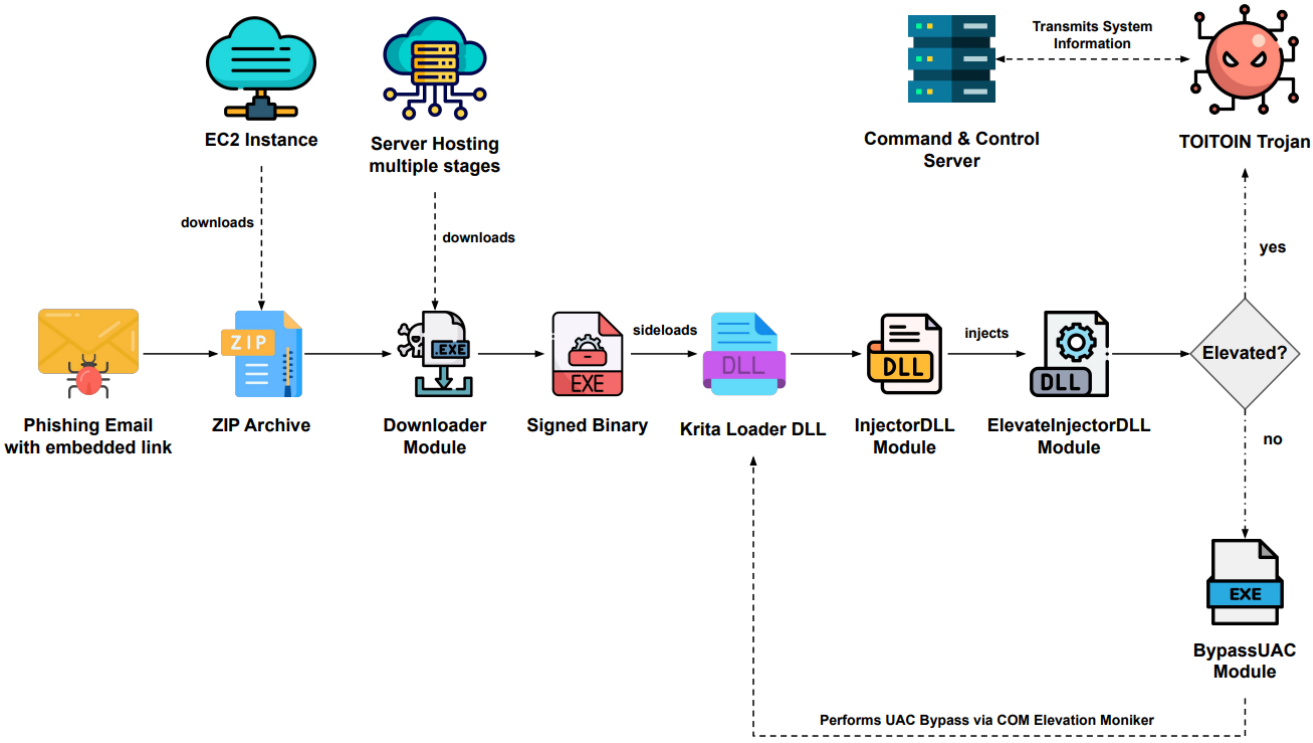

Figure 2 - The multi-staged infection chain.

In the context of this campaign, Figure 3 offers a glimpse into the deceptive email crafted with the specific intent to entrap a prominent Investment Banking company in Latin America. Carefully designed, the email capitalizes on a Payment Notification Lure, alluring the recipient to engage further by clicking on a button labeled 'Visualizar Boleto,' which translates to 'View Invoice' in English. This strategic choice of wording aims to evoke a sense of urgency and entice the target to explore the contents of the email, ultimately falling into the trap set by the threat actors.

Figure 3 - Screenshot of phishing email sent by threat actors behind this TOITOIN campaign.

Upon clicking the button in the phishing email, the user unwittingly initiates a chain of events. The URL **http[:]//alemaoautopecas[.]com/1742241b/40c0/df052b5e975c.php?hash=aHR0cHM6Ly9teS5ub2lwLmNvbS9keW5hbWljLWRucw** is then opened, serving as an intermediary redirect. Subsequently, the victim's browser is redirected once again, this time to the address **http[:]//contatosclientes[.]services/upthon**. It is at this point that the malicious ZIP archive is stealthily downloaded onto the victim's system, and begins infiltrating their defenses.

Notably, several other domains have been identified as vehicles for delivering these malicious ZIP archives. These domains include:

- atendimento-arquivos[.]com
- arquivosclientes[.]online
- fantasiacinematica[.]online

By diversifying the delivery channels, the threat actors behind this campaign have effectively evaded detection based on domain reputation. However, it is worth mentioning that the malicious ZIP archives were hosted on an Amazon EC2 instance as shown below in Figure 4. Leveraging the capabilities of Amazon's cloud infrastructure, the attackers have

managed to stay one step ahead, shielding their activities from domain-based detection mechanisms.

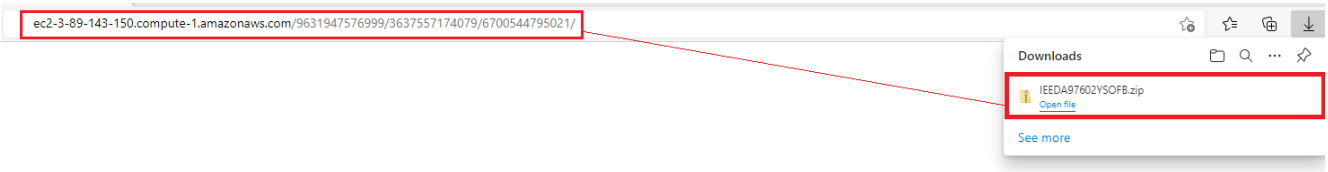

Figure 4 - Malicious ZIP archive downloaded from the Amazon EC2 instance.

To further obfuscate their intentions, the threat actors adopted a dynamic approach to naming the ZIP archives. With each download, the server generates a new and randomly generated file name, thwarting simplistic attempts at detection based on static file naming patterns. This tactic adds an additional layer of complexity to the campaign, making it more challenging to identify and mitigate the threat effectively.

Within the ZIP archive labeled as "**HGATH33693LQEMJ.zip**," a malicious executable file titled "**HCEMH.hqdrm.63130.exe**" resides. This specific file operates as the designated downloader module, orchestrated by the threat actors to initiate the retrieval of numerous payloads from the server under their control. Alongside this primary function, the downloader module, analyzed in the next section, also encompasses a range of evasion techniques, strategically implemented to circumvent detection and hinder security measures.

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.

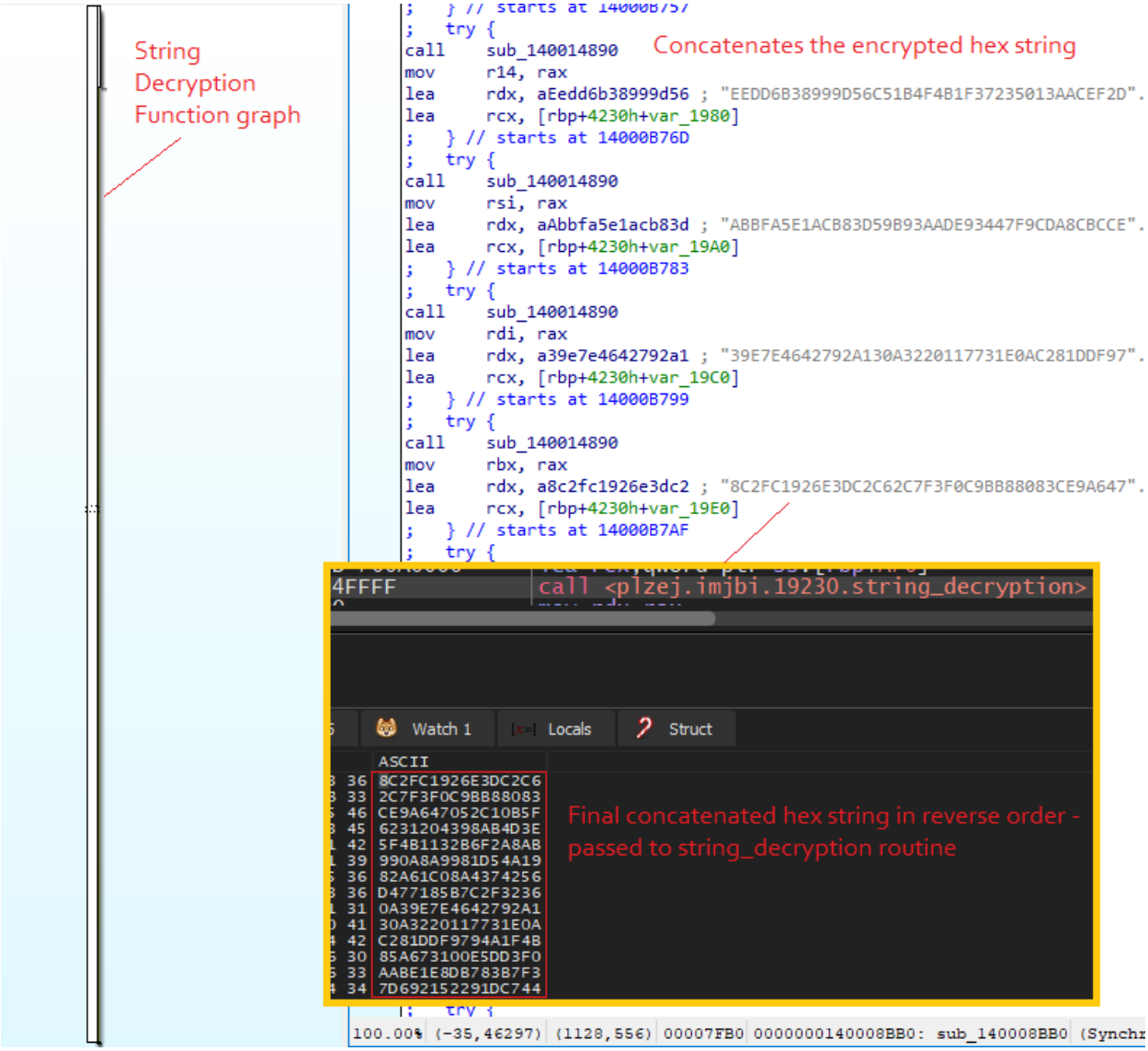

Figure 5 - Illustrates the String decryption routine, showcasing the concatenation process.

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: <hex_string>**." 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.

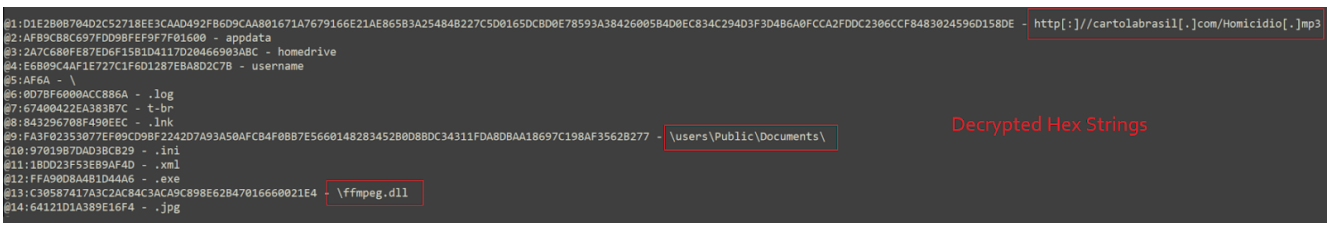

Figure 6 - Overview of the string decryption routine, focusing on the decryption of the downloader URL.

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 "**<reverse_of_computer_name>.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	

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.

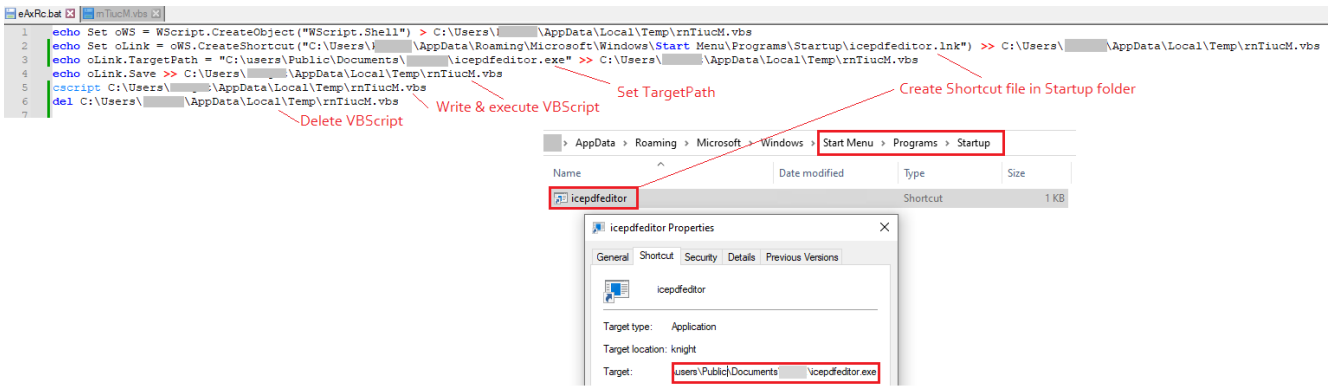

Figure 8 - Batch script creating LNK file in the StartUp folder for persistence.

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.

```
@1:D1E2B0B704D2C52718EE3CAAD492FB6D9CAA801671A7679166E21AE865B3A25484B227C5D0165DCBD0E78593A38426005B4D0EC834C294D3F3D4B6A0FCCA2FDDC2306CCF84830
24596D158DE - http[:]//cartolabrasil[.]com/Homicidio[.]mp3
@31:BF8C07009147B6547E6DEB19 - /1.mp3
@32:320146416CBA7694D1F229DB - /2.mp3
@33:5162F2F58D5B3FDD44772AD8 - /3.mp3
@34:B083787FE7311CFE41029A68 - /4.mp3
@35:0B383E399C4A20C2F5A6D92B - /5.mp3
@36:1122333432E4AB49E380E416 - /6.mp3
```

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**.

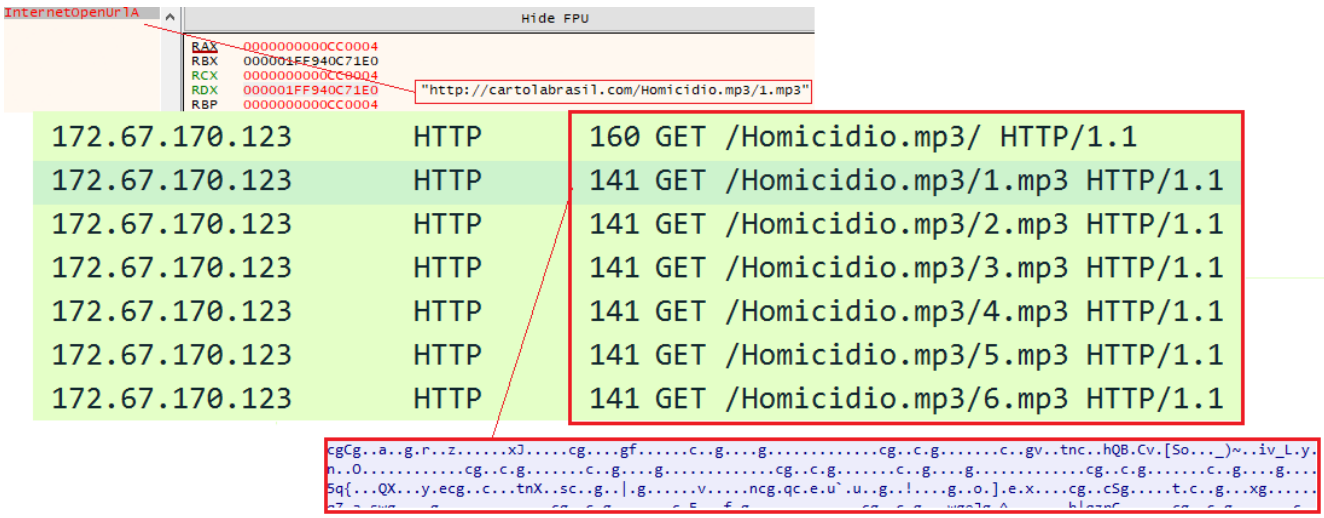

Figure 10 - Downloading multiple payloads from http[:]cartolabrasil.com.

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.

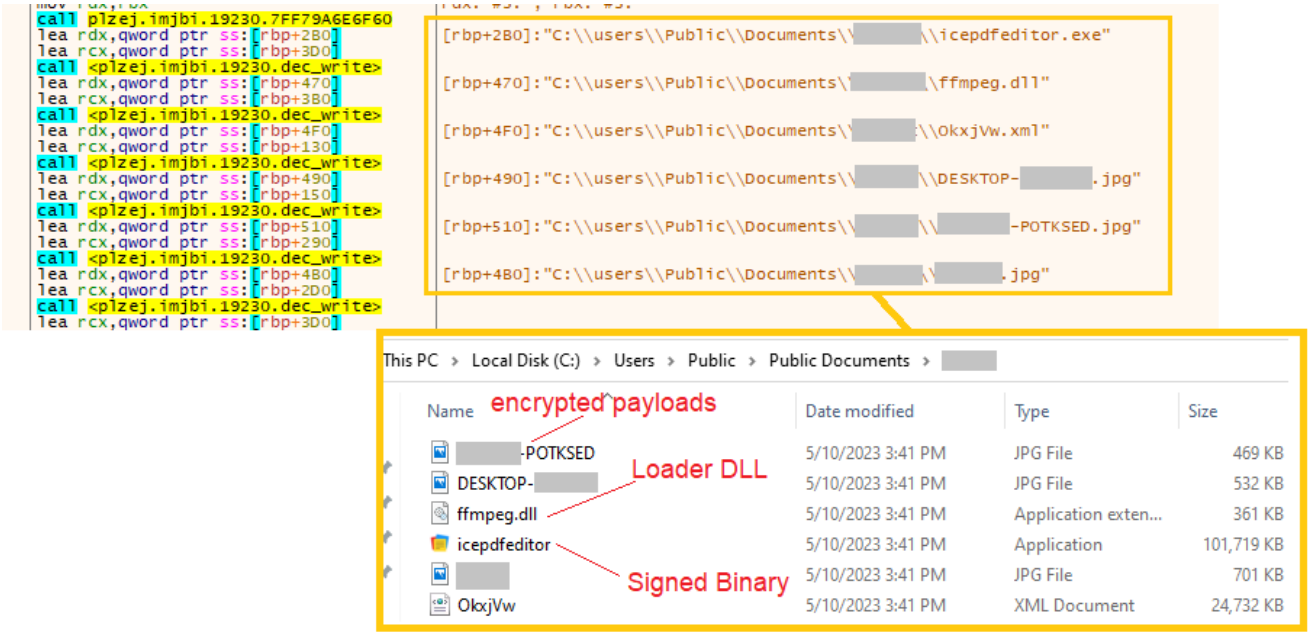

Figure 11 - Multiple payloads downloaded in the public documents folder.

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.

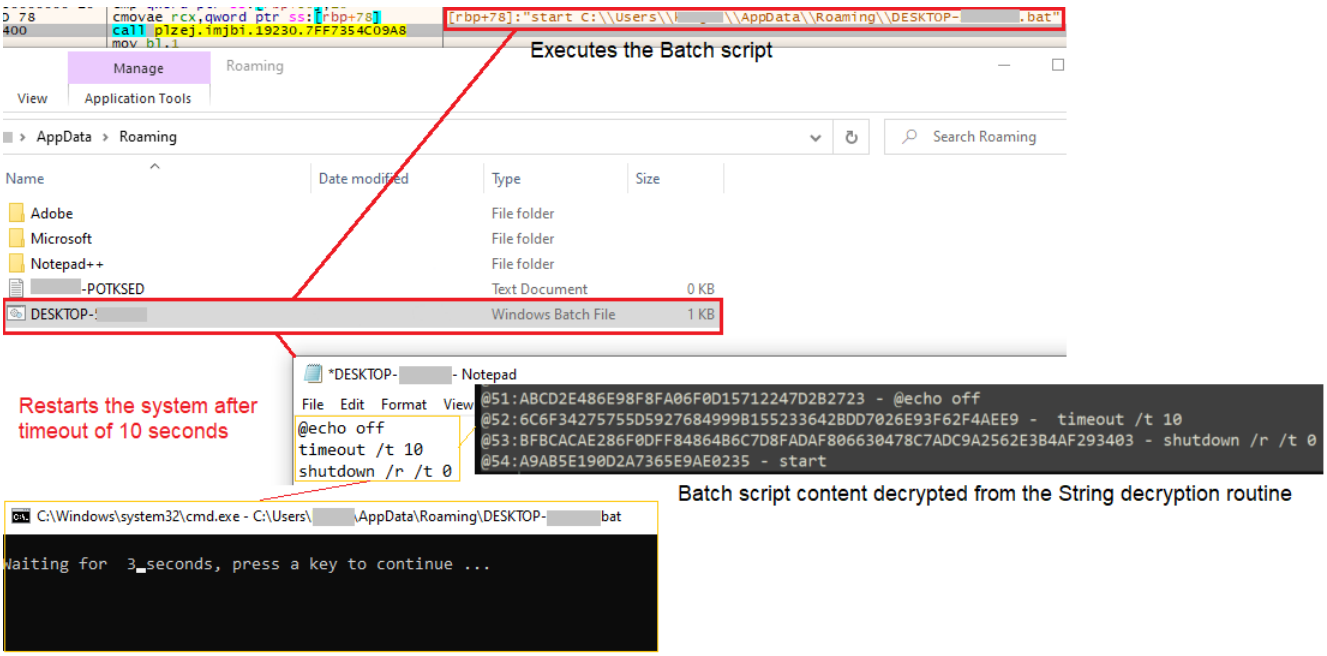

Figure 12 - Evades sandbox & executes the LNK file in the startup folder by restarting the system.

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**."

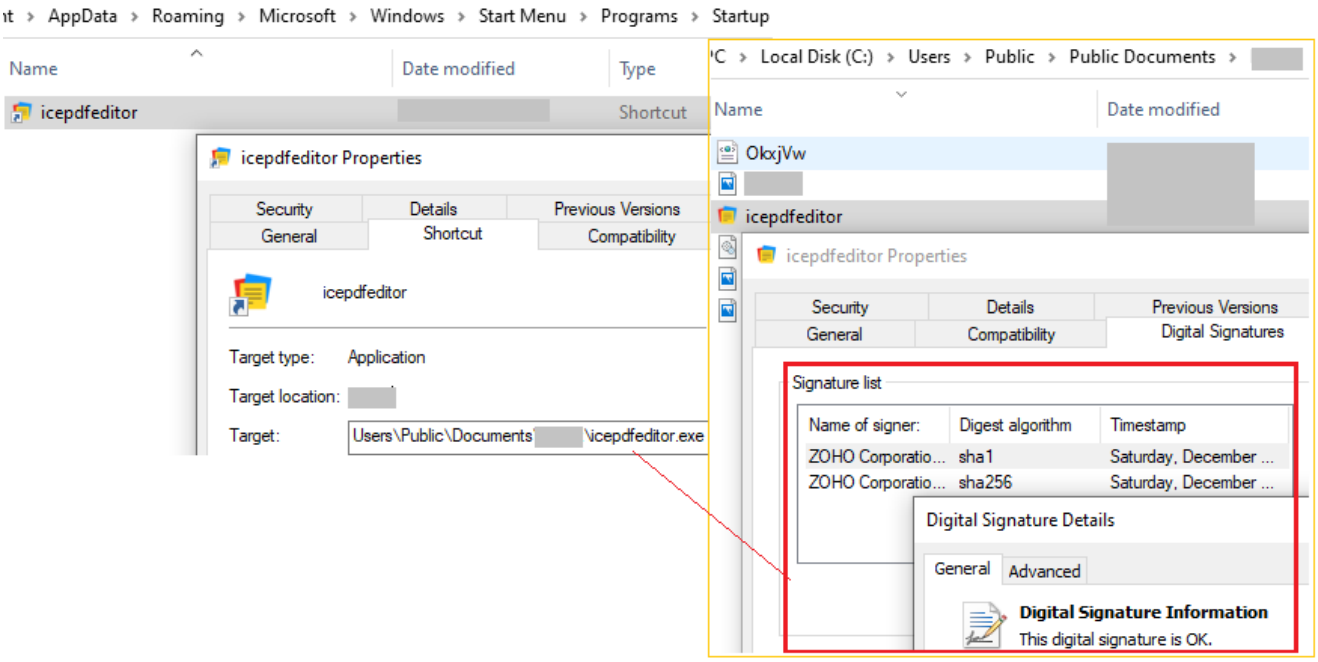

Figure 13 - Signed binary by ZOHO Corporation downloaded alongside malicious payloads.

Upon final execution, the signed binary, "**icepdfeditor.exe**," sideloads the malicious Krita Loader DLL, "**ffmpeg.dll**," from the current directory "**C:\Users\Public\Documents\<username>**" 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.

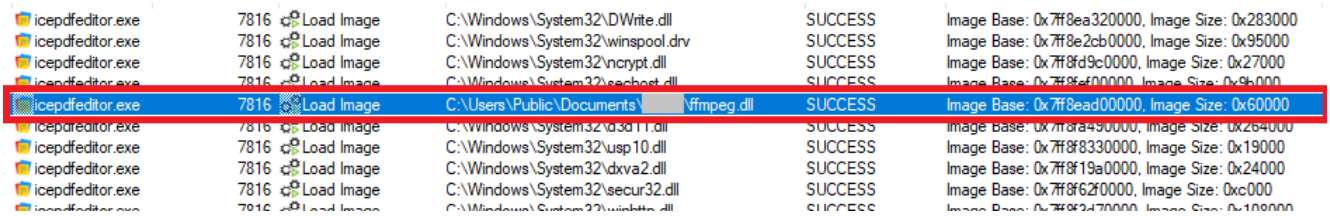

Figure 14 - Signed Binary "icepdfeditor.exe" sideloads the malicious Krita Loader DLL "ffmpeg.dll".

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 **<reverse_computer_name>.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.

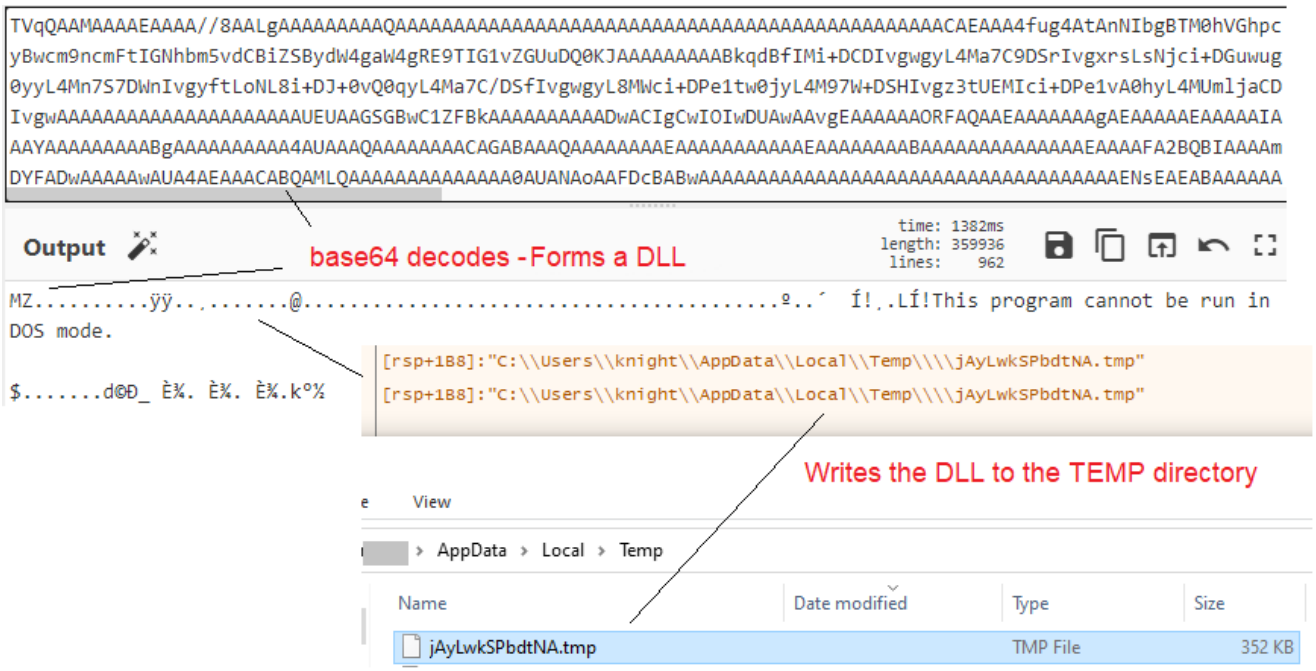

Figure 16 - Demonstrates the process of decoding the DLL through Base64 decoding.

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.

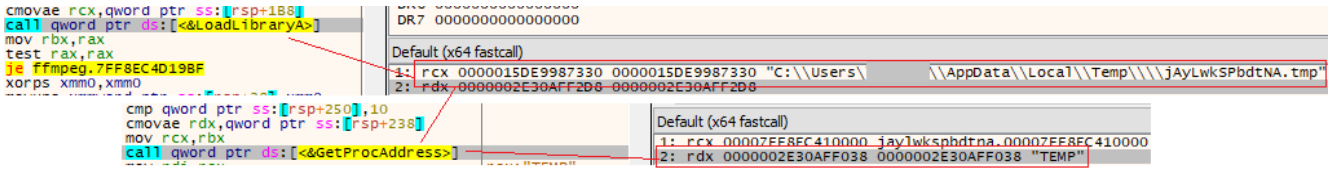

Figure 17 - Illustrates the loading of the InjectorDLL via the LoadLibraryA() function.

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 **<computer_name>.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.

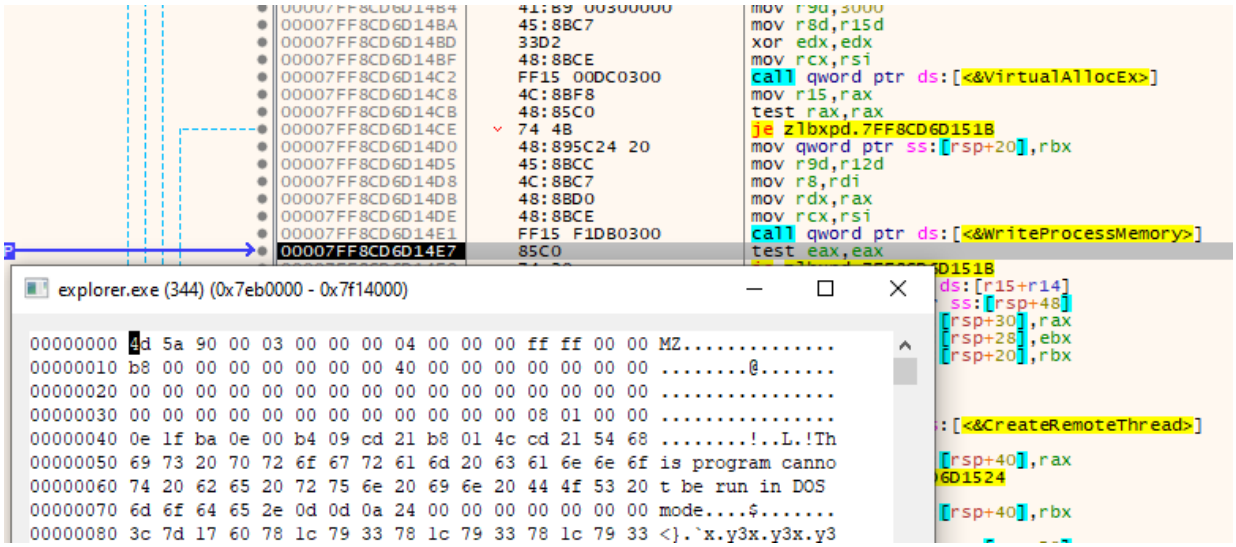

Figure 18 - Demonstrates the injection of the ElevateInjectorDLL module into the remote process "explorer.exe."

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.

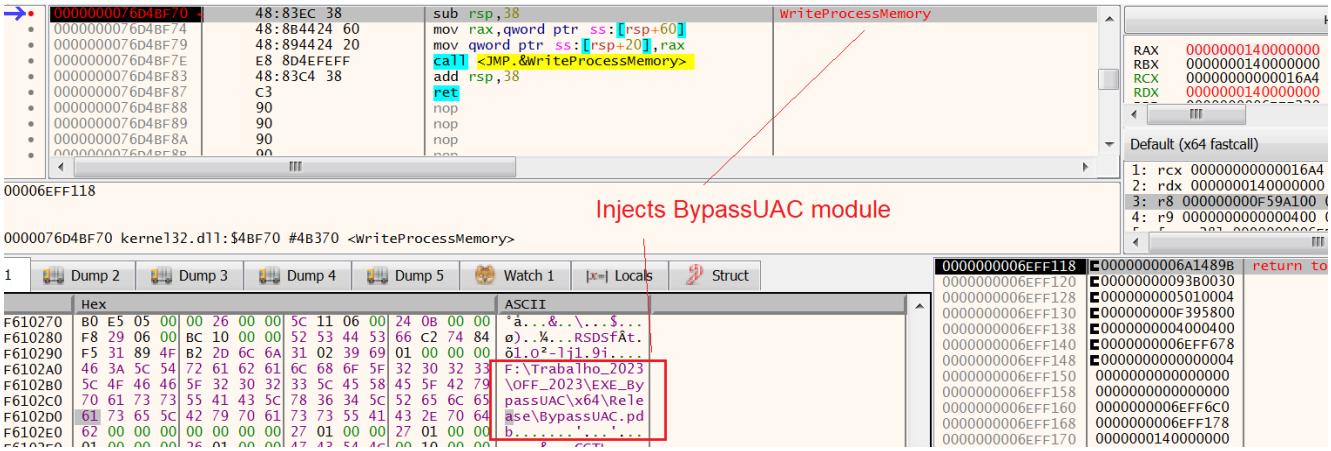

Figure 20 - Demonstrates the injection of the BypassUAC module into the explorer.exe process when the process is not elevated.

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.

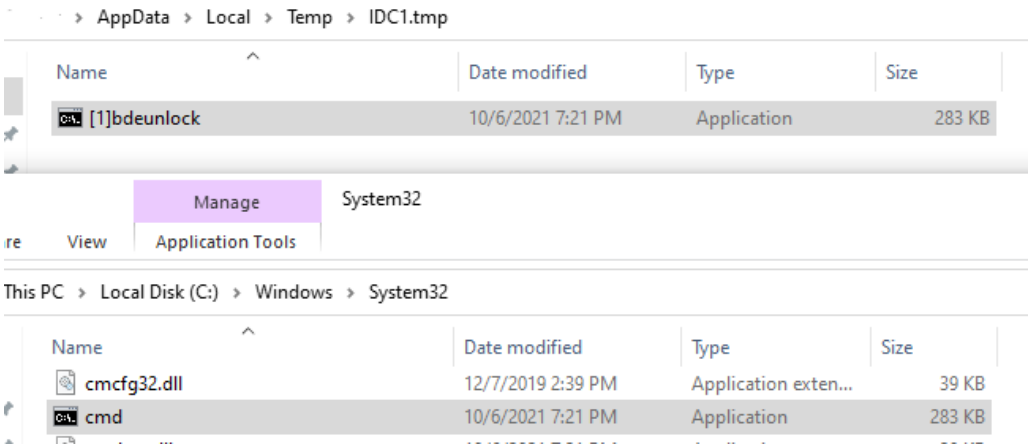

Figure 23 - Depicts the operation of copying the "cmd.exe" file into the Temp directory with administrator privileges using a COM Object.

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 <path_to_signed_binary>**," 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.

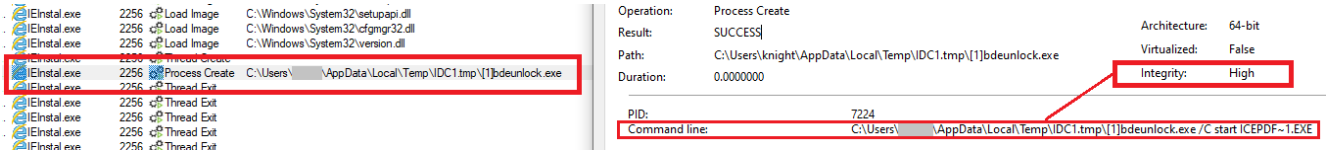

Figure 24 - Demonstrates the UAC bypass executed by the Internet Explorer Add-On Installer. This bypass enables the execution of the Krita Loader DLL with elevated privileges.

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.

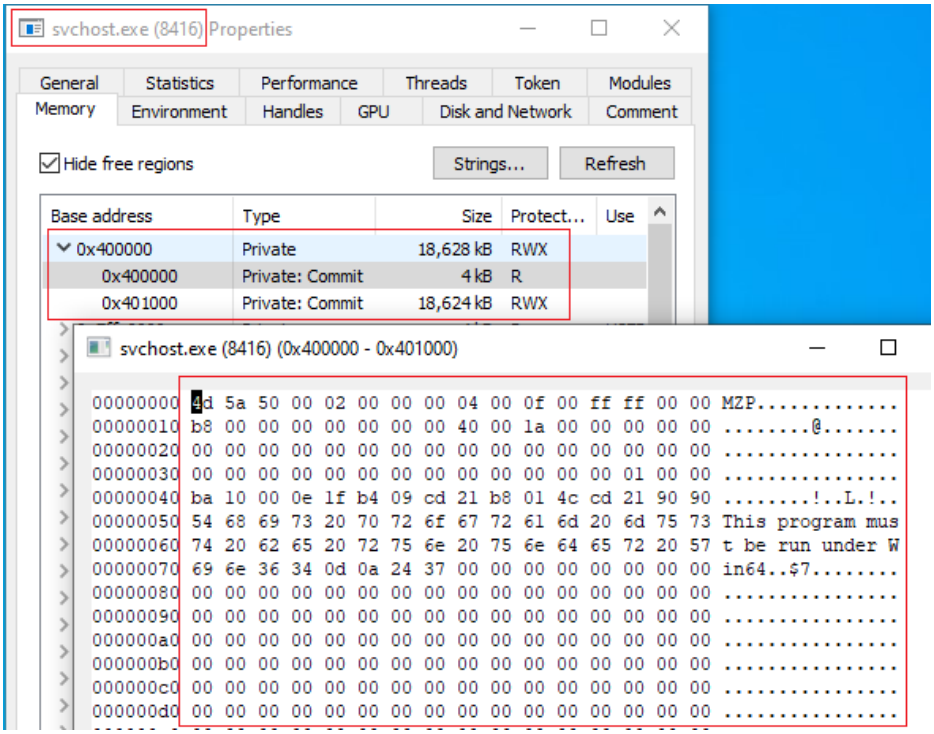

Figure 25 - Displays the injection of the TOITOIN Trojan into the svchost.exe process.

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 **<computer_name>.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.

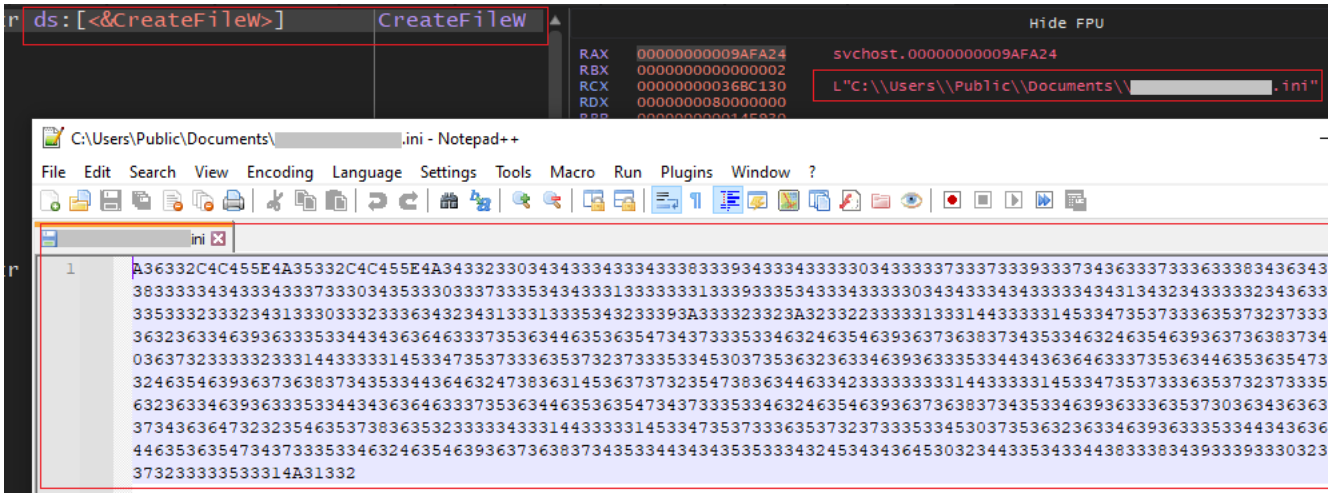

Figure 26 - Demonstrates the process of reading the INI configuration file.

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.

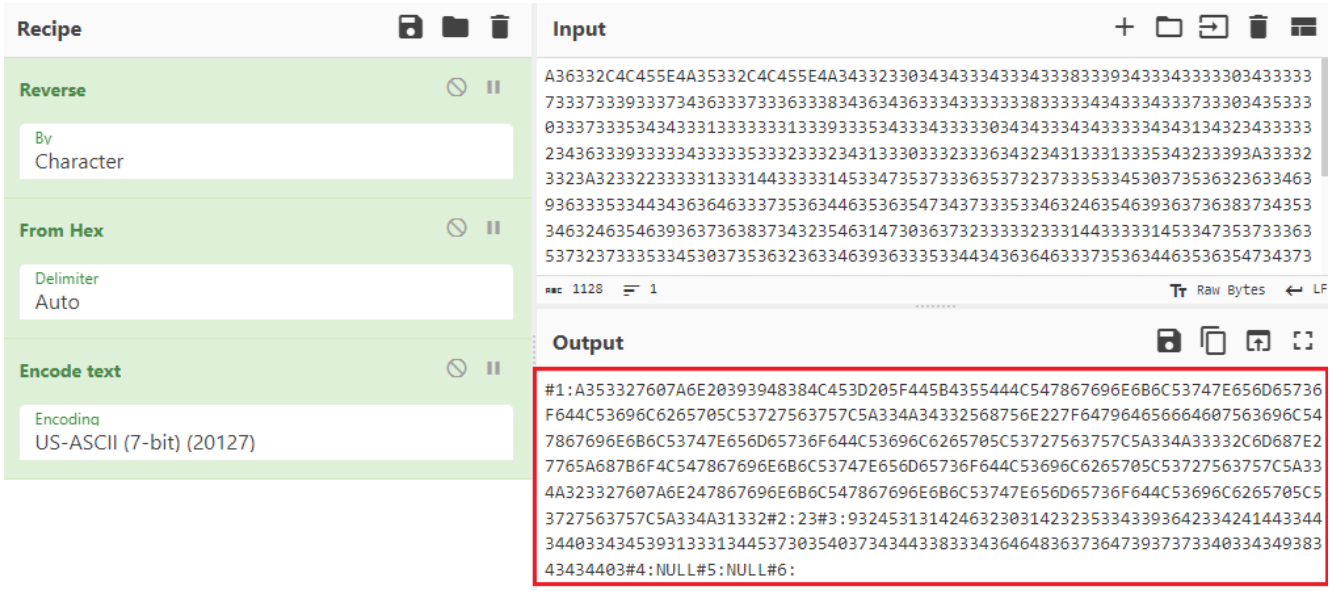

Figure 27 - Showcases the decoding process of the INI Configuration File.

Each of the #<1-6> 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**.

```
A353327607A6E20393948384C453D205F445B4355444C547867696E6B6C53747E656D657
36F644C53696C6265705C53727563757C5A334A34332568756E227F64796465666460756
3696C547867696E6B6C53747E656D65736F644C53696C6265705C53727563757C5A334A3
3332C6D687E27765A687B6F4C547867696E6B6C53747E656D65736F644C53696C6265705
C53727563757C5A334A323327607A6E247867696E6B6C547867696E6B6C53747E656D657
36F644C53696C6265705C53727563757C5A334A31332
```

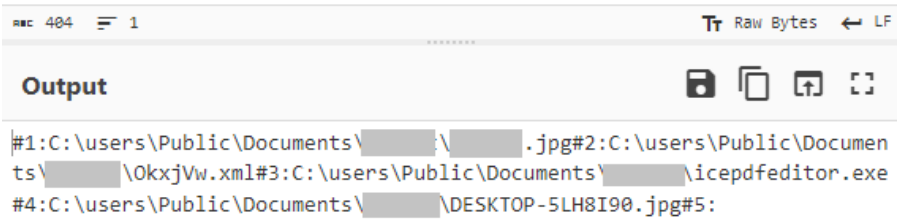

Figure 28 - Depicts the continued decoding process of the INI Configuration File.

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.

```
======INI Decoder======

[+] Encoded INI Data: 93245313142463230314232353433936423342414433443440334345393133313445373035403734344338333436464836373647393737334033434938343434403

[+] Decoded INI Data: 0D4489CC0C7797F768FF4383D470E075D13195CC0D4C4ABC2F93C522A026BA15B9
[+] Encoded INI CnC Data: 0D4489CC0C7797F768FF4383D470E075D13195CC0D4C4ABC2F93C522A026BA15B9
[+] Final Decoded CnC: http://afroblack.shop/CasaMoveis
```

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.

Command & Control Communication:

The TOITOIN Trojan communicates with the Command & Control (C&C) server located at **http[:]//afroblack[.]shop/CasaMoveis\ClienteD.php**, shown in Figure 30 below, to transmit encoded system information, browser details, and **Topaz OFD Protection Module** information.

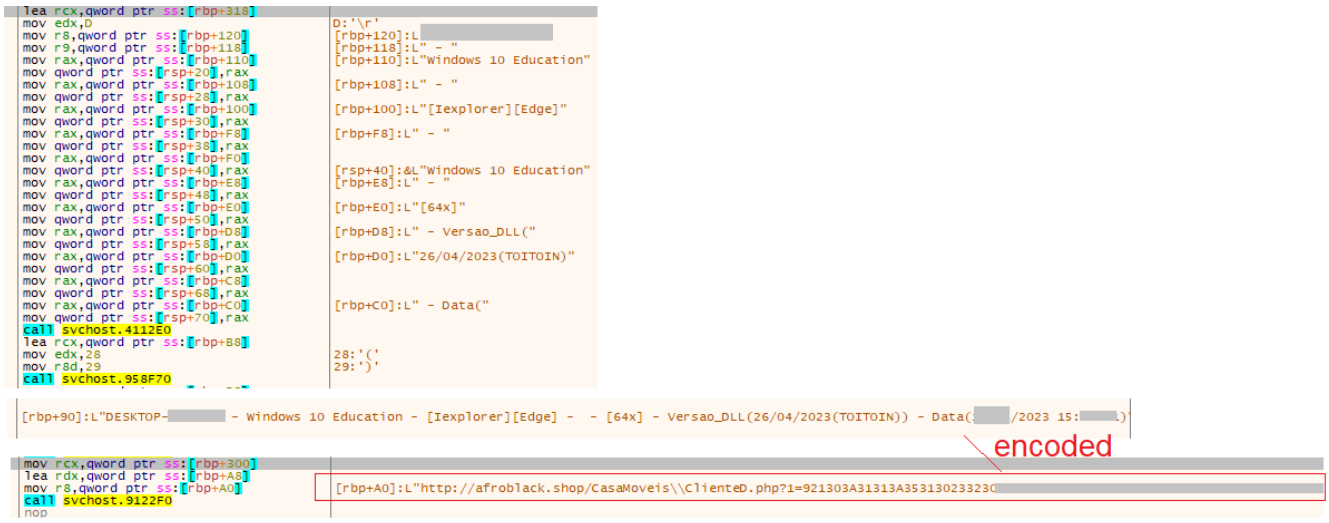

Figure 30 - Displays the information transmitted to the Command & Control (C&C) server located at http[:]afroblack.shop/CasaMoveis\ClienteD.php.

The exfiltrated information, once decoded, includes the following data:

- Computer Name: **DESKTOP-******
- Windows Version: **Windows 10 Education**
- Installed Browsers & Topaz OFD Protection Module: **[Iexplorer][Edge]**
- OS Bit Version: **[64x]**
- DLL Version: **Versao_DLL(26/04/2023(TOITOIN))**
- Data: **Date and time of execution**

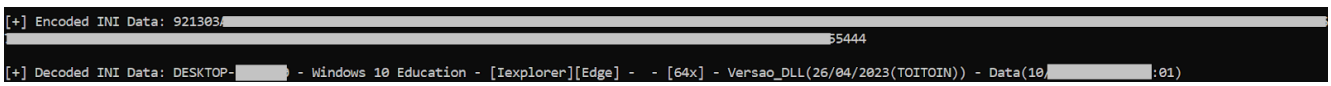

Figure 31 - Capture of decoded information sample transmitted to the Command & Control (C&C) server.

If the TOITOIN Trojan is unable to find the INI configuration file containing the URL to the C&C server, it resorts to sending the system information through a curl command. The encrypted data is then sent to the C&C server via a POST request using curl.

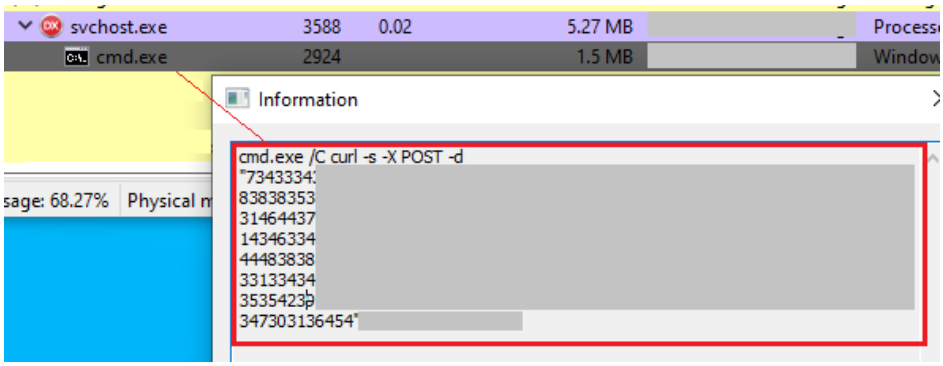

Figure 32 - Demonstrates the transmission of information to the Command & Control (C&C) server through a CURL POST request.

The screenshot below showcases the decrypted data that is sent via a curl POST request, resembling the previously observed request.

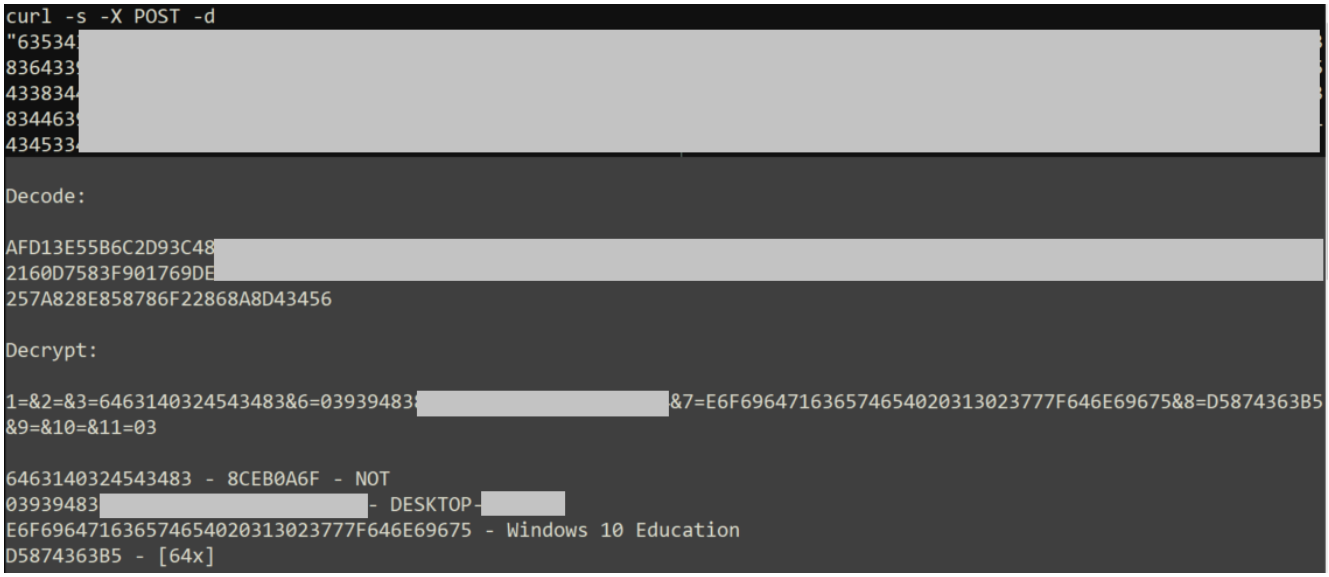

Figure 33 - Screenshot of the decrypted data that is transmitted through a CURL POST request.

Due to the unavailability of the Command and Control (C&C) servers during the analysis, the responses from the server could not be fetched.

Exploring the Open Directory:

While conducting the analysis, researchers came across a decrypted URL, "**191[.]252[.]203[.]222/Up/indexW.php**," which was found by the **InjectorDLL** module. Upon exploring the endpoints associated with this URL, it was discovered that an open directory contained various stager modules, including the **Loader DLL**, **Injection DLL**, **InjectorDLL Module**, **BypassUAC Module**, and the initial **Downloader Module**. These binaries had been hosted on the server since March 2023.

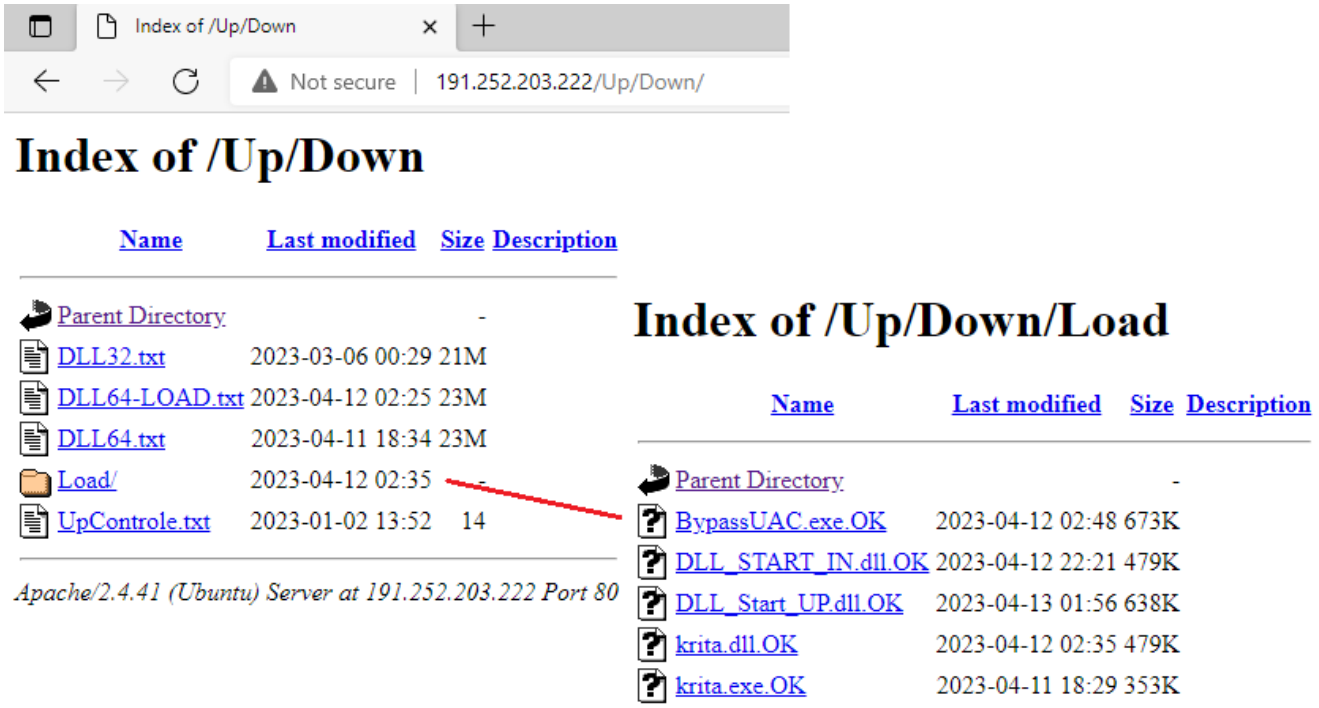

Figure 34 - Reveals the open directory hosting the payloads on the attacker-controlled server.

Zscaler Sandbox Coverage

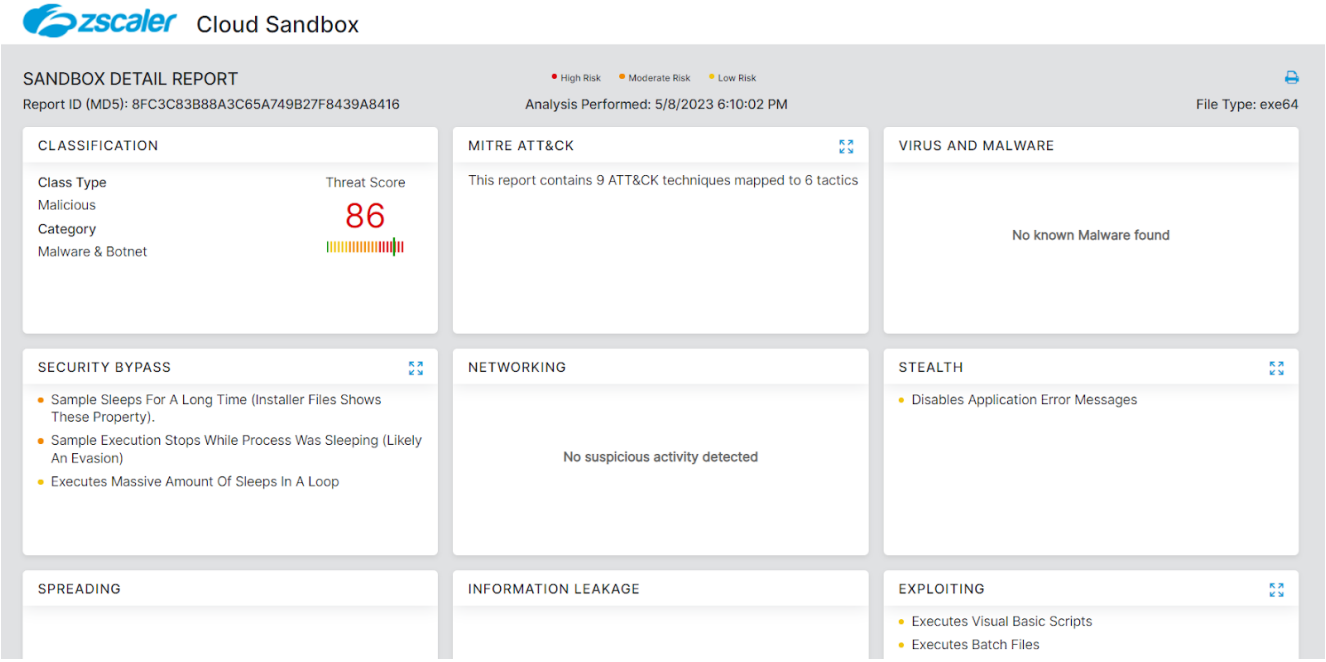

Figure 35 - Figure 35 presents the Zscaler Cloud Sandbox Report, which provides detailed analysis and insights into the behavior and characteristics of the analyzed malware.

The Zscaler Cloud Sandbox report includes information such as file hashes, observed behaviors, network communications, and potential indicators of compromise (IOCs). It serves as a valuable resource for understanding the malware's capabilities, allowing security analysts to take appropriate measures to protect their systems and networks. The Zscaler Cloud Sandbox Report plays a crucial role in identifying and mitigating potential threats and enhancing overall cybersecurity posture.

Win64.Downloader.Toitoin

Conclusion

In summary, the TOITOIN malware campaign targeting businesses in the Latin American region demonstrates the evolving tactics and sophistication of threat actors. Through deceptive phishing emails, intricate redirect mechanisms, and domain diversification, the threat actors successfully deliver their malicious payload. By leveraging resources such as the Amazon EC2 infrastructure and dynamically generated file names, they have shown their adaptability and persistence in compromising targeted systems.

The multi-staged infection chain observed in this campaign involves the use of custom-developed modules that employ various evasion techniques and encryption methods. The malware utilizes XOR decryption to decode configuration files and transmit system information to the command and control server. It also leverages COM Elevation Moniker for user account control bypass, ensuring the execution of malicious code with elevated privileges.

The analysis further revealed the presence of downloader modules, injector modules, and backdoors, each playing a specific role in the overall infection chain. The malware payload is injected into legitimate processes, such as explorer.exe and svchost.exe, to evade detection and maintain persistence on compromised systems.

Furthermore, the malware exhibits the ability to exfiltrate system information, including computer names, Windows versions, installed browsers, and other relevant data, to the command and control server. The communication with the CnC server occurs through encrypted channels, and in the absence of an INI configuration file, a curl POST request is utilized for data transmission.

The analysis also uncovered an open directory hosted on an attacker-controlled server, where various stager modules and payloads were found. These modules, including the Loader DLL, Injection DLL, and BypassUAC module, played critical roles in the infection chain.

Overall, this analysis highlights the importance of robust cybersecurity measures and continuous monitoring to detect and mitigate sophisticated threats like TOITOIN. Organizations should remain vigilant against evolving malware campaigns, implement strong security protocols, and regularly update their security systems to safeguard against such threats. By staying informed and proactive, businesses can effectively defend against emerging cyber threats and protect their critical assets.

One such measure that provides significant protection against malware threats like TOITOIN is the Zscaler Zero Trust Exchange. The Zscaler ThreatLabz team actively monitors and analyzes such campaigns, ensuring that customers are safeguarded against emerging threats. By leveraging the power of the Zscaler platform, organizations benefit from several key features that enhance their security posture.

Firstly, the Zscaler Zero Trust Exchange operates on a zero trust model, which means that all traffic, including email communications and web browsing, is inspected and analyzed in real-time, regardless of the user's location or device. This comprehensive inspection helps identify and block malicious emails, phishing attempts, and suspicious URLs associated with malware campaigns like TOITOIN.

Additionally, the Zscaler platform employs advanced threat intelligence and machine learning algorithms to detect and block known and unknown malware variants. The ThreatLabz team constantly updates the platform with the latest threat intelligence, ensuring that customers are protected against emerging threats as soon as they are detected.

Furthermore, Zscaler's cloud-native architecture enables rapid deployment of security updates and patches across the entire network, ensuring that customers are always equipped with the latest security defenses. This proactive approach minimizes the window of vulnerability for potential malware attacks.

Moreover, the Zscaler Zero Trust Exchange provides granular control over application access and user behavior, limiting the attack surface and reducing the risk of malware infiltration. By implementing strict access policies and enforcing least privilege principles, organizations can prevent unauthorized access and limit the spread of malware

within their network.

In conclusion, while threats like the TOITOIN malware campaign continue to evolve, the Zscaler Zero Trust Exchange provides a robust and comprehensive security framework to protect organizations against such threats. With continuous monitoring, advanced threat intelligence, proactive updates, and granular access control, Zscaler helps ensure that customers stay one step ahead of emerging malware campaigns. The Zscaler ThreatLabz team's unwavering commitment to customer safety reinforces the effectiveness of the Zscaler Zero Trust Exchange in safeguarding organizations against evolving cyber threats.

MITRE ATT&CK TTP Mapping

ID	Technique Name
T1566	**Phishing**
T1064	**Scripting**
T1037	**Startup Items**
T1055	**Process Injection**
T1018	**Remote System Discovery**
T1082	**System Information Discovery**
T1083	**File and Directory Discovery**
T1548.002	**Bypass User Account and Control**
T1574.002	**DLL Side-Loading**
T1055.012	**Process Hollowing**

Indicators of Compromise (IoCs)

1. Downloader Module:

- 8fc3c83b88a3c65a749b27f8439a8416
- 2fa7c647c626901321f5decde4273633
- ec2-3-89-143-150[.]compute-1[.]amazonaws[.]com/storage[.]php?e=Desktop-PC
- ec2-3-82-104-156[.]compute-1[.]amazonaws[.]com/storage.php?e=Desktop-PC
- http[:]//alemaoautopecas[.]com
- http[:]//contatosclientes[.]services
- atendimento-arquivos[.]com
- arquivosclientes[.]online
- fantasiacinematica[.]online
- http[:]//cartolabrasil[.]com

2. Krita Loader DLL:

b7bc67f2ef833212f25ef58887d5035a

3. InjectorDLL Module

690bfd65c2738e7c1c42ca8050634166

4. ElevateInjectorDLL Module

- e6c7d8d5683f338ca5c40aad462263a6
- 191[.]252[.]203[.]222/Up/indexW.php

5. BypassUAC Module

c35d55b8b0ddd01aa4796d1616c09a46

6. TOITOIN Trojan

- 7871f9a0b4b9c413a8c7085983ec9a72
- http[:]//bragancasbrasil[.]com
- http[:]//179[.]188[.]38[.]7
- http[:]//afroblack[.]shop/CasaMoveis\ClienteD.php

Appendix I: String Decryptor - Downloader Module

```python
def reverse_string(enc_str):
    do_rev = ""
    for i in enc_str:
        do_rev = i + do_rev
    return do_rev

def decrypt_string(rev_string):
            string = rev_string
            i = 0
            j = 2
            k = 0
            data = ""
            f_len = len(string)/4

            while k < f_len:
                if (i>=2):
                    i+=2
                    j+=2

                val1 = string[i:j]

                i+=2
                j+=2
                val2 = string[i:j]

                int_conv1 = int(val1, 16)
                int_conv2 = int(val2, 16)

                xor_dec_int = int_conv1 ^ int_conv2
                xor_dec_hex = hex(xor_dec_int)[2:]

                final_dec_str = bytes.fromhex(xor_dec_hex).decode('utf-8')
                data += final_dec_str


                k = k + 1

            return data

print("\n======String Decryptor=====")
enc_str = input("[+] Encrypted String: ")
rev_string = reverse_string(enc_str)
dec_string = decrypt_string(rev_string)
print("\n[+] Decrypted String: " + dec_string)
```

Appendix II: INI Configuration Decryptor

```
def decryptcnc(enc_cnc):
            data = ""
            i = 0
            j = 2
            k = 0
            f_len = len(enc_cnc)

            while k < f_len:

                    str1 = enc_cnc[i:j]
                    k = j + 2
                    str2 = enc_cnc[j:k]
                    int_conv1 = int(str2, 16)

                    val2 = "31"
                    int_conv2 = int(val2, 16)

                    xor_dec_int = int_conv1 ^ int_conv2
                    xor_dec_hex = hex(xor_dec_int)[2:]

                    int_cony1 = int(xor_dec_hex, 16)
                    int_cony2 = int(str1, 16)

                    offset = "FF"
                    fff = int(offset, 16)

                    if int_cony1 < int_cony2:

                            xor_dec_intny = (int_cony1 - int_cony2)
                            add_xor_offset = xor_dec_intny + fff
                            xor_dec_hexy = hex(add_xor_offset)[2:]

                    else:
                        xor_dec_intny = int_cony1 - int_cony2
                        xor_dec_hexy = hex(xor_dec_intny)[2:]

                    final_dec_stry = bytes.fromhex(xor_dec_hexy).decode('utf-8')
                    data += final_dec_stry
                    i = i + 2
                    j = j + 2

            return data

print("\n======INI Decryptor=====\n")
enc_str = input("[+] Encoded INI Data: ")
rev_string = reverse_string(enc_str)
dec_string = decrypt_string(rev_string)
print("\n[+] Decoded INI Data: " + dec_string)
enc_cnc = input("[+] Encrypted INI Data: ")
dec_cnc = decryptcnc(enc_cnc)
print("[+] Final Decrypted Data: " + dec_cnc)
```