Updated StrelaStealer Targeting European Countries

blog.sonicwall.com/en-us/2024/04/updated-strelastealer-targeting-european-countries/

Security News

April 2, 2024

Overview

SonicWall Capture Labs threat research team has observed an updated variant of StrelaStealer. StrelaStealer is an infostealer malware known for targeting Spanish-speaking users and focuses on stealing email account credentials from Outlook and Thunderbird. StrelaStealer was reported in the wild in early November 2022. StrelaStealer has been updated with an obfuscation technique and anti-analysis technique.

Technical Analysis

MD5: 1E37C3902284DD865C20220A9EF8B6A9

SHA256: F2D7CF39392D394D6CCD0F9372DB7D486D4CB2BB6C3BBFD0D8BFBB6117A5E211

This updated version of malware delivered via JavaScript comes in archive files as attachments in emails. The initial vector is JavaScript which will drop the 64-bit executable file in the %userprofile% folder and execute the malware process. We have observed that StrelaStealer is being delivered as a 64-bit exe as well as a DLL via JavaScript. We are explaining the analysis for the 64-bit executable in this blog. This 64-bit executable is a wrapper that will act as a loader for the actual payload.

In the main 64-bit executable file, the data section has an encryption key, and the size of the encryption key is 0x2714 bytes. The encoded payload is embedded in the data section at the end of the encryption key. The size of the payload is 0x1C600. A single-byte XOR encryption is performed to decrypt an encoded PE file from the data section.

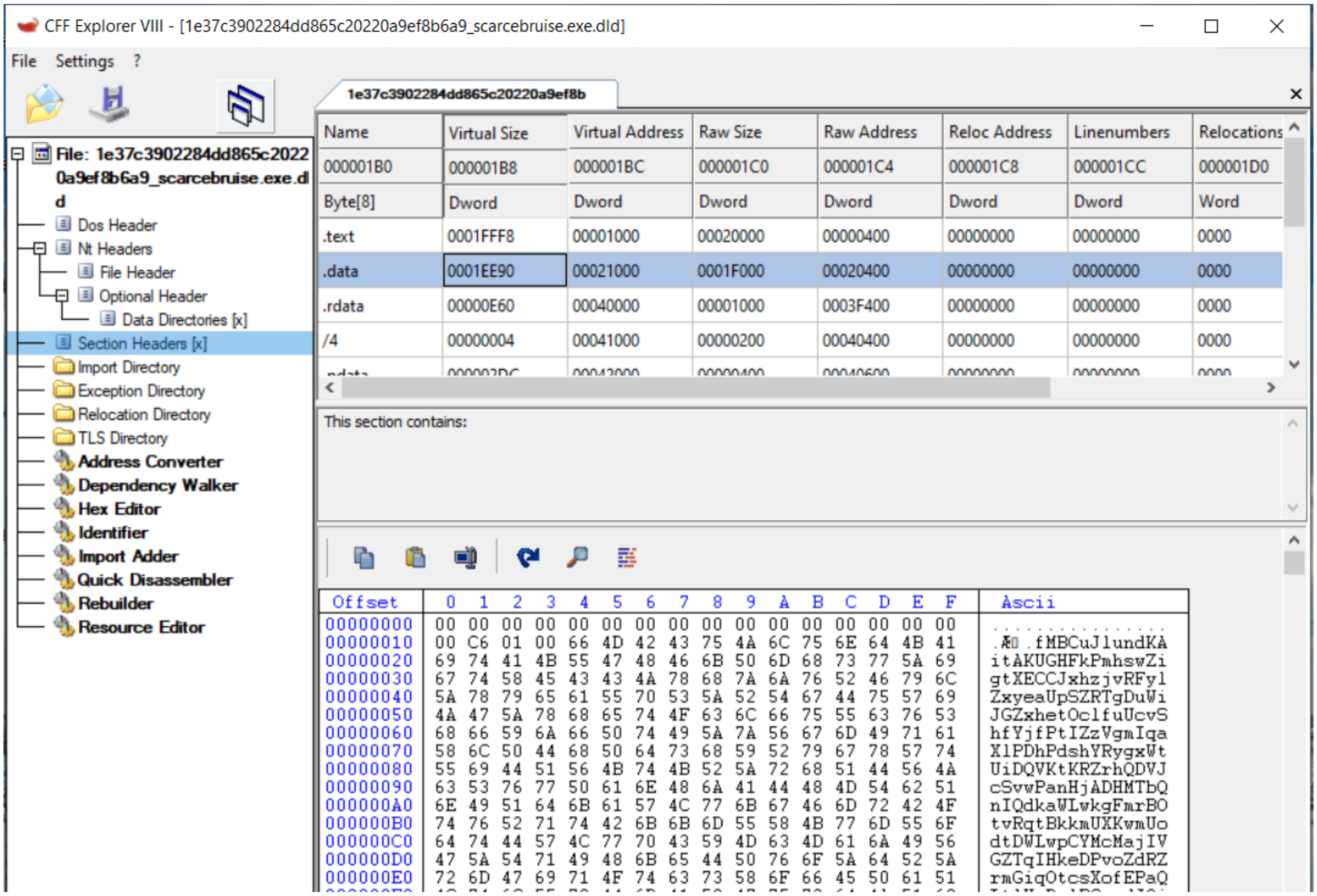


Figure 1: Encryption key started from 0x10th offset in the data section

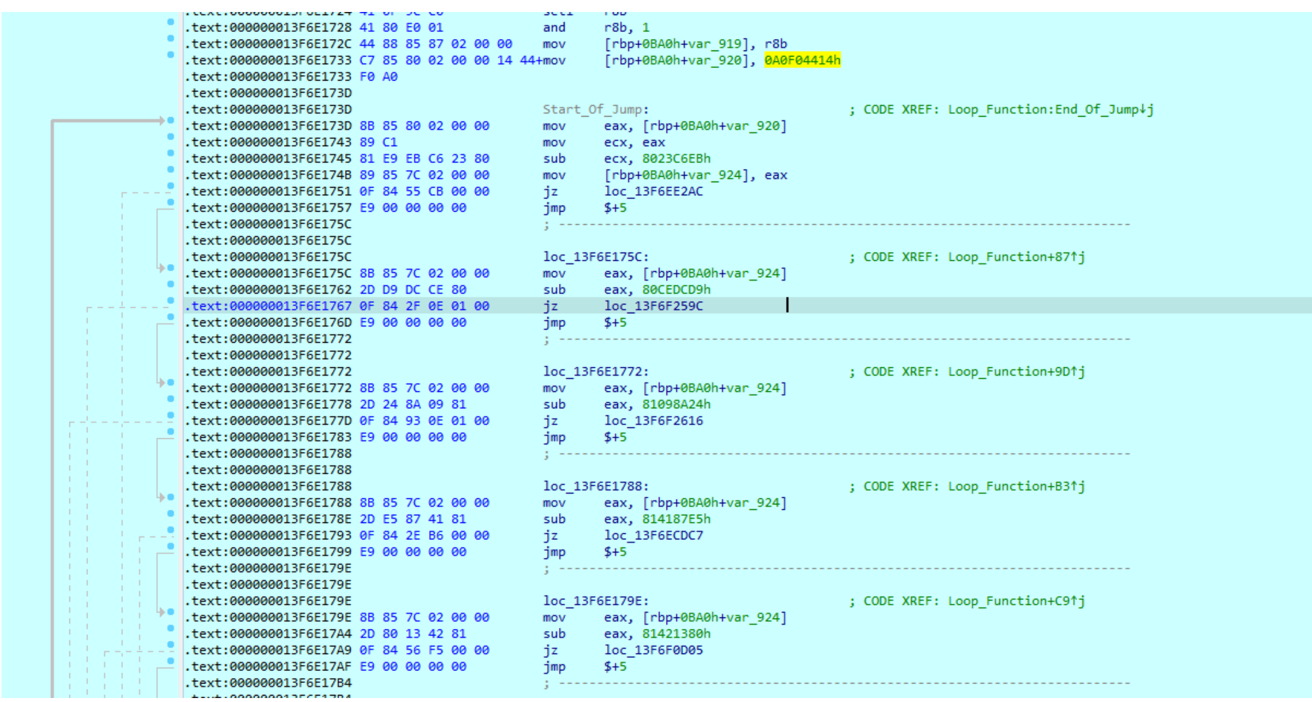


Figure 2: Obfuscated Jumps

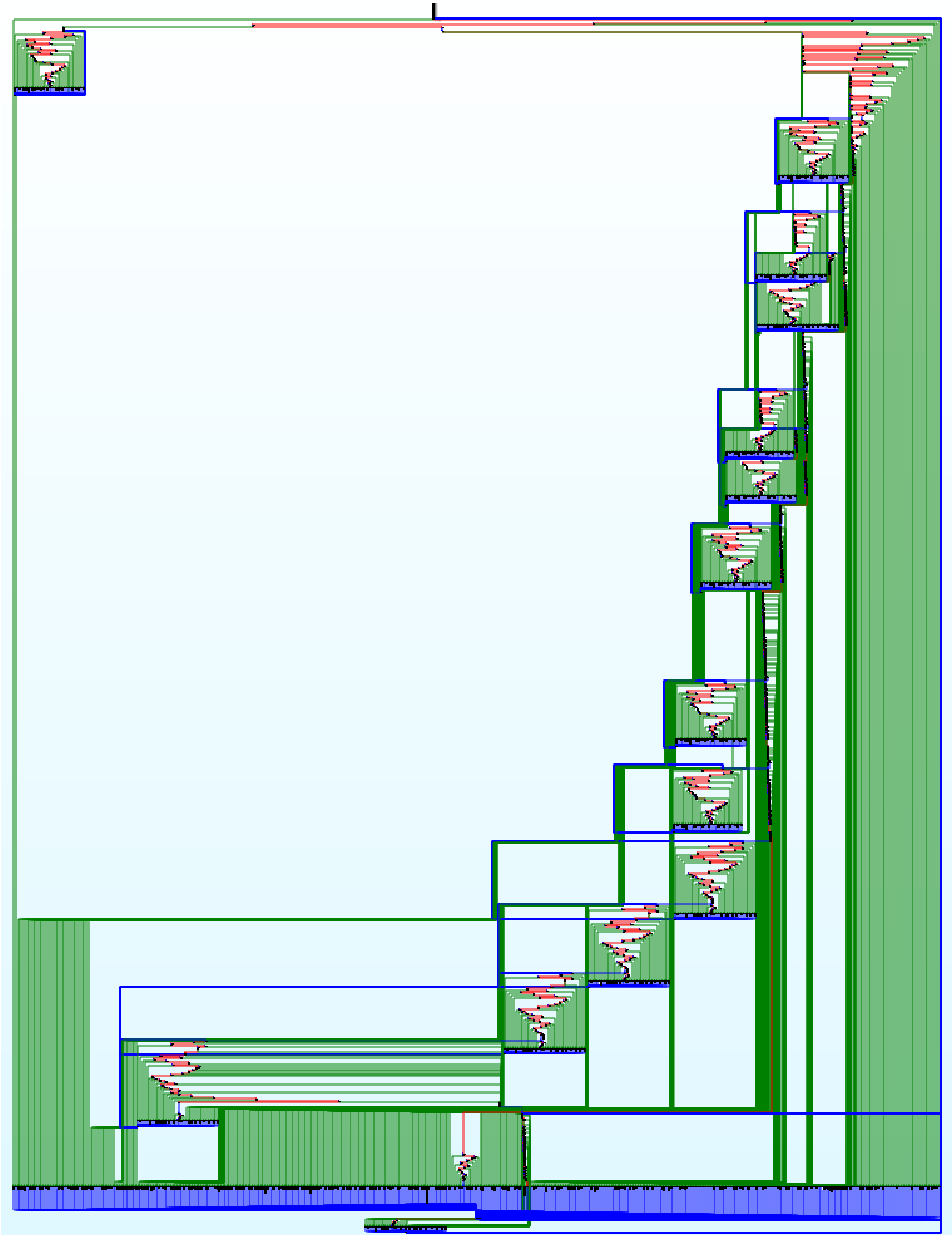


Figure 3: Graph view for obfuscated function

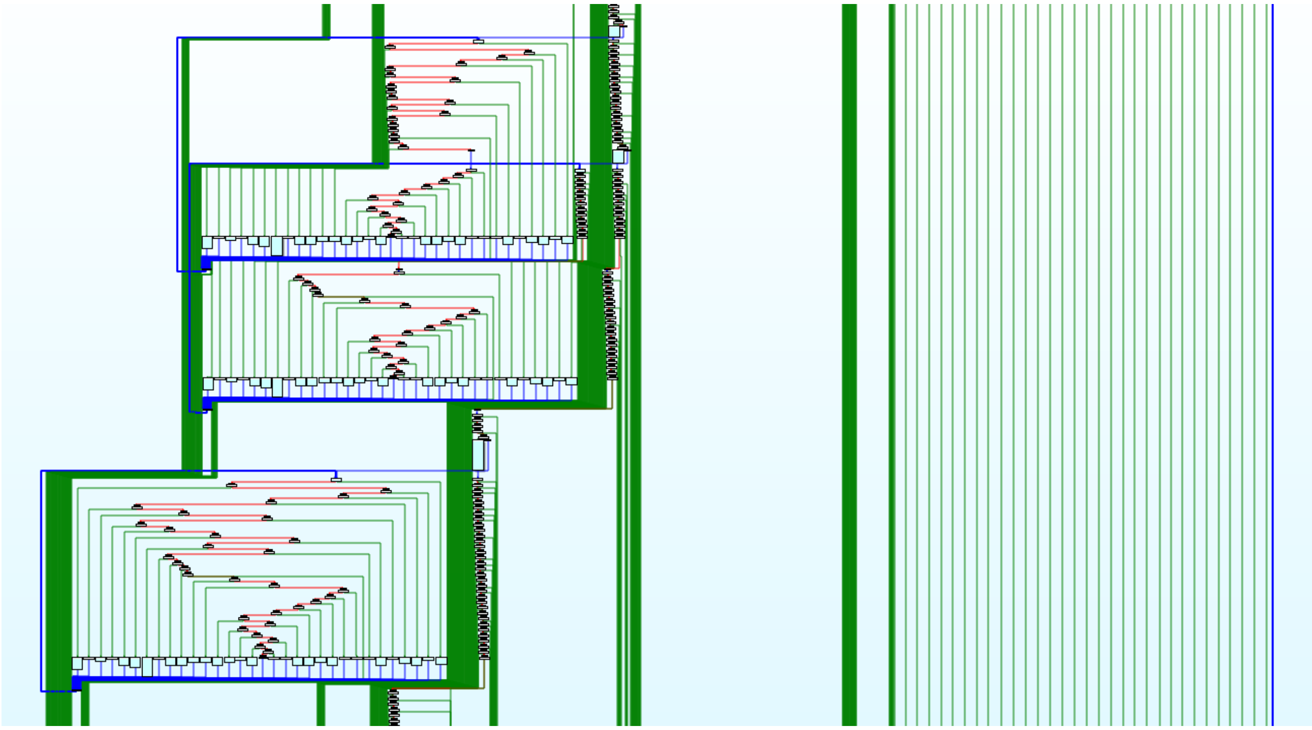
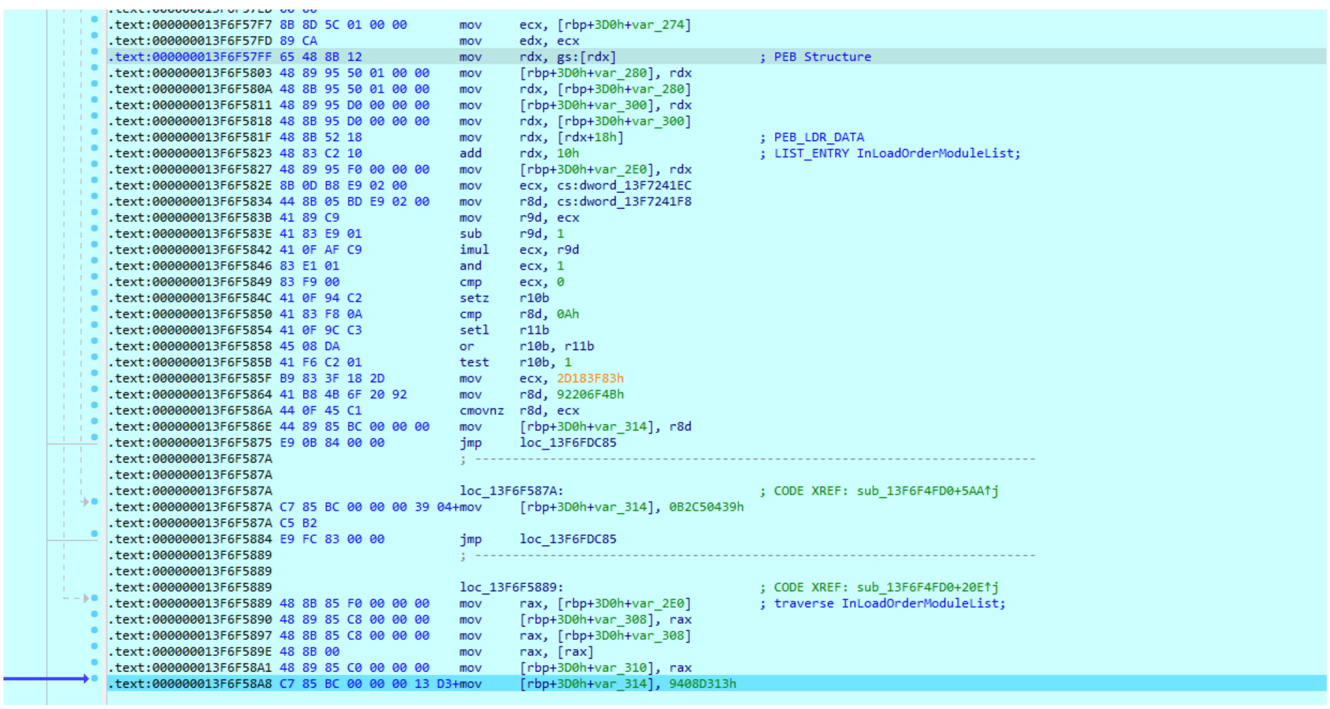


Figure 4: Another graph view of the obfuscated function



```
.text:000000013F6F57F7 8B 8D 5C 01 00 00       mov     ecx, [rbp+3D0h+var_274]
.text:000000013F6F57FD 89 CA                   mov     edx, ecx
.text:000000013F6F57FF 65 48 8B 12             mov     rdx, gs:[rdx]                  ; PEB Structure
.text:000000013F6F5803 48 89 95 50 01 00 00    mov     [rbp+3D0h+var_280], rdx
.text:000000013F6F580A 48 8B 95 50 01 00 00    mov     rdx, [rbp+3D0h+var_280]
.text:000000013F6F5811 48 89 95 D0 00 00 00    mov     [rbp+3D0h+var_300], rdx
.text:000000013F6F5818 48 8B 95 D0 00 00 00    mov     rdx, [rbp+3D0h+var_300]
.text:000000013F6F581F 48 8B 52 18             mov     rdx, [rdx+18h]                 ; PEB_LDR_DATA
.text:000000013F6F5823 48 83 C2 10             add     rdx, 10h                       ; LIST_ENTRY InLoadOrderModuleList;
.text:000000013F6F5827 48 89 95 F0 00 00 00    mov     [rbp+3D0h+var_2E0], rdx
.text:000000013F6F582E 8B 0D B8 E9 02 00       mov     ecx, cs:dword_13F7241EC
.text:000000013F6F5834 44 8B 05 BD E9 02 00    mov     r8d, cs:dword_13F7241F8
.text:000000013F6F583B 41 89 C9                mov     r9d, ecx
.text:000000013F6F583E 41 83 E9 01             sub     r9d, 1
.text:000000013F6F5842 41 0F AF C9             imul    ecx, r9d
.text:000000013F6F5846 83 E1 01                and     ecx, 1
.text:000000013F6F5849 83 F9 00                cmp     ecx, 0
.text:000000013F6F584C 41 0F 94 C2             setz    r10b
.text:000000013F6F5850 41 83 F8 0A             cmp     r8d, 0Ah
.text:000000013F6F5854 41 0F 9C C3             setl    r11b
.text:000000013F6F5858 45 08 DA                or      r10b, r11b
.text:000000013F6F585B 41 F6 C2 01             test    r10b, 1
.text:000000013F6F585F B9 83 3F 18 2D          mov     ecx, 2D183F83h
.text:000000013F6F5864 41 B8 4B 6F 20 92       mov     r8d, 92206F4Bh
.text:000000013F6F586A 44 0F 45 C1             cmovnz  r8d, ecx
.text:000000013F6F586E 44 89 85 BC 00 00 00    mov     [rbp+3D0h+var_314], r8d
.text:000000013F6F5875 E9 0B 84 00 00          jmp     loc_13F6FDC85
.text:000000013F6F587A                         ; ---------------------------------------------------------------
.text:000000013F6F587A
.text:000000013F6F587A                         loc_13F6F587A:                         ; CODE XREF: sub_13F6F4FD0+5AA↑j
.text:000000013F6F587A C7 85 BC 00 00 00 39 04+mov     [rbp+3D0h+var_314], 0B2C50439h
.text:000000013F6F587A C5 B2
.text:000000013F6F5884 E9 FC 83 00 00          jmp     loc_13F6FDC85
.text:000000013F6F5889                         ; ---------------------------------------------------------------
.text:000000013F6F5889
.text:000000013F6F5889                         loc_13F6F5889:                         ; CODE XREF: sub_13F6F4FD0+20E↑j
.text:000000013F6F5889 48 8B 85 F0 00 00 00    mov     rax, [rbp+3D0h+var_2E0]        ; traverse InLoadOrderModuleList;
.text:000000013F6F5890 48 89 85 C8 00 00 00    mov     [rbp+3D0h+var_308], rax
.text:000000013F6F5897 48 8B 85 C8 00 00 00    mov     rax, [rbp+3D0h+var_308]
.text:000000013F6F589E 48 8B 00                mov     rax, [rax]
.text:000000013F6F58A1 48 89 85 C0 00 00 00    mov     [rbp+3D0h+var_310], rax
.text:000000013F6F58A8 C7 85 BC 00 00 00 13 D3+mov     [rbp+3D0h+var_314], 9408D313h
```

Figure 5: PEB parsing code fragments inside the jump code block

This obfuscation is quite effective. Anti-analysis techniques delay the execution, and the researcher has to search the code fragments inside the jump blocks, which is a tedious task.

Along with jump blocks and multiple loops, there are multiple dummy functions that are not doing anything but wasting time while analyzing the sample.

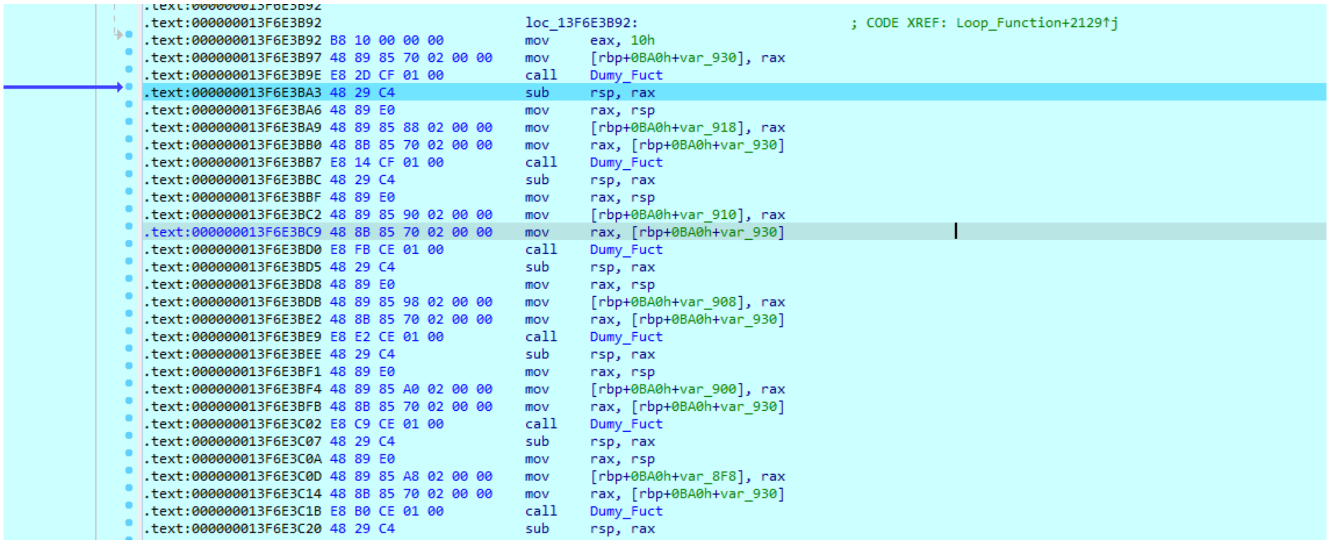


Figure 6: Dummy functions inside nested Jumps

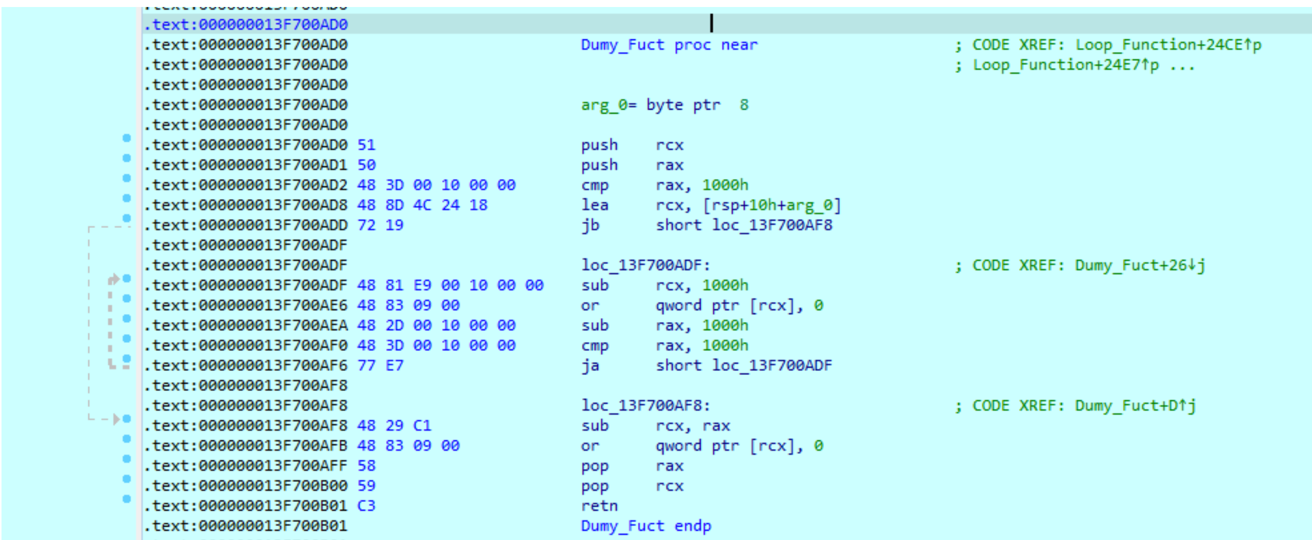


Figure 7: Dummy functions

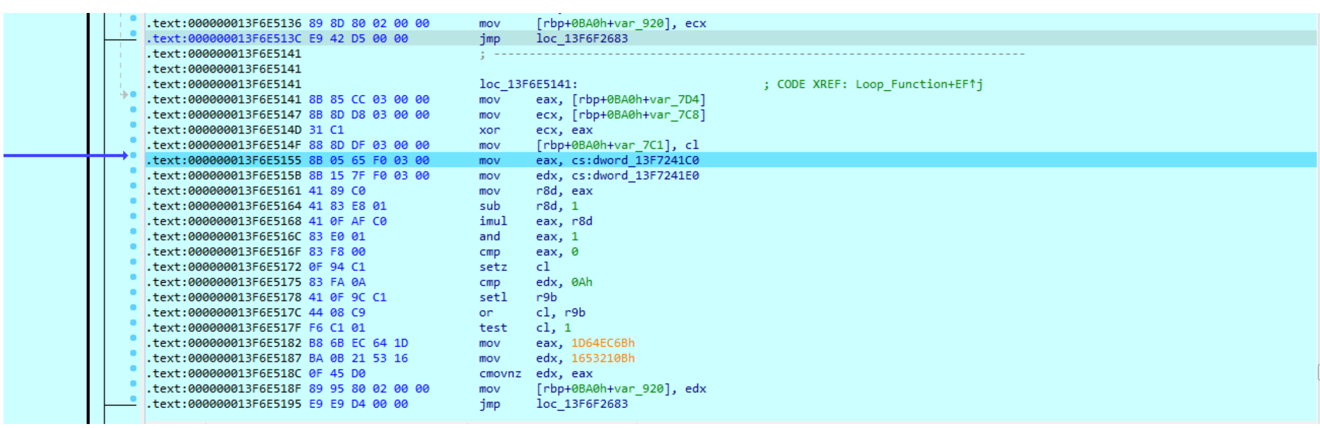


Figure 8: XOR decryption to decrypt the encoded payload

Once it decrypts the payload, it reads the encoded API string array at the end of the encoded payload embedded in the data section. Within the payload, the first DWORD is the size of the array and next is the API function array. This array is of size 0x52 bytes and the encryption key used earlier to decrypt the payload will also be used to decrypt the API array. The only difference between the decryption of the payload and the array is malware uses an encryption key of size 0x52 bytes from the 4th offset of encryption key.

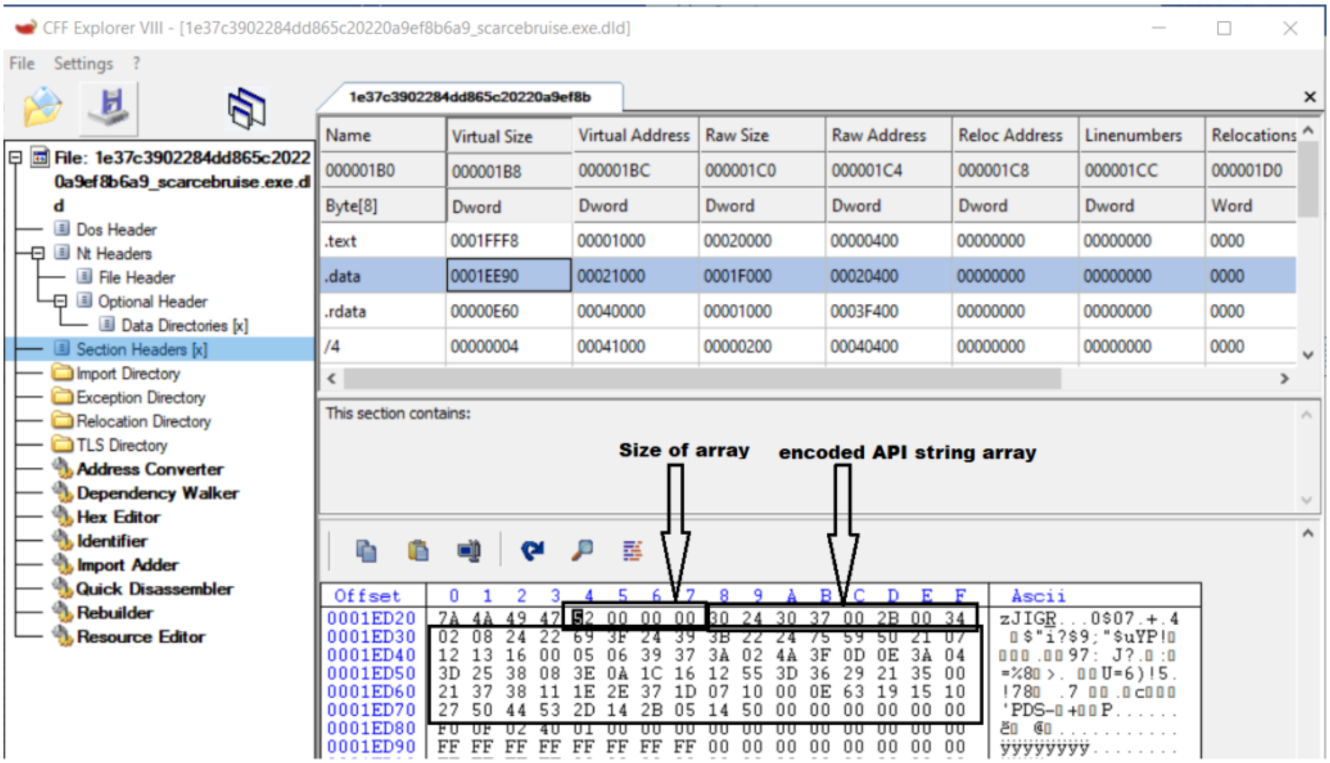


Figure 9: Encoded API array

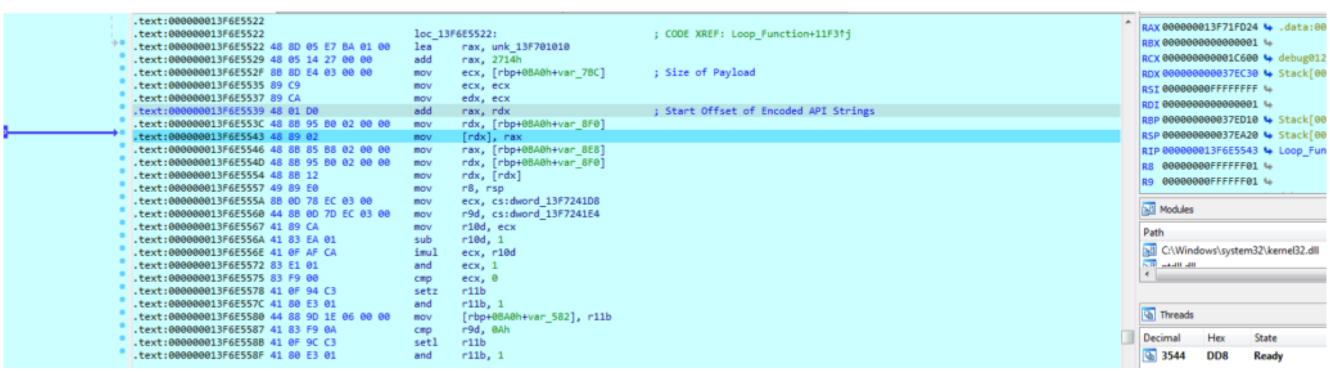


Figure 10: Malware calculates the start offset of the encoded API string and starts decrypting it

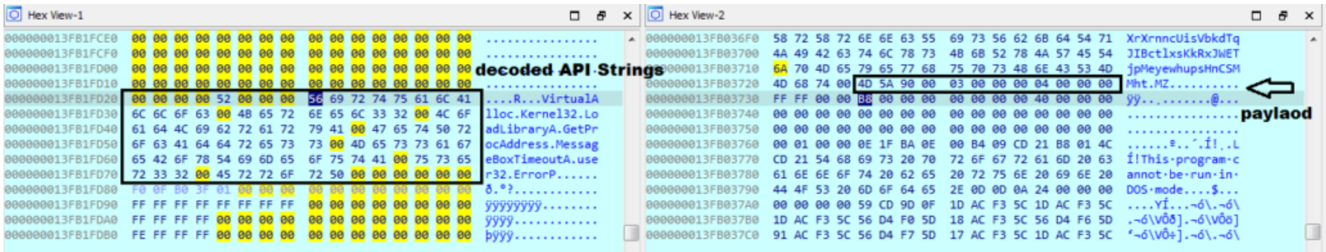


Figure 11: API array after an XOR decryption

It accesses the PEB structure and parses it to get the list of loaded modules in process memory.

The following is an example of the instructions set to parse the PEB.

```
mov rax, gs:60h       ; Access PEB structure

mov r15, [rax+18h]   ; Access PEB_LDR_DATA structure

add r15, 10h           ;  Pointer to LIST_ENTRY InLoadOrderModuleList
```

Figure 11B: Instructions

Here InLoadOrderModuleList is a doubly-linked list that contains the loaded modules for the process.

The malware parses this “InLoadOrderModuleList” to get the Imagebase address of kernel32.dll with the goal of resolving the VirutalAlloc API Then the malware will parse the PE structure of kernel32.dll to get the name of each exported function and matches them with the API string that got decrypted earlier in 0x52 byte array. If the API name matches the exported function name, then the malware will read the associated function RVA from the export directory and add it to the Imagebase of kernel32.dll,. Using this method, the malware resolves each API dynamically. It will resolve 4 APIs – here VirtualAlloc, LoadLibraryA , GetProcAddress , and MessageBoxTimeoutA. Once its finished resolving the APIs, the malware will show the error massage box and then continue execution.

Now, the malware calls the “VirtualAlloc” API to allocate memory in the process and start its task as loader to load the actual payload.

- The malware parses the PE file structure of the payload from the data section where previously it decrypted the PE file and read each section header one by one.
- To map the process as per section alignment, it reads the virtual address of each section and adds it to the image base of the injected PE and copies each section of data to this offset in memory.
- The malware will not copy the PE header to the injected PE, this has been done intentionally to evade detection from AV products.

- It reads the relocation section and does the fixup as it gets loaded at the different base address in the memory.
- It reads the import address table of the payload file from the data section region and resolves the API address dynamically using the “LoadLibraryA” and “GetProcAddress” APIs and copies these all function pointers to the IAT of the injected payload.
- When the injected PE file is ready for execution, it will read the RVA of the address from the entry point from the PE file in the data section and add the base address of the injected payload and redirect execution to the injected code.

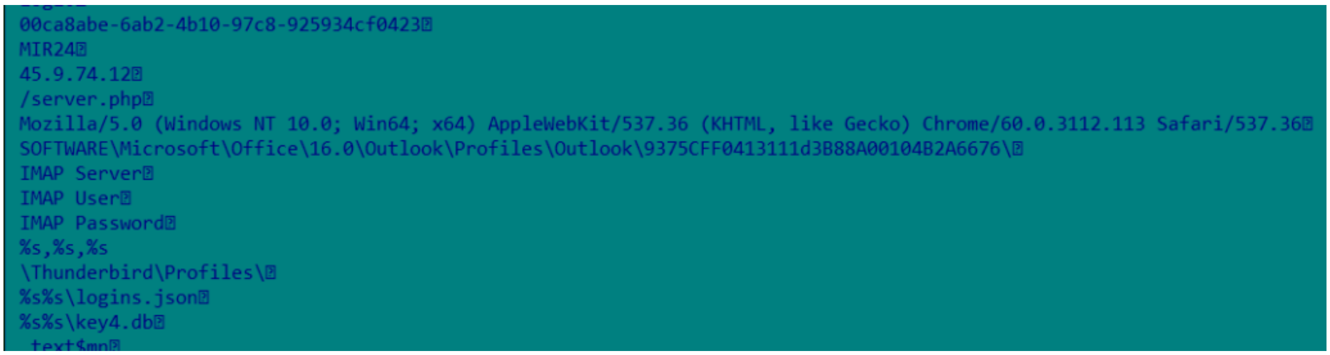


Figure 12: Configuration setting for the payload

The injected payload is 64-bit executable file, it will call the “GetKeyboardLayout” API and check the lower words of the return value with the hardcoded values in binary. It tries to check if the keyboard layout is from the following countries. If it is, then the malware will continue its execution, otherwise it terminates itself.

Language	Location (or type)	Language ID
German	Germany	0x0407
Spanish	Spain	0x040A
Spanish	Spain	0x0C0A
Catalan	Spain	0x0403
Basque	Spain	0x042D
Italian	Italy	0x0410
Polish	Poland	0x0415

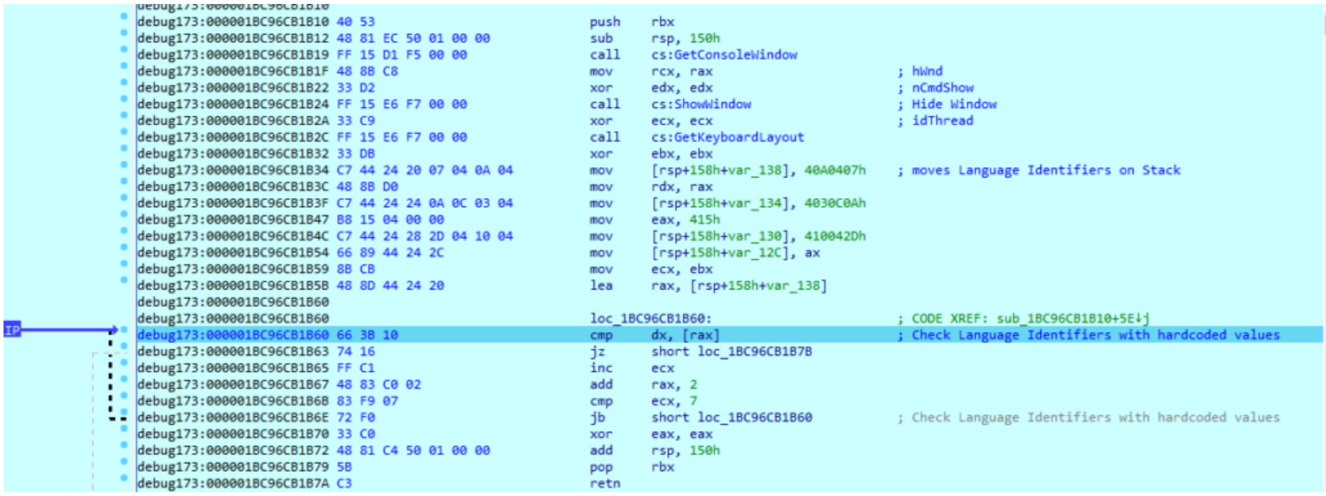


Figure 13: Call to the “GetKeyboardLayout” API and check language identifiers

Now, the payload retrieves the computer name by calling the “GetComputerNameA” API and encrypts the first 4 bytes of the computer name string using single byte XOR encryption. The encryption key is “MIR24”, which is hardcoded in binary. It will create a Mutex with the name of this partially encrypted computer name string. If a Mutex already exists, it will terminate it.

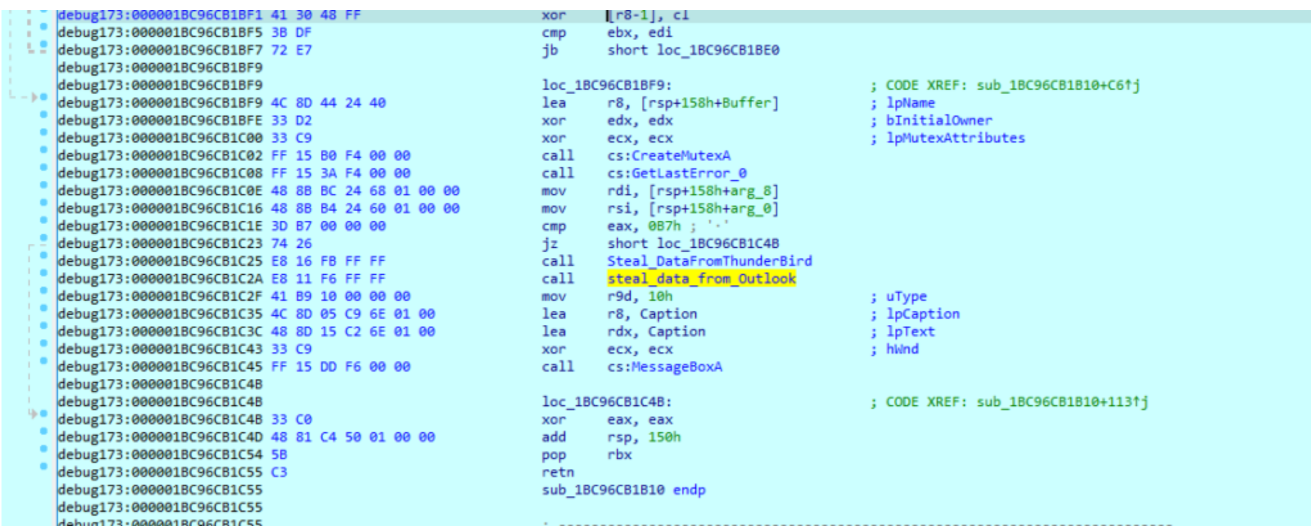


Figure 14: Creating a Mutex and executing its core functionality to steal data from the infected machine

As we can see in Figure 14, it will execute the function which will steal confidential data from the infected machine.

Here, we have found two functions in the malware. The first is used to steal data from Mozilla Thunderbird, which is a free and open-source email client software. The other function is intended to steal data from Outlook.

It searches for the folder path “C:\Users\<username>\AppData\Roaming\Thunderbird\Profiles\”

All of your data such as messages, passwords and user preferences as well as changes made while you use Thunderbird are stored in a special folder called *profile*.

- If it finds this folder path on the system, it will call the FindFirstFileAand FindNextFileA APIs to search for two files in the subdirectory. The first is “logins.json” (account and password) and the second is “key4.db” (password database).
- It reads the data from both of these files and appends both files’ data one after another, starting network communication.
- It establishes a connection to its server and prepares an HTTP post request with the user-agent “Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36” and then exfiltrates this data to its server.

http[:]//45[.]9[.]74.12/server.php .

- The server IP is hardcoded in binary which is “45.9.74[.]12”
- Before sending data to the server, it will encrypt it with the single byte XOR encryption. The encryption key is hardcoded in binary which is “00ca8abe-6ab2-4b10-97c8-925934cf0423”

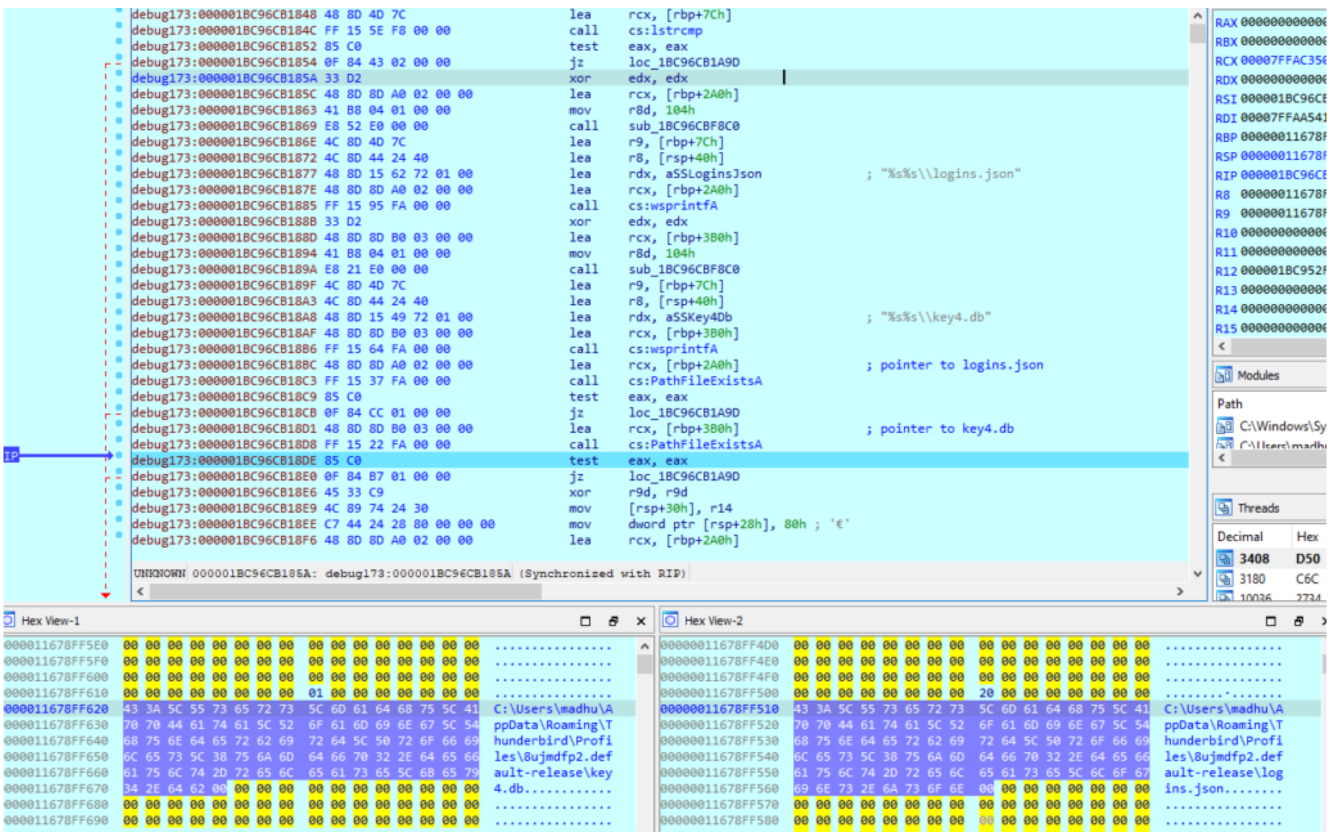


Figure 15: Searches for the “logins.json” and “key4.db” files from the profile folder

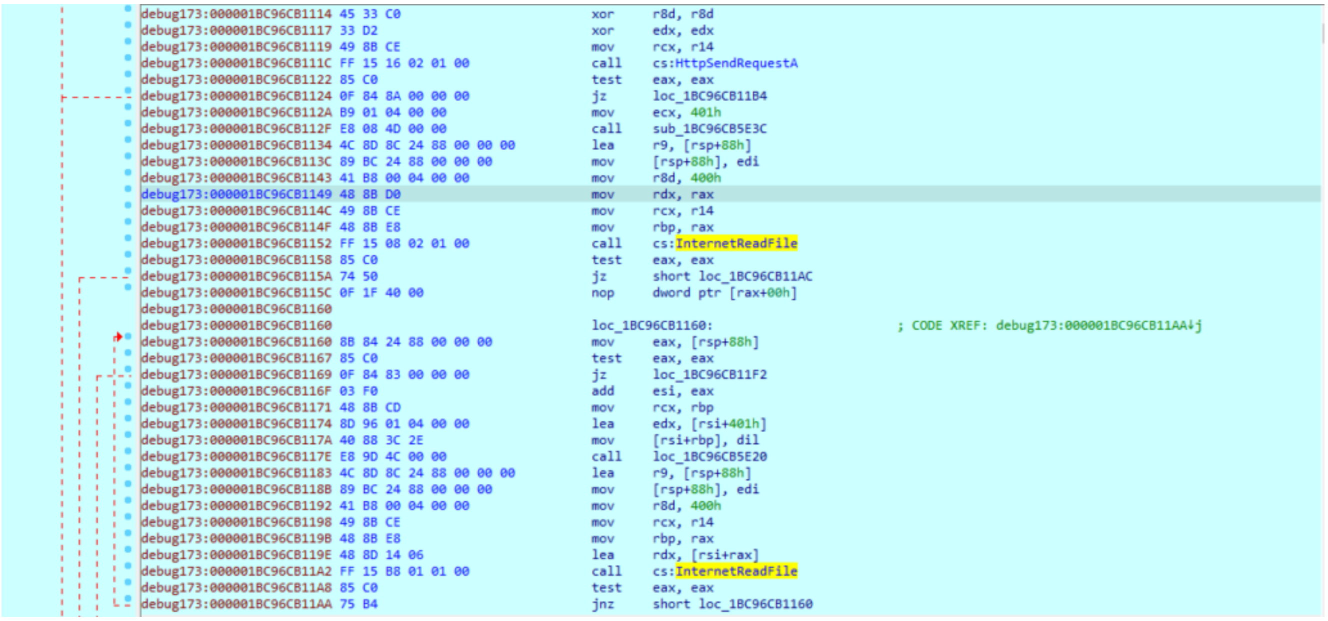


Figure 16: StrelaStealer is expecting the response from its server

We have analysed the second function statically where it reads the windows registry key, enumerates data from it and tries to locate the 'IMAP User', 'IMAP Server' and 'IMAP Password' values.

The IMAP Password contains the user password in encrypted form. The malware will call the Windows "CryptUnprotectData" API to decrypt it.

The following registry key is enumerated to steal Outlook data:

"SOFTWARE\Microsoft\Office\16.0\Outlook\Profiles\Outlook\9375CFF0413111d3B88A00104B2A6676\"

Figure 17: Outlook registry key would have been enumerated to steal data from the infected machine

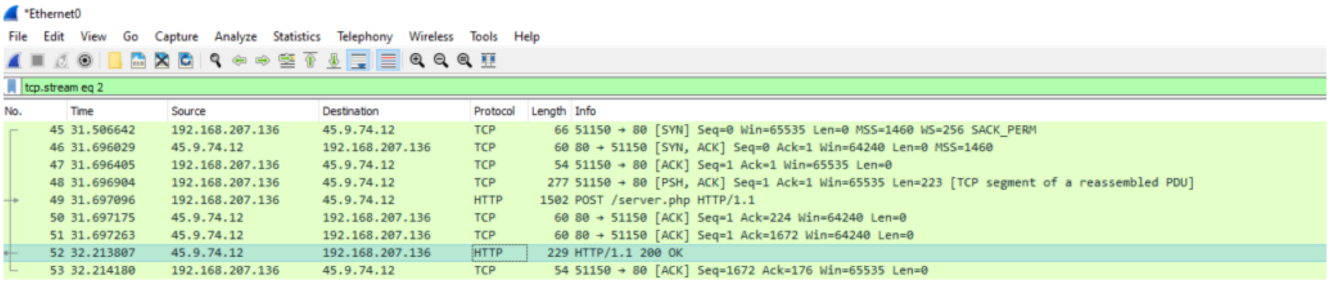


Figure 18: Network communication with server

The archive file cannot be found in any of the popular threat intelligence sharing portals like VirusTotal at the time of writing this blog.

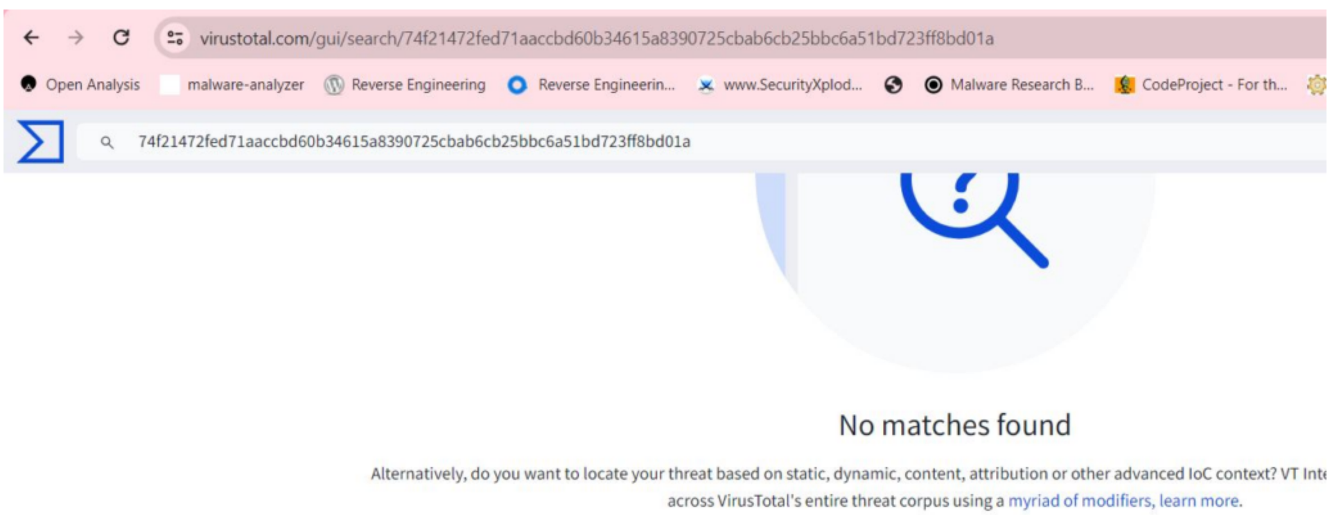


Figure 19: File is not available on VirusTotal

This threat is detected by SonicWall Capture ATP w/RTDMI . Evidence of the detection by our RTDMI engine can be seen below in the Capture ATP report for this file.

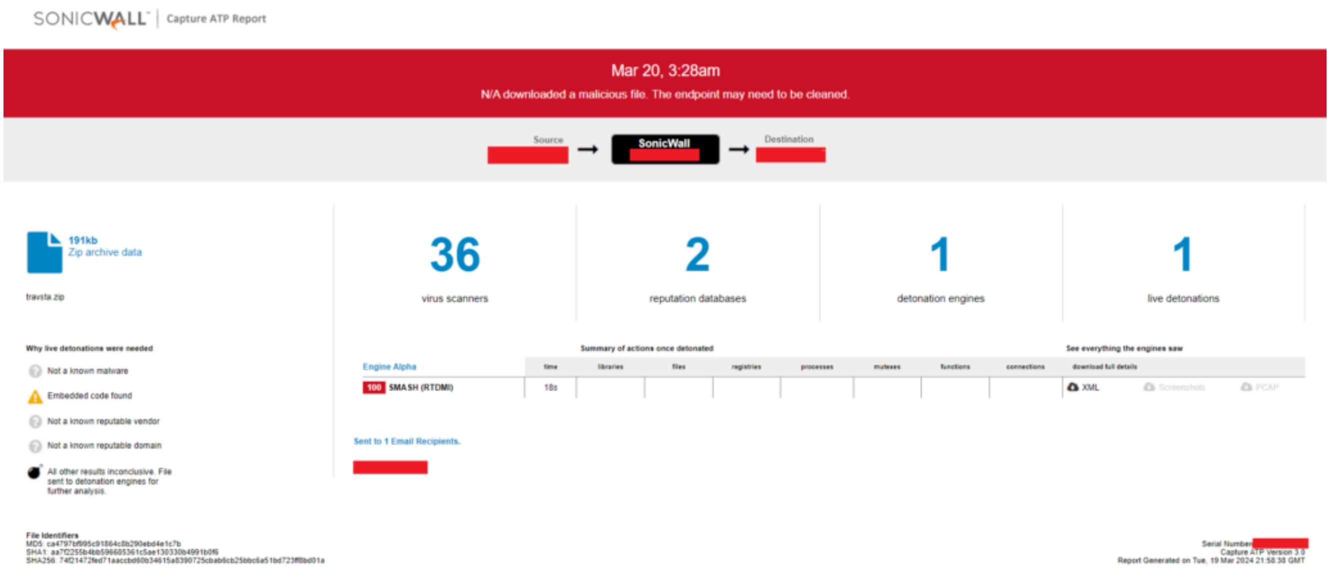


Figure 20: Capture report

IOCs

Archive file
MD5: ca4797bf995c91864c8b290ebd4e1c7b
SHA256: 74f21472fed71aaccbd60b34615a8390725cbab6cb25bbc6a51bd723ff8bd01a

JavaScript (Initial vector)
Md5 : C235CE3765F9B1606BDA81E96B71C23B
SHA256 : E083662C896C47064FD47411D47459BF4B1CB26847B5D26AEDD7F9D701CABD43

Main 64-bit executable file
MD5 : 1E37C3902284DD865C20220A9EF8B6A9
SHA256 : F2D7CF39392D394D6CCD0F9372DB7D486D4CB2BB6C3BBFD0D8BFBB6117A5E211

Injected 64-bit Payload
MD5 : 95F51B48FB079ED4E5F3499D45B7F14E
SHA256 : C02BB26582576261645271763A17DE925C2D90D430E723204BAEC82030DC889A

Server IP : “45[.]9.74[.]12”

Security News

The SonicWall Capture Labs Threat Research Team gathers, analyzes and vets cross-vector threat information from the SonicWall Capture Threat network, consisting of global devices and resources, including more than 1 million security sensors in nearly 200 countries and territories. The research team identifies, analyzes, and mitigates critical vulnerabilities and malware daily through in-depth research, which drives protection for all SonicWall customers. In addition to safeguarding networks globally, the research team supports the larger threat intelligence community by releasing weekly deep technical analyses of the most critical threats to small businesses, providing critical knowledge that defenders need to protect their networks.