

Operation Turb00 — Part 1: Analyzing and Hunting a Vidar Campaign

By Gianluca Tiepolo

Published: 2026-06-22 · Archived: 2026-10-25 02:41:30 UTC



38 min read

Jun 15, 2026

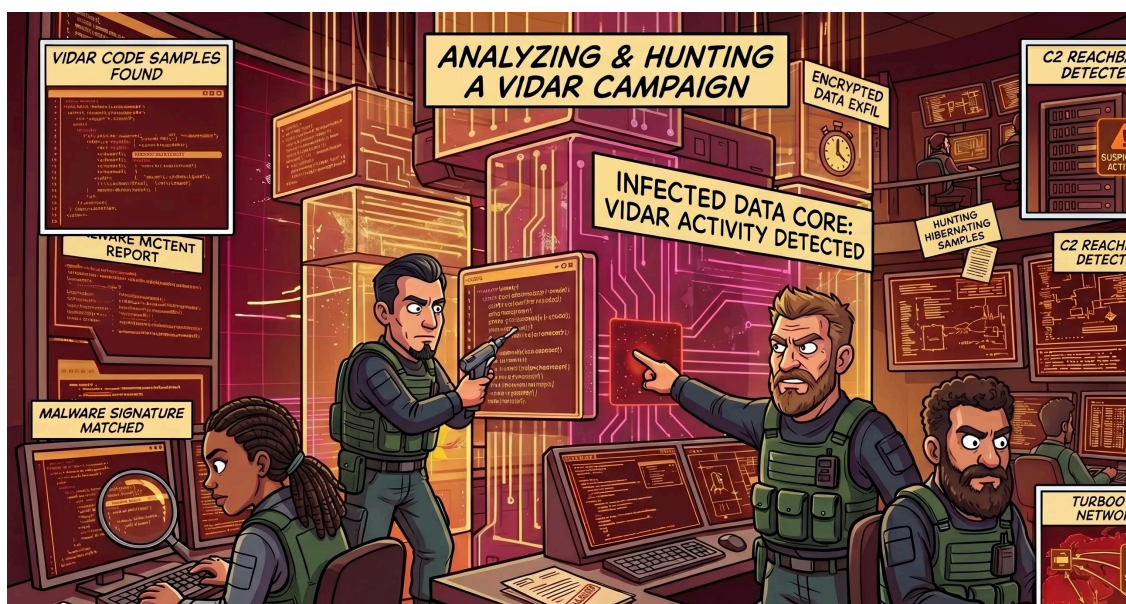


Table of contents

1. Executive Summary
2. Initial Triage
 - 2.1. *First Stop: VirusTotal*
 - 2.2. *Quick Triage: Detect-It-Easy, PEStudio and BearPE*
3. Static Analysis
 - 3.1. *The Go Loader*
 - 3.2. *The Second-Stage Payload*
4. Dynamic Analysis
 - 4.1. *Execution*
 - 4.2. *Persistence*
 - 4.3. *Exfiltration*
 - 4.4. *Command and Control*
5. Clustering and Pivoting

5.1. Campaign Clustering

5.2. Family Hunting

5.3. Adversary Infrastructure Tracking

6. Summary: Closing The CTI Loop

Detection Opportunities

IOCs

1. Executive Summary

Last week I had a long free weekend, Friday through Sunday, and I was looking for something interesting to dig into. The opportunity arrived on Friday, June 12th, at around 10am, when a researcher friend pinged me with a fresh sample he just recovered from a client's environment. The sample had triggered an AV alert at 09:20 that morning, a ML-based detection flagging infostealing-like behavior. The environment had no EDR and the AV detection was right in that annoying space where you know *something* happened but you don't know *what*: no family attribution, no prior sightings, nothing from threat feeds. Just a behavioral flag and a binary.

What followed was a weekend-long rabbit hole. What started as "let me figure out what this binary does" turned into a full reconstruction of a multi-stage malware operation: loader, payload, sample clustering, infrastructure tracking, and all.

The malware turned out to be a two-stage threat. The sample I initially analyzed is a Go loader that decrypts and executes a hidden second stage entirely in memory. That second stage is a **Vidar infostealer**, belonging to a cluster I ended up naming "**turb00**" after its command-and-control domains. From that single sample I was able to pivot outward into a validated cluster of 55 related samples, map a slice of the adversary's backend and management infrastructure, and build a layered set of YARA, Sigma and Suricata rules — plus a VirusTotal monitoring query — to track both new samples and new infrastructure as the campaign evolves.

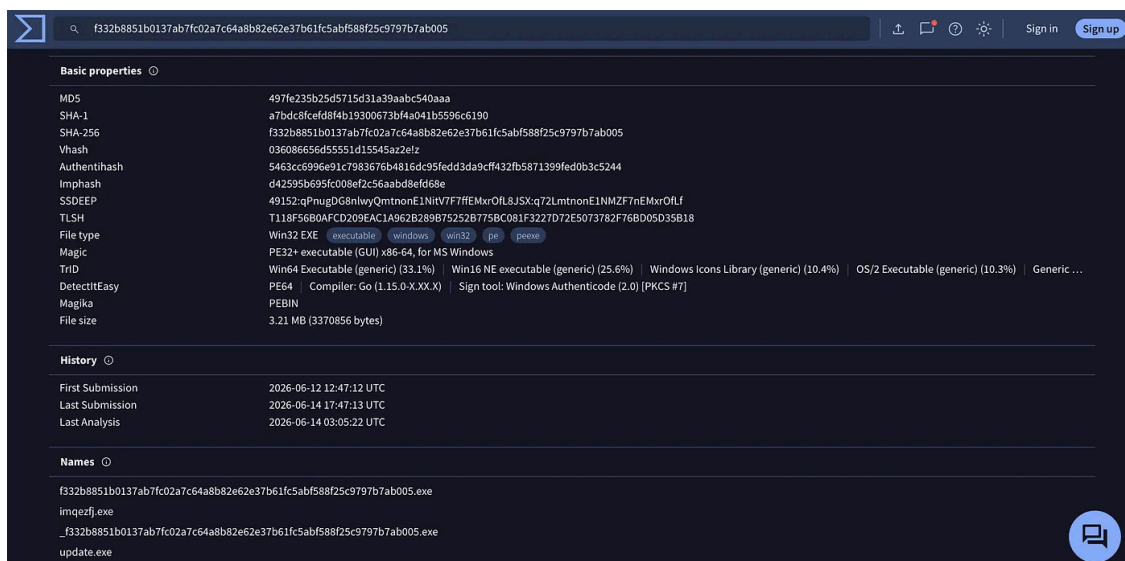
This post is a little different from my usual write-ups. Rather than just describing the malware and the campaign, I walk through the *analysis itself*, step by step — the tools I reached for, the decisions I made, the dead ends I hit, and how each finding opened the next one. If you're learning CTI, reverse engineering, or threat hunting, my hope is that seeing what researching a campaign actually looks like (including the parts that didn't work!) is more useful than a polished list of conclusions.

2. Initial Triage

2.1 First Stop: VirusTotal

The very first thing I did, as always, was search the SHA-256 [hash](#) on VirusTotal. No results. The sample simply wasn't in the database yet. That's already a signal worth noting, as either this binary had been compiled recently and distributed narrowly, or the actor was deliberately rotating samples. Either way, it was enough to make me want to understand it properly, so I uploaded the sample and, while waiting for the VT sandboxes to process it, switched to my local lab.

Press enter or click to view image in full size



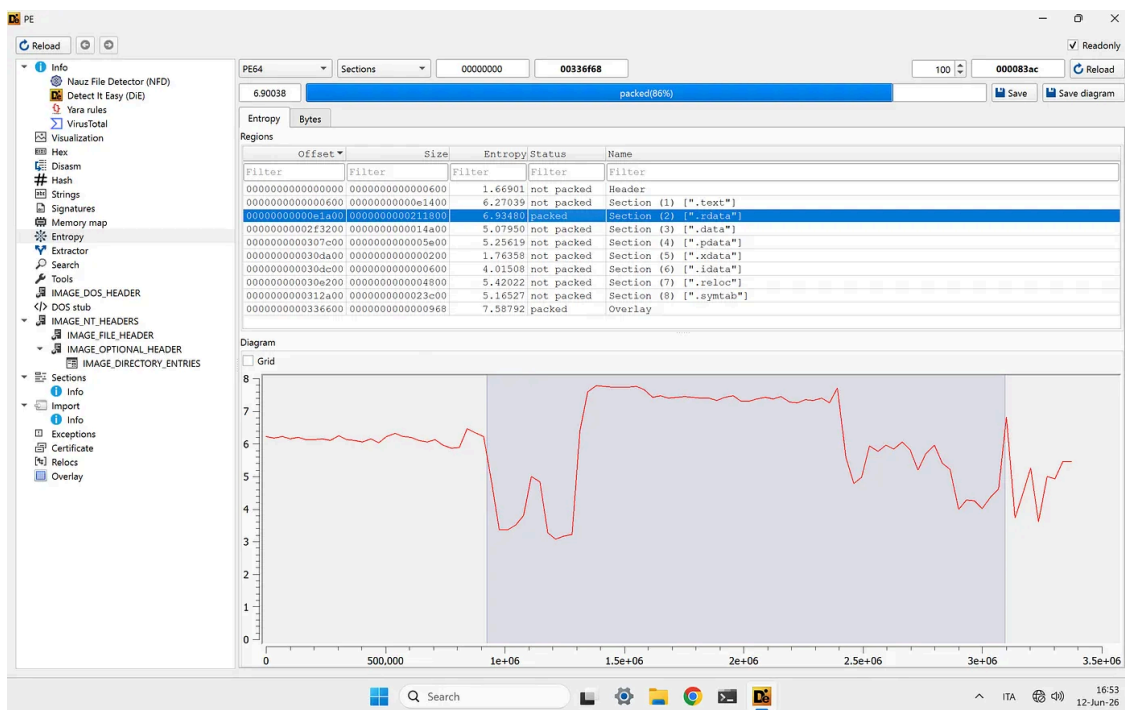
Once VT finished processing, the *Details* tab gave me the first layer of static context. File type came back as a PE32+ executable targeting Windows x64, which is consistent with modern malware targeting contemporary Windows environments. The file size was approximately 3.3 MB, which is on the larger side and already slightly suspicious for a generic executable. The compiler detection field was notable: VT identified it as a **Go binary**, based on the presence of Go runtime metadata in the file. Detection coverage was sparse, as at the time only around 12 engines flagged it, and only under generic labels like trojan or generic. No engine had a family name, so there was nothing to anchor on.

2.2 Quick Triage: Detect-It-Easy, PESTudio and BearPE

While VT was running its sandboxes, I loaded the sample into my lab environment and started working through the standard triage toolkit.

Detect-It-Easy (DiE) was first. It immediately confirmed the Go compiler identification, as the binary's section layout, string table patterns, and metadata all matched a Go-compiled executable. The entropy analysis is flagging it as packed, around 86%. The overall entropy is about 6.9, which is high but not extreme.

Press enter or click to view image in full size



For a normal Go binary, some entropy elevation is already expected because Go statically links a lot of runtime code and metadata. But here, the interesting part is the section-level entropy.

.rdata entropy 6.93480 packed
overlay entropy 7.58792 packed

The `.rdata` section is unusually high-entropy. In normal PE files, `.rdata` usually contains read-only data: strings, constants, RTTI-like metadata, import-related data, Go runtime metadata, etc. In a Go malware sample, `.rdata` can naturally be large and somewhat noisy, but entropy close to 7 strongly suggests compressed/encrypted/encoded data may be stored there. The overlay is also suspicious. Malware often uses overlays to store payloads, configs, encrypted archives, or second-stage content. Here the overlay is small, around 0x968 bytes, but entropy 7.58 is high. That could be compressed/encrypted config, junk, anti-analysis data, or a small encrypted blob.

PEStudio gave me the imports and strings view. The import table was almost empty, as the binary imported only a handful of standard Windows APIs from `kernel32.dll` and the Go runtime support libraries. Nothing like `VirtualAlloc`, `WriteProcessMemory`, or any other classic shellcode-loader APIs was visible statically. That's a deliberate obfuscation choice: resolve everything at runtime so static analysis tools have nothing to fingerprint. The strings view was similarly sparse, mostly Go runtime boilerplate, a handful of Go package paths. No URLs, no domains, no registry keys, no obvious IOCs.

I confirmed the same data using **BearPE**. At this point the picture was consistent: a Go binary with a near-empty import table and no readable IOCs. The high entropy suggested an embedded payload or encrypted blob.

The VT sandbox results came back around the same time, and they added behavioral context: the sandboxes observed the process allocating executable memory (`VirtualAlloc` with `PAGE_EXECUTE_READWRITE`), then

transferring control into it, which is the classic in-memory load pattern. The sandbox also captured network activity and a number of dropped files. Time to open Ghidra and have an in-depth look at this sample.

3. Static Analysis

Static analysis was where the bulk of the weekend went. There were two distinct targets: the Go loader, which was my first sample, and the second-stage PE that I extracted from it. I'll cover each in turn, in the order I worked through them.

3.1 The Go Loader

The first thing I always do when opening a new binary in Ghidra is listing the functions to get a sense of the landscape. In this case, the output immediately told me what I was dealing with. The binary had the full Go runtime symbol set intact, such as `runtime.main`, `runtime.newproc`, `runtime.goroutine`, all the usual scaffolding. But on top of that, there was a small main package with **18 functions carrying randomized names**: `main.wbkyc`, `main.iolmsfzesqsge`, `main.xuffawcnutyfhdewhgd`, `main.ikkux`, `main.kanvlgloovtnjyy`, and so on. A small, name-mangled main package sitting on top of an otherwise intact Go runtime is the fingerprint of a **garble-style obfuscator** — the tool randomizes symbol names at compile time, so the runtime survives untouched but the author's own functions become meaningless strings.

The second thing I noticed, as soon as I started decompiling functions, was an **opaque-predicate junk pattern** appearing in every single routine. It looked like this:

```
uVar = 0x2b5626;
while (uVar != 0) {
    if (((uVar & 3) != 0) && ((ushort)uVar >> 9 < 0x31)) {
        if ((ushort)uVar == 0x552e) {
            runtime_duffzero_140074974(...);
            local_988 = 0xc00c000000000000;

        } else if (...) {
            runtime_duffcopy_140074b80(..., &DAT_140144600);
        }
    }
    uVar = 0;
}
```

This block, essentially, does nothing. The loop always exits immediately because `uVar` is set to 0 at the bottom of the body. The branches inside write floating-point constants into stack locals that are **never read** by the function's actual logic. The only purpose is to inflate the control-flow graph and waste analyst time. Once I recognized the constant `0x2b5626` as the marker for these blocks — it appears at the start of every one of them — I could mentally strip them on sight. That turned out to be essential, because they wrap the real logic in every function in the binary.

Finding the string decoder

Rather than randomly picking functions and decompiling them, I started where Go always starts execution:

`main.init` . This is the package initialization function, and it runs before `main.main` . Experienced Go reverse engineers know that `init` is often the most informative function in a malicious Go binary, because it's where global state gets set up.

In this case, `main.init` was the key that unlocked everything. Decompiling it (after stripping the opaque-predicate junk) revealed this pattern:

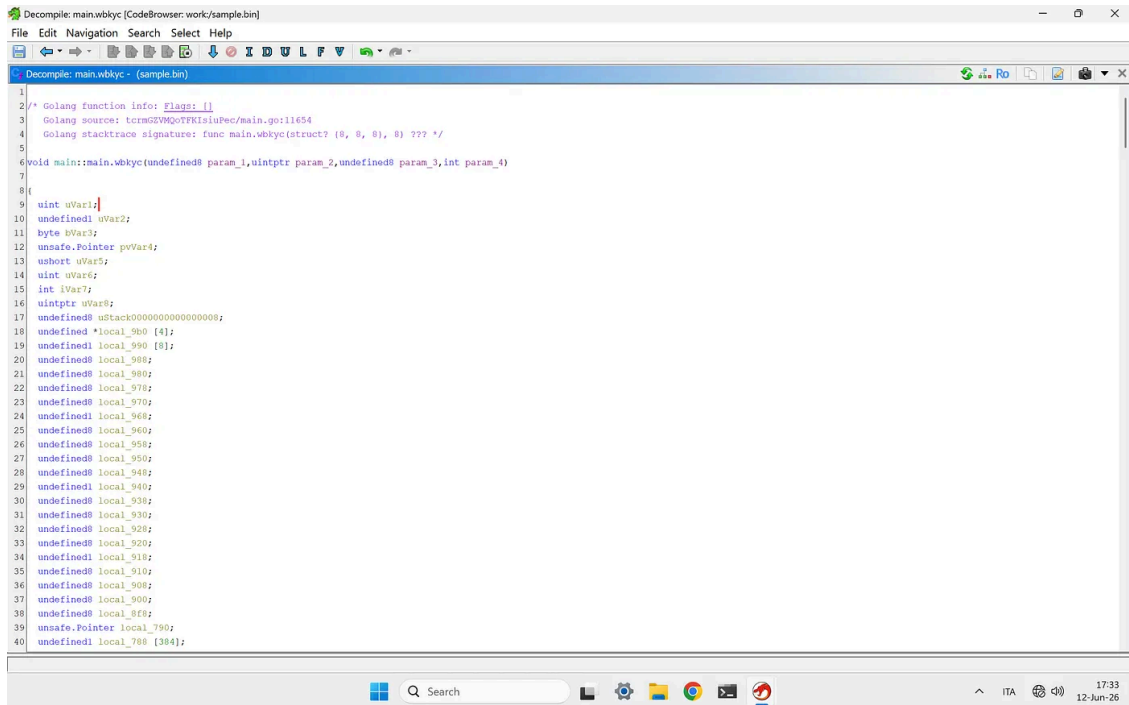
```
local_3c = 0xe6a0aabe;
local_38 = 0xe163ac586625d8cf;
auVar1 = main_wbkyc(&local_3c, 0xc, 0xc, 0xcb38d);
sVar2 = runtime_slicebytetostring(...);
runtime_bss = runtime_newobject(&syscall_LazyDLL_type);
(runtime_bss->Name) = sVar2;
```

Three things became clear at once. First, `main.wbkyc` **is the string decoder** — it's called with a buffer and a key and returns something that feeds a `syscall.LazyDLL` object, which is Go's mechanism for dynamically loading Windows DLLs. Second, the **ciphertext is stored inline as stack immediates**, not in a separate data section, but pushed directly onto the stack before each call, making bulk extraction with a strings scan impossible. Third, the **decryption key is the 4th argument**, and it's different for every string (`0xcb38d` for the first, `0x2cdc0` for the second). The decoded results feed `syscall.LazyDLL` and `syscall.LazyProc` , meaning the loader resolves Windows APIs entirely at runtime by decrypting their names on demand.

Reversing the cipher

With `main.wbkyc` (`0x1400ca600`) identified as the central routine, I decompiled it fully. It was large and saturated with the opaque-predicate junk from above. After stripping all of that, the real logic reduced to a clean **four-stage byte cipher** operating on a heap copy of the input buffer, keyed by the 32-bit fourth argument K:

Press enter or click to view image in full size



- **Stage A — per-index substitution:** The function iterates over every byte. For even indices: $b[i] = (b[i] - (K \% 0x35) - 0x11*i) \& 0xff$. For odd indices: $b[i] = ((31*i) \wedge (K \% 0x61) \wedge b[i]) \& 0xff$.
- **Stage B — full buffer reversal:** A two-pointer swap from both ends, reversing the entire buffer in place. Simple, but critically important for the payload extraction (more on that shortly).
- **Stage C — adjacent-pair swap:** Swap bytes at positions (0,1), (2,3), (4,5), and so on through the buffer.
- **Stage D — finalize:** Two passes over the buffer: first $b[i] = (b[i] - ((K \gg 8) \& 0xff)) \& 0xff$, then $b[i] = (b[i] \wedge (K \& 0xff) \wedge i) \& 0xff$.

In a Python reimplementation, it looks like this:

```
def decode(buf, K):
    b = bytearray(buf); n = len(b)
    for i in range(n):
        if i % 2 == 0: b[i] = (b[i] - (K % 0x35) - 0x11*i) & 0xff
        else:          b[i] = ((31*i) ^ (K % 0x61) ^ b[i]) & 0xff
    b.reverse()
    i = 0
    while i+1 < n: b[i], b[i+1] = b[i+1], b[i]; i += 2
    for i in range(n): b[i] = (b[i] - ((K>>8)&0xff)) & 0xff
    for i in range(n): b[i] = (b[i] ^ (K & 0xff) ^ i) & 0xff
    return bytes(b)
```

Crucially, the key *K* never appears *inside* *wbkyc*, but it arrives as an argument. The function body is completely identical regardless of which string it's decoding; only the caller's key differs. This is a deliberate design choice that makes the cipher slightly harder to fingerprint statically.

I immediately validated the decoder by feeding the ciphertext from `main.init` with keys `0xcb38d` and `0x2cdc0` and got back `kernel32.dll` and `GetProcAddress` — clean Windows API names. That confirmation was the green light to trust the decoder for everything else.

Mapping the execution chain

With the decoder working, `main.main ( 0x1400ce8a0 )` became tractable. I grepped its decompilation for calls to `main_wbkyc` and found **12 string decode calls** plus one outlier that was significantly different in scale:

```
puVar19 = &DAT_140146622;
for (iVar14 = 0x21846; iVar14 != 0; iVar14--) { *dst++ = *puVar19++; }

main_wbkyc(local_12b0, 0x10c237, 0x10c237, 0x1062c);

local_13a8 = main_kanvlgloovtnjjy(local_12b0, len, cap);

auVar30 = main_ikkux(*local_13a8, local_13a8[1], 0x3000, 0x40);

syscall_Syscall(base + entry_offset, 1, 2, 3, 4);

runtime_newproc(fn);
```

The execution chain was now completely clear:

- **main.ikkux** decompiled to a dynamic-call wrapper that decrypted `kernel32.dll` (key `0xcb38d`) and `VirtualAlloc` (key `0x836a0`) and invoked them via `syscall.LazyProc.Call` — the `0x3000` and `0x40` arguments are `MEM_COMMIT|MEM_RESERVE` and `PAGE_EXECUTE_READWRITE`, confirming RWX allocation.
- **main.kanvlgloovtnjjy** checked `*(short*)payload == 0x5a4d` (the MZ DOS magic), read `e_lfanew`, and validated `0x4550 ( PE\0\0 )` — a manual PE loader.

That last detail was important, as it told me the output of `wbkyc` with key `0x1062c` *must* be a valid PE, which gave me an immediate ground-truth test for the payload decryption.

Decoding the 12 inline strings revealed the decoy's purpose: the loader was printing a fake maze-generation and BFS pathfinding program to the console, complete with Cyrillic text, `ГЕНЕРАТОР ЛАБИРИНТА · ПОИСК ПУТИ (BFS)`, and maze-drawing glyphs. While all of this is printing to the console and looking benign, the real payload is running in memory. The Russian-language theme is a weak attribution signal, but more importantly it's an effective sandbox confuser.

Extracting the payload

This is where I hit the first real obstacle of the weekend, and it's worth documenting in detail because it illustrates

a non-obvious property of the cipher. I initially pulled the ~1 MB encrypted blob from the data section, starting at the Ghidra label `DAT_140146622`, with length `0x10c237`. I ran it through my Python decoder with `K=0x1062c`. The output was garbage, definitely not an MZ header, not anything parseable.

The cause was that the cipher **reverses the entire buffer**. This means the first output bytes are derived from the *last* input bytes. The consequence is that the decoded result is acutely sensitive to both the exact start offset and the exact length of the input. An off-by-N error anywhere doesn't corrupt a few bytes, it scrambles the entire output because everything is reversed and then XOR'd with a position-dependent key. I went back to the decompiler and looked more carefully at the copy loop in `main.main`. The copy destination pointer on the stack sat 7 bytes earlier than the Ghidra-labelled `DAT_140146622`, meaning the blob's true start in the file was `0x14014661b`, not where Ghidra had conveniently placed the label.

To confirm this, I wrote a quick brute-force: for each candidate (start offset $\pm$ some range, various lengths), compute only `output[0]` and `output[1]` and check whether they equal `0x4D` and `0x5A` (the MZ magic). This is $O(1)$ per candidate — you only need two bytes of the reversed buffer to know if you've got the right alignment, so the brute-force was fast. The unique solution was **start = blob_address - 7, length = 0x10c237**.

Re-running the full decode with the correct offset and length produced a valid MZ, a clean PE32+ x86-64 Windows executable. Stage one was solved, as I now had the second-stage payload.

3.2 The Second-Stage Payload

I loaded the payload into Ghidra and ran the auto-analysis. The [sample](#) presented two deliberate obstacles from the start: no import table and only 7 defined strings, all of which were PE section names (`.text`, `.rdata`, `.data`, `.reloc`). That's it. Every meaningful string — every API name, every URL, every command — was absent from the binary as it sat on disk.

The only entry point Ghidra gave me was entry at `0x5c00028b0`. Decompiling it revealed opaque-predicate math leading to `FUN_5c00e08a0` (initialization checks) and then to `FUN_5c00d52bc`, which was the real main. I was going to have to defeat both the API resolution and the string obfuscation before this binary would tell me anything useful.

Defeating the API hashing

I started by decompiling main and its direct callees and looking for functions called repeatedly by many others. One function, `FUN_5c00e8e20`, was called approximately 10 times from within main and its immediate children. Its body resolved a function pointer via a sub-call `FUN_5c00e1934` (`0x39f62520`) and then invoked it.

The 4-byte constant `0x39f62520` being passed as an argument to a resolver was the signature of a **hash-based API resolution wrapper**. I followed the chain:

```
FUN_5c00e8e20 (wrapper, hardcoded hash)
  → FUN_5c00e1934 (dispatcher)
    → FUN_5c00e1104 (PE export-table walker)
      → FUN_5c00e058c (the hash function itself)
```

`FUN_5c00e1104` was a textbook manual exports resolver: validate MZ/PE, iterate `AddressOfNames`, hash each export name, match against the target hash, then follow `AddressOfNameOrdinals` to `AddressOfFunctions` to get the function pointer.

The actual hash function was decompiled to:

```
uint32_t hash(const char *name) {
    uint32_t h = 0x3aff8d85;
    for (; *name; name++)
        h = (uint32_t)((*name ^ h) * 0x010006bb);
    return h;
}
```

This is structurally equivalent to FNV-1a, but with a custom offset basis and custom multiplier. Standard FNV-1a uses offset basis `0x811c9dc5` and prime `0x01000193`, but here the actor changed both. I implemented this custom version in a Python script and verified it: `hash("Sleep")` should equal `0x39f62520`, the constant hardcoded in the wrapper. It did, so algorithm confirmed!

At this point in the analysis, I had a working hash function and knew every API was identified by a 4-byte constant somewhere in the binary. The naive approach would be to decompile all ~52 hash wrappers one by one. I tried this, but it was too complicated and time consuming, so I changed approach entirely: take a large wordlist of Windows API names, hash every one with the custom FNV-1a, and scan the binary's raw bytes for matching little-endian dwords. This is $O(n)$ in the file size, near-instantaneous. Essentially, we're brute forcing API names!

The result was revealing: 101 named APIs recovered in one pass, grouping cleanly into capability buckets:

- DPAPI-based credential theft (`CryptUnprotectData`)
- File system harvesting (`CreateFileA`, `FindFirstFileA`, `FindNextFileA`, `GetLogicalDrives`, `ReadFile`)
- GDI-based screen capture (`GetSystemMetrics`, `CreateCompatibleDC`, `CreateCompatibleBitmap`, `BitBlt`)
- Full WinHTTP exfiltration stack (`WinHttpOpen`, `WinHttpConnect`, `WinHttpOpenRequest`, `WinHttpSendRequest`, `WinHttpReceiveResponse`, `WinHttpReadData`)
- Process injection (`NtAllocateVirtualMemory`, `NtWriteVirtualMemory`, `NtProtectVirtualMemory`, `NtCreateThreadEx`, `CreateRemoteThread`, `VirtualAllocEx`, `WriteProcessMemory`)
- Sandbox/timing evasion (`NtDelayExecution`, `GetTickCount`, `QueryPerformanceCounter`).

Some of these hash constants are worth calling out as specific YARA anchors: `0xa493fb4f` maps to `CryptUnprotectData`, `0xd7ea5966` to `BitBlt`, `0x45823614` to `WinHttpSendRequest`, `0x565a68a3` to `CreateRemoteThread`, and `0x331b9a32` to `NtCreateThreadEx`. The combination of those five in a single binary is a high-confidence indicator.

Defeating the string obfuscation

With the API picture clear, I turned to the string problem. I needed a function that I knew produced a string and could follow the string-building logic backward from there. I picked `FUN_5c00a0838`, which appeared to be a log-

message builder based on its position in the call graph and the size of its stack frame. Its decompilation showed it writing approximately 99 bytes to the stack and then calling `FUN_5c00ca0c0(8buf, 99, dst, 0x14)`. That callee was the string decoder.

I went in expecting a simple XOR or substitution cipher, but what I found was significantly more interesting:

`FUN_5c00ca0c0` was a **dispatch-table interpreter**. For each source byte `b`, it looked up `handler = table[DAT_5c010b0c0 + b*8]` and called it, passing a small state struct. The table had 256 slots, one per possible byte value. What I'd found was essentially a small virtual machine (VM) — not a system emulator — but a *string-obfuscation VM*: a tiny custom interpreter with its own instruction set. Each byte of the encoded input is treated as an instruction (“opcode”) that mutates an internal state until a plaintext string falls out the other end. Malware authors like this technique because there's no fixed cipher to fingerprint; to recover the strings, you have to reverse the interpreter itself.

The problem is that in the file on disk, the dispatch table was all zeros, as it's built at runtime. The initializer `FUN_5c00c9ee8` populated only **14 of the 256 slots** at startup. To understand the VM, I needed to reverse the initializer to find which 14 opcodes were being registered and what each one did. I decompiled `FUN_5c00c9ee8` and extracted all 14 written addresses, then promoted each raw code label into a proper Ghidra function. The 14 handlers formed a small **accumulator machine**:

Press enter or click to view image in full size

Opcode	Operation
0x18	EMIT : <code>b = src[idx++]; out = ACC ^ b; emit(out); ACC ^= out</code>
0x17, 0xb2	<code>ACC = rol8(ACC, 5)</code>
0x19	<code>ACC ^= last_emitted</code>
0x1f	<code>ACC = (ACC * 0x69) & 0xff</code>
0x3a / 0x68 / 0xb5	<code>ACC -= src[idx++] / ACC += src[idx++] / ACC ^= src[idx++]</code>
0x71	<code>ACC += 0x60</code>
0x72	<code>ACC = ~ACC</code>
0x88	<code>ACC = SBOX[ACC]</code> (256-byte permutation at <code>DAT_5c0107b40</code>)
0x8d	<code>ACC ^= 0x1f</code>
0xb0	<code>ACC ^= (output_index & 0xff)</code>
0x62	<code>ACC ^= (CONST >> ((output_index & 3) * 8)) & 0xff</code>

Medium doesn't support tables yet, sorry 😞

The seed and the `CONST` value come from `FUN_5c00c9e40`, which returns `0x311b7f33` on the normal path, and `0xDEADBEEF` only if the PEB looks tampered, which is another layer of anti-analysis. The accumulator seeds to

ACC = 0x33 and CONST = 0x311b7f33 .

One element worth highlighting for detection purposes: the 256-byte S-box at DAT_5c0107b40 (beginning CC EA 32 C5 F3 62 ...) is a stable, key-independent fingerprint of this malware family. It doesn't change between runs, it's large enough to be distinctive, and it's unlikely to appear in legitimate software. It became one of the anchors for the YARA rules.

I ported all 14 opcodes, the S-box, and the seeding logic to Python and validated against the 99 bytes from FUN_5c00a0838 . The output confirmed the VM was working. I then found all callers of FUN_5c00ca0c0 , extracted each caller's stack blob, and bulk-decoded approximately 30 strings. What came out read like the malware's own internal documentation:

Press enter or click to view image in full size

```

tiepolo - /dev/tty000
tiepolo@MacBookAir Downloads % cat /tmp/strings_out.txt
5c00cc7b [ 13] 'Tw38[Nx0eyQx8140z-'
5c00df88 [ 22] 'Dead drop: %s (sw: %s)'
5c0012994 [ 23] 'Config parse failed: %s'
5c0013070 [ 18] 'Connect failed: %s'
5c0014068 [ 33] 'C2 unavailable, attempt %lu/%lu'
5c0015080 [ 27] 'cmd.exe /c rmdir /s /q "%s"'
5c0015848 [ 17] 'cleanup udd: %s'
5c0016e90 [ 13] 'MapViewOfFile'
5c001a55c [ 50] 'Software\Microsoft\Windows\CurrentVersion\Explorer'
5c001b464 [ 15] 'W81X041X0081'
5c001c598 [ 64] '4f8a2b1c9d3e7f6a0b5c1d8e2f9a0b7c6d3e0f1a2b5c8d9e0f1a2b3c4d5e6f7a'
5c0021054 [ 13] 'IS\X100\vr7uXxt-N'
5c0023840 [ 25] 'M===== Sandbox Check ====='
5c0025c3c [ 21] 'sb: debugger FAIL'
5c0026298 [ 19] 'sb: peb.Flags OK'
5c00274e0 [ 25] 'sb: cpus %lu FAIL'
5c0028528 [ 19] 'sb: modules OK'
5c002a064 [ 22] 'sb: user %s OK'
5c002b314 [ 19] 'sb: av_sandbox OK'
5c002b868 [ 22] 'sb: pc %s OK'
5c002c50c [ 54] 'sb: uptime %lu sec d\vx80\vx94 too fresh, waiting %lu ms'
5c002d600 [ 34] 'sb: uptime %luh %lum %lus %s'
5c002e3cc [ 34] 'sb: uptime %luh %lum %lus %s'
5c002f068 [ 29] 'sb: uptime %lum %lus %s'
5c0031304 [ 13] '\x00f\x12Ww-Ja-Ee'
5c0037658 [ 13] '5x07/0x0f-e01e-ir'
5c003a41c [ 20] 'parse_response: invalid token'
5c003d5f4 [ 25] 'Config %s: empty response'
5c00400c8 [ 15] 'AV: Trend Micr'
5c004050c [ 13] 'AV: Kaspers'

# total unique strings: 30
tiepolo@MacBookAir Downloads %

```

A handful of additional strings used a second, inline arithmetic scheme, so I decoded those by hand where needed, which resulted in: .exe , .dll , .ps1 , powershell.exe -ExecutionPolicy Bypass -WindowStyle Hidden -NoProfile -File "%s" , and cmd /c start rundll32.exe "%s" .

The slickest defensive measure in this malware payload is actually wired directly into the string decoder. Recall that the VM's CONST seed (0x311b7f33) comes from FUN_5c00c9e40 . That function inspects the PEB, and on a clean host it returns 0x311b7f33 , but if it decides the process is being tampered with (debugger artefacts in the PEB), it returns 0xDEADBEEF instead. Because CONST feeds every string the VM decodes, a tampered PEB doesn't crash the malware or pop an obvious "debugger detected" message, but it still silently poisons every decoded string into garbage. It's an elegant time-sink, as an analyst single-stepping the binary in a debugger sees the string routine "work" and produce nonsense, with no clear signal that an anti-debug check is the cause.

Cracking the malware's configuration

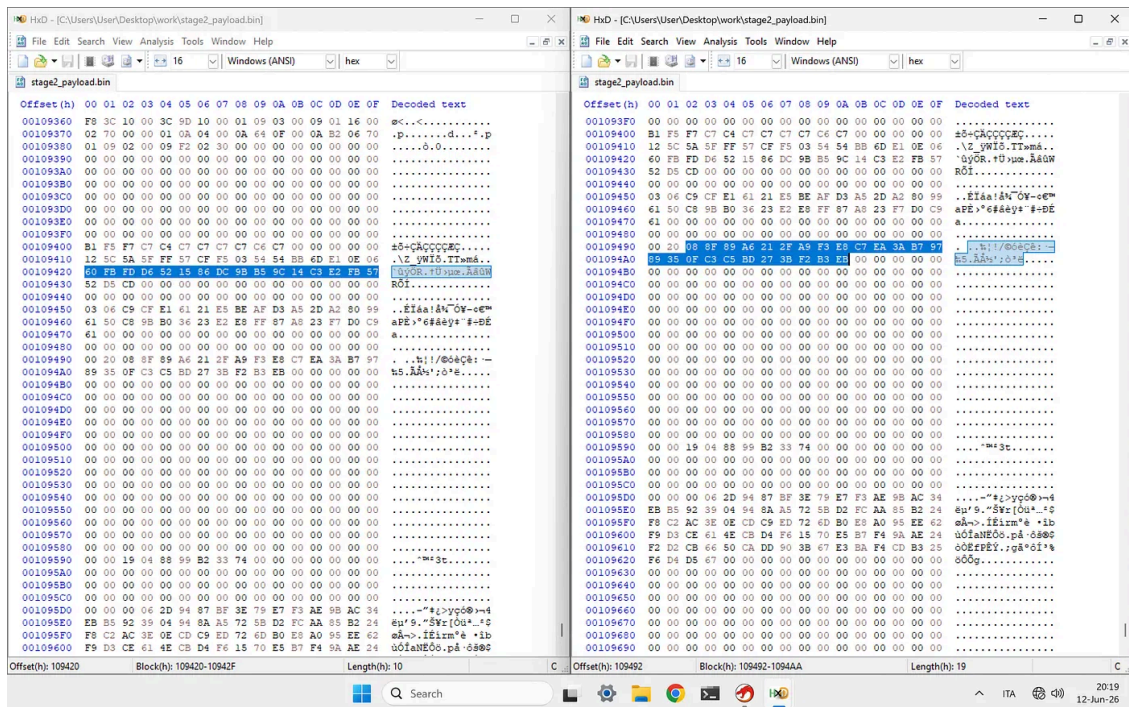
With strings decoded and APIs mapped, I traced the C2 communication path by following cross-references from the WinHTTP API wrappers. This led to the dead-drop log string Dead drop: %s (sw: %s) in FUN_5c00fbb84 ,

and from there to the fetch-and-extract function `FUN_5c00fad70` , and finally to the config initializer and decryptor:

```
for (i = 0; i < srclen; i++)
    dest[i] = src[i] ^ key[i % 16];
```

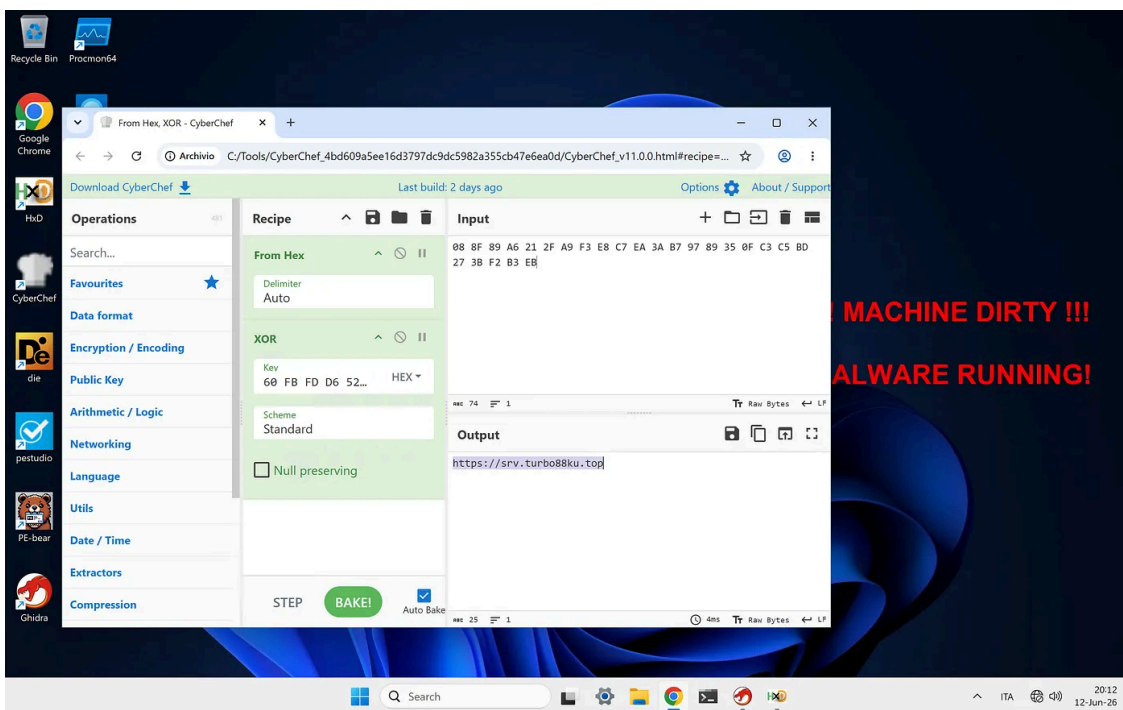
The decryptor is a **repeating 16-byte XOR**, applied field by field (the key position resets to 0 at the start of each field, not a continuous stream). The XOR key in this sample was `60 FB FD D6 52 15 86 DC 9B B5 9C 14 C3 E2 FB 57` , as you can see in the following image:

Press enter or click to view image in full size



Decrypting the .data config block yielded everything I'd been looking for, especially the primary hardcoded C2 server:

Press enter or click to view image in full size



I then proceeded to extract the full configuration, including the campaign ID and UAs used in command-and-control network requests:

Press enter or click to view image in full size

Field	Value
C2 URL	<code>https://srv.turbo88ku[.]top</code>
Campaign / Bot ID	<code>f22734cb4f99abb603ffd6d4d2474226</code>
User-Agent #1	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:156.0) Gecko/20100101 Firefox/156.0
User-Agent #2	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Safari/537.36 Edg/147.0.0.0

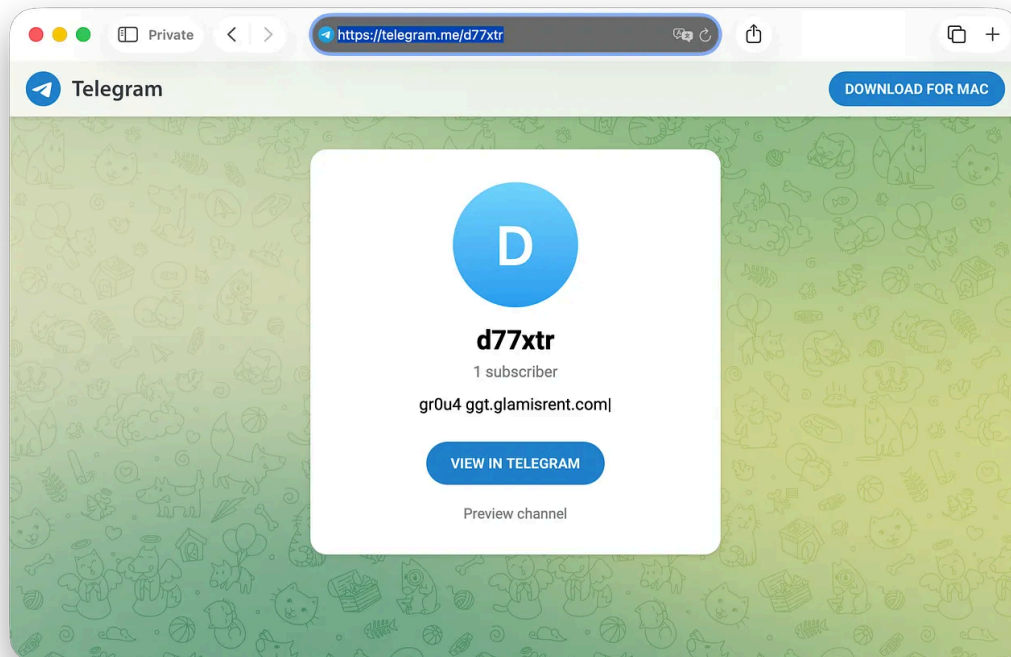
The C2 domain — `turbo88ku[.]top` — is where the operation gets its nickname.

It's worth pausing on User-Agent #1: it advertises `Firefox 156.0`, a version that doesn't exist yet. Real browsers don't invent version numbers, so a hardcoded, impossibly-high version is a reliable tell (the second UA is similarly ahead of reality). Better still, these strings are *static* (baked into the encrypted config) so every beacon from this family carries the same impossible User Agent. That makes it an excellent, low-false-positive anchor for a network rule: a Suricata/Snort signature on the exact User-Agent will flag this family's C2 traffic regardless of which domain it's rotated to.

Further analysis allowed me to trace the C2 dead-drop functions, which revealed a classic Vidar behavior that helped confirm family attribution. The stealer queries two hardcoded public pages — `telegram[.]me/d77xtr` and

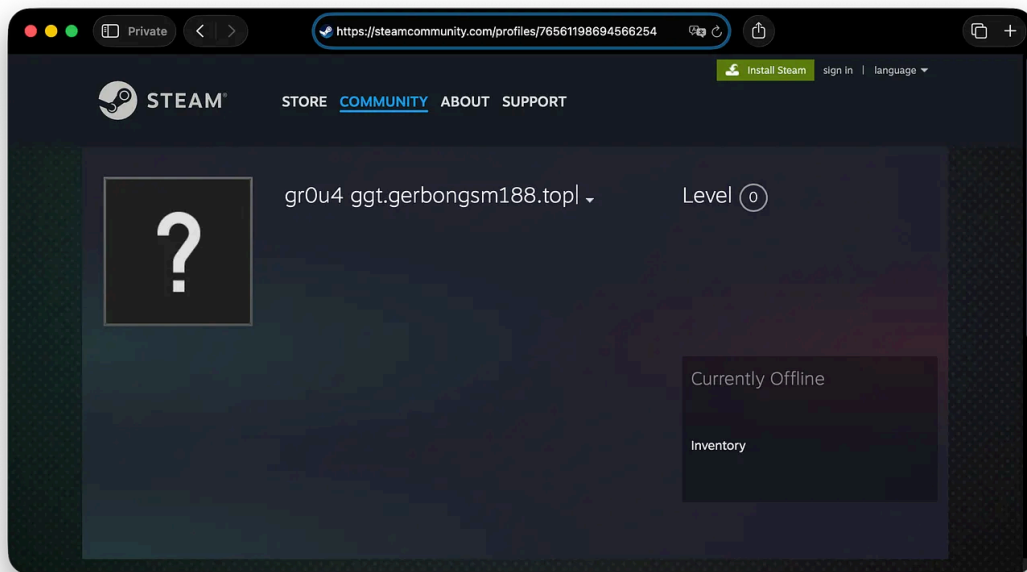
steamcommunity[.]com/profiles/76561198694566254 — and searches each for a plaintext marker string, which in my sample is gr0u4 . Everything after that marker, up to a | character, is extracted verbatim and wrapped as https://<value> to form a backup C2 URL.

Press enter or click to view image in full size



On the observed pages at the time of analysis, those values were ggt.glamisrent[.]com (Telegram) and ggt.gerbongsm188[.]top (Steam). These dead-drop pages weren't contacted during our run, as the primary C2 was alive and responding. However, if the primary C2 is blocked, the stealer silently fails over to whatever the actor has posted on their Telegram channel or Steam profile.

Press enter or click to view image in full size



To make all of this repeatable for future samples, I wrote a quick Python script that extracts and prints the C2 config from a sample. Rather than hardcoding the XOR key, the tool recovers it automatically by exploiting two properties:

- The zero-padding between config fields makes the key “leak through” as a visible 16-byte repeating pattern
- The known-plaintext crib `Mozilla/5.0`, which allows unambiguous key recovery.

The tool then decrypts the config, prints it, and runs the hash-wordlist scan to emit the capability map, all from a raw binary, no hardcoded offsets.

That was the end of the static analysis phase. After roughly a day of work, I had the loader fully understood, the payload extracted and decrypted, 101 APIs mapped, ~30 internal strings decoded, the config decrypted, and a few Python tools to automate everything. The family attribution was medium-confidence Vidar at this point, which the subsequent AV detections on the unpacked payload (`Win64/Vidar.D`) confirmed. Time to move to dynamic analysis.

4. Dynamic Analysis

Static analysis tells you what a binary *can* do. Dynamic analysis tells you what it actually *does* when it runs. After a day in Ghidra reconstructing the loader and payload from the inside out, I definitely wanted to see the malware execute in a controlled environment and validate my findings against real runtime behavior. I detonated the sample in my lab — a clean Windows 11 x64 VM, internet-isolated except for the C2 channels I wanted to observe — with **Procmon**, **Process Hacker**, and **Wireshark** running throughout. I also correlated against a Joe Sandbox run of the same sample for additional behavioral signatures and network telemetry.

The short version of what happened: the malware runs, passes its own sandbox checks, decrypts and executes the second stage in memory, harvests credentials and browser data from the simulated user profile I’d seeded, stages

everything as SQLite databases in a temp directory, exfiltrates over HTTPS, drops an additional persistent loader, installs a Windows service, and then cleans up after itself. The whole active phase took under two minutes.

4.1 Execution

The process tree that unfolded was a four-hop chain, which was more elaborate than I'd expected from the static analysis alone. The static work had shown the loader decrypting and executing the Vidar PE in memory, but I hadn't anticipated the additional dropped components. Here's what Procmon and Process Hacker captured:

```
sample.exe ← the delivered loader; Vidar core runs in this process
├── cmd.exe
│   ├── conhost.exe
│   └── loader.exe ← small dropped loader/injector, 17.9 KB
│       └── rundll32.exe ← proxy-executes agt.dll
```

The initial process is the Go loader I'd been analyzing. It runs the Vidar stealer core in-process, exactly as the static analysis predicted. The stealer does its credential collection work here, in memory, without writing the Vidar PE to disk. What I hadn't seen coming was the next step: after the stealer finishes its primary collection, it drops a small secondary executable and uses `cmd /c start "C:\Users\user\AppData\Local\Temp\[random]\loader.exe"` to launch it.

The dropped binary is tiny (only 17.9 KB) which is a strong signal that it's a loader or injector rather than a self-contained payload. It does two things:

- It launches `rundll32.exe` to proxy-execute `C:\ProgramData\CorsairUtil\agt.dll` via its `ExecuteTask` export
- It installs the persistence mechanism covered in the next section. The use of `rundll32` for DLL execution is a classic living-off-the-land technique (T1218.011), as the DLL runs under a signed Windows binary, which helps it blend into process telemetry.

4.2 Persistence

This was the part of the dynamic analysis that most surprised me relative to what I'd found statically. The static work on the Vidar PE had documented its self-delete behavior (`cmd.exe /c rmdir /s /q "%s"`), which is consistent with the classic non-persistent Vidar model — run, steal, clean up, disappear. But this sample doesn't stop there, as it installs a persistence mechanism that survives the self-delete and outlasts the initial infection.

The Vidar binary drops `C:\ProgramData\CorsairUtil\agt.dll` , which is a 398 KB PE32+ DLL. The directory name CorsairUtil is deliberate camouflage, chosen to resemble legitimate Corsair peripheral software. The dropped loader then registers a Windows service with a carefully crafted identity designed to survive casual service-list inspection. Watching the registry writes, you immediately notice a cluster of writes to `HKLM\SYSTEM\ControlSet001\Services\CadenceOptimizer` :

Press enter or click to view image in full size

Value	Data
Type	0x20 — SERVICE_WIN32_SHARE_PROCESS (runs inside <code>svchost.exe</code>)
Start	2 — SERVICE_AUTO_START (boots with the system)
ImagePath	<code>%SystemRoot%\System32\svchost.exe -k Brn_Agent</code>
ObjectName	LocalSystem
DisplayName	Audit Engine
Description	Provides driver capabilities for scheduling management and configuration optimization across system components.

And under `CadenceOptimizer\Parameters` :

Press enter or click to view image in full size

Value	Data
ServiceDll	<code>C:\ProgramData\CorsairUtil\agt.dll</code>
ServiceMain	AllocatorRegister

Plus a registration under `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost` : `Brn_Agent` → `CadenceOptimizer`.

Get Gianluca Tiepolo's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

The way this works is that Windows resolves `svchost.exe -k Brn_Agent` by looking up the `Brn_Agent` group name in the `Svchost` registry key, which maps to the `CadenceOptimizer` service, which tells `svchost` to load `C:\ProgramData\CorsairUtil\agt.dll` and call its `AllocatorRegister` entry point. The result is `agt.dll` running inside a shared `svchost.exe` process as `LocalSystem` on every boot — maximum privilege, blended into the crowd of legitimate `svchost.exe` instances, with a display name (`Audit Engine`) and description generic enough to not raise immediate suspicion.

This is a significant escalation beyond classic Vidar behavior. The non-persistent Vidar model is well understood by defenders and relatively easy to remediate. This variant leaves behind a `SYSTEM`-level DLL that will reload on reboot. `agt.dll` needs its own analysis pass, as it may be a backdoor or additional loader.

Finally, Vidar queues itself for deletion via `HKLM\SYSTEM\ControlSet001\Control\Session Manager\PendingFileRenameOperations`, pointing at its own path. On next reboot, Windows deletes it automatically. The dropper disappears, while the adversary gains Persistence through the service-executed DLL.

4.3 Exfiltration

The credential collection and staging is visible in Procmon as a burst of file system activity shortly after the main process started. The pattern is exactly what the static API analysis had predicted: SQLite databases being dumped out of browser profile directories and staged into a working directory.

The staging directory

The Vidar sample first created `C:\Users\user\AppData\Local\Temp\[random]\` and wrote 12 SQLite databases into it with random-looking hex names, ranging in size from 41 KB to 5.2 MB. The largest store (5.2 MB) is almost certainly the browser history or web data database.

What was targeted

The behavioral signatures captured the full scope of collection:

- Browser credentials, cookies, and history across Firefox profiles, Chrome, and Edge;
- WinSCP session data from the registry;
- FileZilla server configurations;
- Cryptocurrency wallet artifacts, specifically looking for Electrum, Exodus, Ethereum, and Coinomi wallet directories;
- PDF and Office documents from the Desktop;

The `CryptUnprotectData` API resolved during static analysis is what allows it to decrypt DPAPI-protected browser credential stores, as Chrome and Edge encrypt saved passwords with DPAPI, so you need that call to get plaintext credentials.

Press enter or click to view image in full size

File Read
File Path
C:\Users\user\Desktop\0k42jece.15x.exe
C:\Users\user\AppData\Roaming\Mozilla\Firefox\Profiles\fq92o4p.default-release\prefs.js
C:\Users\user\AppData\Local\Google\Chrome\User Data\Default\Login Data
C:\Users\user\AppData\Local\Microsoft\Edge\User Data\Default\Login Data
C:\Users\user\AppData\Roaming\Mozilla\Firefox\Profiles\fq92o4p.default-release\key4.db
C:\Users\user\AppData\Local\Google\Chrome\User Data\Default\Login Data For Account
C:\Users\user\AppData\Local\Microsoft\Edge\User Data\Default\History
C:\Users\user\AppData\Roaming\Mozilla\Firefox\Profiles\fq92o4p.default-release\cookies.sqlite
C:\Users\user\AppData\Local\Google\Chrome\User Data\Default\Network\Cookies
C:\Users\user\AppData\Local\Microsoft\Edge\User Data\Default\Web Data
C:\Users\user\AppData\Local\Google\Chrome\User Data\Default\History
C:\Users\user\AppData\Roaming\Mozilla\Firefox\Profiles\fq92o4p.default-release\places.sqlite
C:\Users\user\AppData\Local\Google\Chrome\User Data\Default\Web Data
C:\Users\user\AppData\Roaming\Mozilla\Firefox\Profiles\fq92o4p.default-release\cert9.db

Packaging and exfiltration

After staging the SQLite databases, Vidar packaged and exfiltrated the collected data over HTTPS using the WinHTTP stack I'd mapped statically. Wireshark showed TLS traffic to the C2, contents encrypted. The C2 endpoints visible in memory and captured by the sandbox included short paths (`/bdb`, `/c`, `/p`, `/top`, `/k;`, `/s;`, `/S;`) that match the Vidar panel API pattern seen in earlier public analyses of the family.

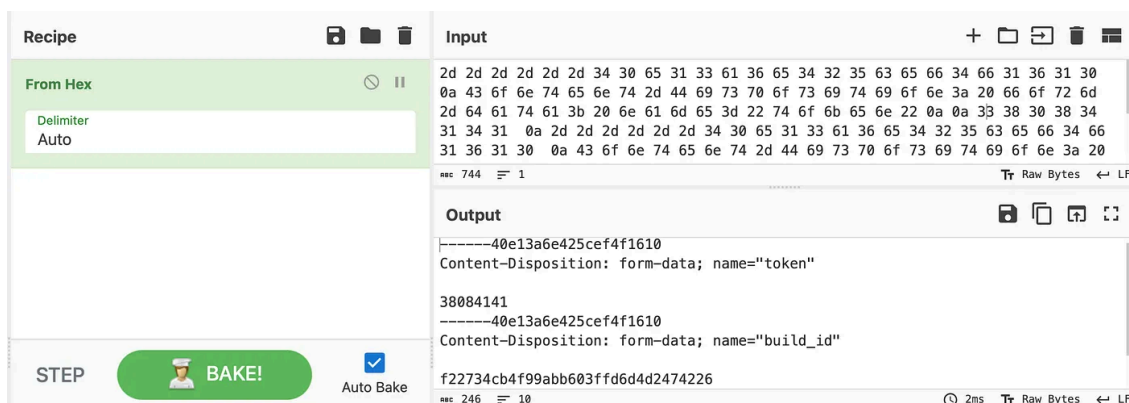
4.4 Command and Control

A few distinct network destinations appeared during the run, each serving a different role. The **primary C2** server which was contacted by my sample was `srv.turbo88ku[.]top`, which resolves to `104.21.31.153`. This is the same hardcoded C2 recovered from the XOR-encrypted config during static analysis, now confirmed active. All resolved IPs hit Cloudflare (ASN 13335) — the `104.21.x.x` range is a Cloudflare anycast block, meaning the IP tells you nothing about the true backend. The malware posts stolen data here and retrieves a JSON configuration response.

A different domain (`netblokirovka[.]asia` → `172.67.164.102`) was used to deliver the secondary payloads (the loader, and `agt.dll`). Both were downloaded during the run. This confirms what the static analysis suggested: this malware is a stealer *and* a downloader. It exfiltrates credentials, then downloads and executes follow-on payloads.

Regarding the C2 communication protocol, the sample first compressed all the stolen data to a zip archive, then it encoded it to Base64, and finally sent it to the C2 server, together with some additional data, such as a “token” and “build_id”:

Press enter or click to view image in full size



Detection note: Because all the malicious hosts resolve to Cloudflare IPs, IP-based blocking is the wrong approach here, as it will both fail to stop the actor and risk collateral damage to legitimate Cloudflare-hosted services. The right blocking primitives in this scenario are **FQDN/SNI-based**, plus a Snort/Suricata rule that tracks the fake User-Agent.

So, to recap, until now we analyzed the sample statically, classifying it as a Go loader that contains an encrypted payload, with Vidar-like functionality. We then confirmed the malware’s stealing behavior through a dynamic analysis. However, this is only one half of CTI work. The other half is taking what you’ve learned about one

sample and using it to reconstruct the broader operation: who else is running this malware, what infrastructure are they using, and how do you set up detection that survives the actor rotating their tools and domains.

5. Clustering and Pivoting

At this point in my research, I had two samples and a handful of durable facts from the RE phase: the loader's custom wbkyc cipher, the payload's custom FNV-1a hash constants, the 256-byte string VM S-box, the XOR config key, and the C2 domain `srv.turbo88ku[.]top`. The question was: given all of that, how far could I extend the cluster, and what was the right way to do it?

With that data in hand, I split the outward-facing work into three phases, moving from the most concrete to the most speculative:

- First, **campaign clustering**: pull together the samples that are demonstrably part of *this* wave of activity — same code, same live infrastructure.
- Second, **family hunting**: turn what's stable across that set into detection rules that will catch *future* samples of the same malware family, even after the actor re-keys the crypter and rotates domains.
- Third, **infrastructure pivoting**: work outward from the known servers and domains to reconstruct the operator's backend, and — ideally — surface infrastructure they've staged but not yet used.

Each phase feeds the next, as the cluster tells you what's stable, the stable features become hunts, and the hunts plus infrastructure pivots reveal where the operation is heading.

5.1 Campaign Clustering

For campaign clustering, I worked in two complementary directions simultaneously: *infrastructure-based clustering* (who else talks to the same C2 and associated domains) and *family-based clustering* (what other samples share the same code and structural fingerprints). Both directions produced results, both produced dead ends, and the dead ends were often as informative as the hits.

Infrastructure-first: Starting from the C2 domain

The obvious first move was to look up what VT already knew about the sample and its infrastructure. The strongest single infrastructure pivot was the C2 domain, so I looked up `srv.turbo88ku[.]top` and immediately hit the first dead end.

As we already discussed, the domain resolved to `104.21.31.153` and `172.67.178.52` — both Cloudflare anycast addresses. Pivoting on those IPs is meaningless, as you'd pull a random slice of the entire internet. Cloudflare fronting is a deliberate infrastructure choice by the actor precisely because it kills IP-based clustering. So I moved on, and started running a few quick VirusTotal queries to understand what samples were talking to that domain:

```
vt domain communicating_files srv.turbo88ku.top -I
vt domain referrer_files srv.turbo88ku.top -I
vt domain downloaded_files srv.turbo88ku.top -I
vt domain memory_pattern_parents srv.turbo88ku.top -I
```

These queries surfaced two sibling samples (labeled wingo, a Go loader), both sharing the `C:\Windows\<random>.exe` drop pattern. I then checked what other network connections these new samples were opening and realized that these samples also contacted Telegram (the dead-drop) and Cloudflare (the C2), which is consistent with the sample we already analyzed, so I took note of their SHA256, and moved on.

The more interesting clustering pivot came from the dead-drop domains recovered during static analysis. I pivoted on `glamisrent[.]com` in VT and found 7 communicating files, none of which were in my current cluster. All were archives carrying cracked-software and game-cheat lures:

Press enter or click to view image in full size

Archive name	Detection label
<code>build_2.0.6.zip</code> , <code>release_v2.8.1.zip</code> (~3.3 MB)	trojan.hrgn (Go)
<code>ToolsUnlockPro.rar</code> (48 MB)	wingo (Go)
<code>Skeet-CS2.zip</code> (CS2 cheat, 76 MB)	abmrisk
<code>Flac (extract.me).zip</code> (44 MB)	loader/gozi

A single shared backup C2 domain exposed the distribution vector: fake cracked software and game cheats, packaged in zip/rar archives on download sites.

Family-first: Clustering on structure

At this point, I decided to focus more on our sample's structure, rather than the infrastructure, to find similar samples that could be related to our campaign. I pulled the loader's structural fingerprints and ran a few family-oriented searches, looking for similar files based on their **vhash**, **imphash**, **ssdeep** and **tlsh**:

```
vt file <loader_sha> -- include=vhash,imphash,ssdeep,tlsh
vt search 'vhash:036086656d55551d15545az2e!z fs:2026-06-11+'
```

The first dead-end I ran into was that clustering on **imphash** returned thousands of files, which was completely useless. The reason is obvious in retrospect: a Go binary imports only a handful of DLLs from the Go runtime, so thousands of completely unrelated Go binaries share the same imphash.

The right structural pivot to get our first cluster turned out to be VirusTotal's **vhash**, which is a structural similarity hash based on section layout, PE characteristics, and related structural properties. For our Go loader, this resulted in **83 files** discovered over the last 24h (use vt's search modifier, `fs:2026-06-11+`, to only get samples first seen since a date), all ~3.3 MB Go PEs labeled `wingo` or `loader`.

Until now I focused only on the loader, but I was also curious to see if there were any matches for the second-stage payload, so I ran a raw hex-content search using the 256-byte S-box recovered in Ghidra as a content fingerprint:

```
vt search content '{cc ea 32 c5 f3 62 05 8e 12 0b e3 ca 44 4c f9 bf}'
```

This provided 18 more results, in various size bands. Crucially, one hit carried explicit **Vidar** AV labels:

Win64/Vidar.D , Trojan.PWS.Vidar.497 . Combined with the Telegram/Steam dead-drop mechanism documented during the previous analysis, this confirmed the family attribution from the payload side, independently of behavioral sandbox labels.

The two searches complement each other: vhash catches the packed loader (as the second-stage payload is encrypted, invisible), while the S-box content search catches the exposed payload. Together they cover both forms.

At this point I had a cluster of different sample, but it's important to keep in mind that vhash clusters the loader, not the payload. The hardest question now is, does "same loader" mean "same family"? In fact, a crypter can wrap Vidar for one actor and a different stealer for another, so I set out to validate this empirically by downloading the samples and attempting to unpack the payload.

Here i reached another dead end, as unpacking samples at scale proved impossible. I did manage to build a generalized Python unpacker that locates the encrypted payload by entropy and brute-forces the MZ/PE alignment, but two instructive sub-problems appeared: a 256-byte alignment ambiguity (the `wbkyc` cipher reverses the buffer, making a naive MZ check lock onto corrupted-but-MZ-headed alignments), and an overlay rabbit hole (trailing bytes that didn't affect the config but burned time to diagnose). The unpacker validated on the seed sample, but when I launched it on our file cluster, it failed on all of them. The reason is that the crypter in our Go loader re-keys `wbkyc` per build, so none of the twin samples decrypted with the seed's key `0x1062c` . The `wbkyc` decode key is a per-build or per-campaign value, so static unpacking with a fixed key does not scale.

With static unpacking failing, I turned to **behavioral validation** and pulled each file in the cluster's network behavior from VT:

```
vt file contacted_domains <hash>
vt file memory_pattern_domains <hash>
```

I then did some quick regex magic to aggregate the results based on which domain the samples were contacting, and the result flipped entirely. Of 56 samples with behavioral data from our cluster, 55 contacted the same dead drops (telegram and steam) and domains that were consistent with the naming of our known C2 server.

Press enter or click to view image in full size

Of 84 samples in the cluster	Count
Confirmed family: telegram/steam dead-drop and <i>glamis/turbo</i> pattern	55
Confirmed <i>different</i> family	1
No behaviour data	28

The verdict from the behavioural data is the single most important finding of the clustering phase: this Go loader/crypter is bespoke to this Vidar operation. That matters because a loader is generic by nature — the same packer can wrap Vidar for one actor and an entirely different stealer for another, which is exactly why “same loader” doesn’t automatically mean “same family”. Here, though, 55 of the 56 samples with behavioural data resolved to the same dead-drop pair and the same glamis/turbo C2 naming. A loader used by only one operation, across multiple concurrent campaigns, is effectively a family signature in its own right.

With 55 confirmed members all detonating inside a 24-hour window, I had a starting cluster, and it let me separate what’s *family-stable* from what’s *campaign-specific*. The PE structure (vhash) was constant across every sample, while only the outermost C2 details varied:

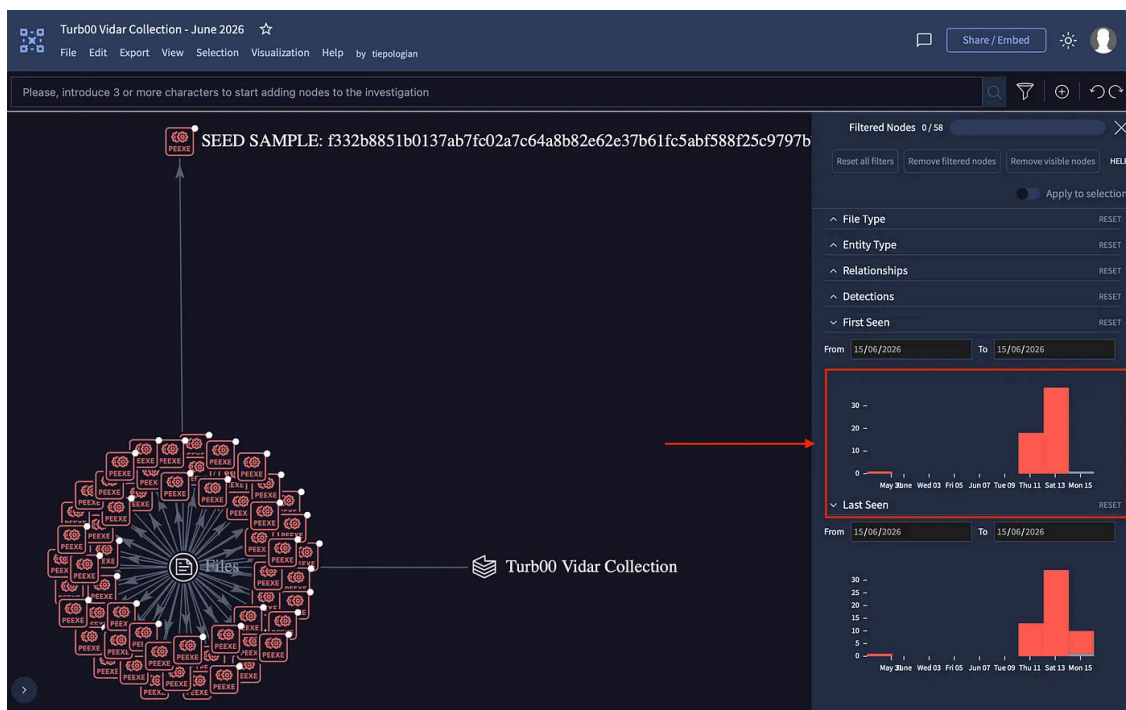
- **C2 domains:** `srv.turbo88ku[.]top` , `ggt.glamisrents[.]com` , `srv.glamisdunesrentals[.]com`
- **Telegram dead-drop handles:** `telegram[.]me/d77xtr` and `telegram[.]me/turb00m`
- **Secondary payload servers** `netblokirovka[.]asia`

The naming convention gives the operator away: glamisrent, glamisrents, glamisdunesrentals are all variations on “Glamis,” alongside the “Turbo” theme — one operator reusing a vocabulary across dozens of domains. The practical takeaway for detection is the split itself: the C2 *host* is disposable and rotates constantly, but the Telegram + Steam dead-drop pattern and the glamis/turbo naming theme are durable, and that’s where the detection value lives.

5.2 Family Hunting

To recap, up to this point we have a confirmed cluster of 55 samples that share identical PE structure (same vhash) and contact C2 servers with the same naming convention. The [collection](#) I created in VirusTotal Graph allows us to visually see the campaign, with most of the samples first seen between the 11th-13th June, except for one sample that dates back to the 30th May.

Press enter or click to view image in full size



With this cluster confirmed, the next goal was to build hunting rules that would allow me to catch *new* samples from this family across *future* campaigns, including samples that wouldn't share the current C2 domain structure.

Because Vidar's fingerprints live inside the *encrypted* payload and the loader is full of noise, no single rule can do the job. I layered three complementary rules:

- Rule 1 ([vidar turb00 payload](#)): The first rule is a high precision, limited reach rule that is anchored on the custom FNV-1a constants (`0x3aff8d85` basis, `0x010006bb` prime), the string VM initialization constant (`0x311b7f33`), the 16-byte XOR config key, and a cluster of resolved API hash dwords. It fires on exposed payloads, unpacked samples, memory dumps, or non-encrypting variants. It obviously does not fire on the packed loader.
- Rule 2 ([vidar turb00 go loader](#)): The second rule is static, has a broad reach and targets the delivered loader before it executes. Combines PE32+ AMD64 architecture, Go build ID string, characteristic size band (~3–3.5 MB), a high-entropy embedded region in `.data` , and the one surviving family-distinctive code sequence. I validated it against all the files in our cluster. This rule works, but will definitely bring in false positives.
- Rule 3 ([vidar turb00 loader behaviour](#)): This rule builds on the previous one, and is meant to be used as a *VT Livehunt* rule only. It fires only after VT detonates the sample, giving access to behavioral data. The byte anchor is deliberately dropped and replaced by behavioral confirmation, which is a regex on the actor's C2 naming theme (`/glamis[a-z]*rent/`) and the presence of both the Telegram and Steam dead-drop pair in the sample's network activity. The point here is that behavior is durable where bytes are not, so this rule will keep firing even after the actor rotates samples, as long as they maintain the same C2 operational patterns.

The three rules cover each other's gaps: precision on the exposed payload (Rule 1), reach on the loader pre-detonation (Rule 2), durability across future campaigns post-detonation (Rule 3). Together, these rules alert me when new samples emerge, allowing me to track the malware family cluster, even across different campaigns.

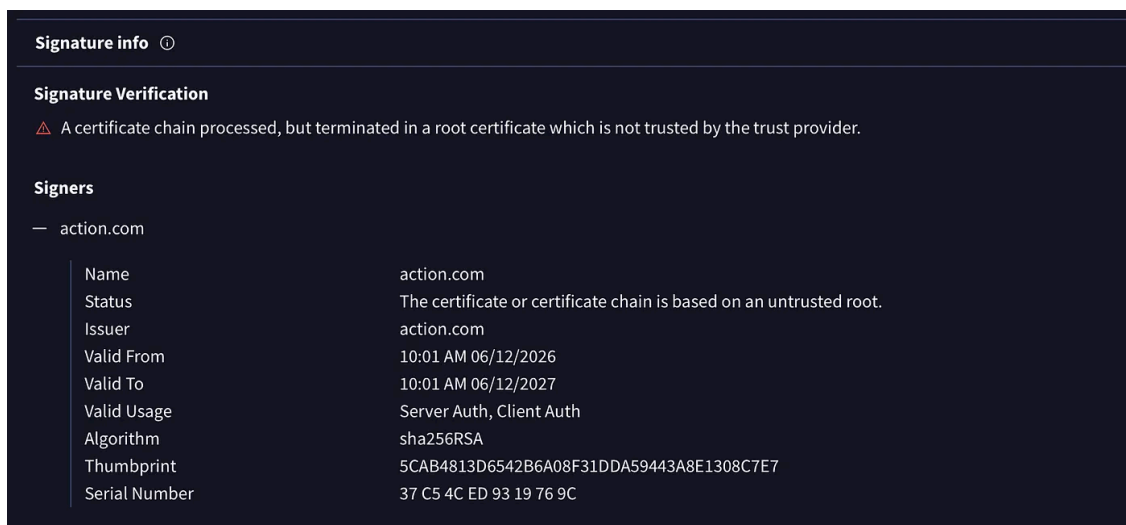
5.3 Adversary Infrastructure Tracking

While the clustering work had given me a map of the malware itself, what I really wanted was to pivot on the infrastructure and push past the Cloudflare front to find the actor's real servers and, possibly, find new C2 domains even before the actor started using them.

Pivot A — The PE code-signing certificate

One of the most useful pivoting techniques is to analyze the PE's code-signing certificate. Back to our original sample, I noticed that the loader carried an invalid signature, based on a certificate with serial number `37C54CED9319769C`, issued by `action[.]com`.

Press enter or click to view image in full size



I initially queried VirusTotal for any other samples that shared the exact same certificate, which returned no new matches. However, searching VT for all files signed under `action[.]com` returned 4 samples including one new one, which wasn't present in our cluster. That sample was definitely the same family — a 3.37 MB Go loader with a one-character **vh**ash variant — and it contacted the usual `telegram[.]me` dead-drop, `srv.glamisdunesrentals[.]com`, and a non-Cloudflare backend `65.21.96[.]134` directly.

Press enter or click to view image in full size

```

tiepolo — /dev/tty001
tiepolo@MacBook-Air ~ % vt search 'tag:signed signature:"5CAB4813D6542B6A08F31DDA59443A8E1308C7E7"' --identifiers-only
- "f332b8851b0137ab7fc02a7c64a8b82e62e37b61fc5abf588f25c9797b7ab005"
tiepolo@MacBook-Air ~ % vt search 'tag:signed signature:"37 C5 4C ED 93 19 76 9C"' --identifiers-only
- "f332b8851b0137ab7fc02a7c64a8b82e62e37b61fc5abf588f25c9797b7ab005"
tiepolo@MacBook-Air ~ % vt search 'tag:signed signature:"action.com"' --identifiers-only
- "7bd5a491f52e28fb1ce93b6ffcd527b1f8dd47d95ab545e12c2692da698a874d"
- "e2a7d5be1ad88804dc71518b830bd8eac819a69e9417d0964bf14e82a47699e5"
- "cc966f6b70c632092efdbb6f05f288fb155d976c1bb7fca7495dd1349ff6a9f1"
- "f332b8851b0137ab7fc02a7c64a8b82e62e37b61fc5abf588f25c9797b7ab005"
tiepolo@MacBook-Air ~ %

```

The signing certificate was another family clustering anchor, and it handed me a sample that directly confirmed the backend IP. From this pivot alone you can understand how easy it is to end up in a rabbit-hole, as there are so many new pivoting anchors that emerge, such as hunting for the new **vhash** variant that just emerged.

Pivot B — The DNS resolver banner

The `65.21.96[.]134` host was, so far, the only backend IP both confirmed malicious (same family communicating files) and not Cloudflare-fronted, so I decided to check if this machine was exposing anything. A quick Shodan query showed that this machine had port 53 open — a BIND DNS server — and the banner leaked a resolver hostname: `server.taaktook.website`.

Pivoting on this fresh indicator, Shodan returned a 4-server cluster that had the same resolver hostname:

Press enter or click to view image in full size

IP	Hosting	Role
65.21.96[.]134	Hetzner (FI)	DNS/mail — confirmed C2 backend, same malware family communicating to this host
65.21.80[.]152	Hetzner (FI)	DirectAdmin panel
54.38.174[.]165	OVH (FR)	Management server — <code>server.taaktook.website</code> itself
54.38.231[.]234	OVH (FR)	DNS/mail/web

Two independent fingerprints tied them together: the `taaktook.website` DNS resolver banner and DirectAdmin self-signed certificates. These were the actor's backend hosting and management servers, the real infrastructure some of the Cloudflare-fronted C2 domains sit on.

I immediately checked whether the OVH boxes were themselves malicious C2 nodes. Both had 0 VT malicious/suspicious ratings and 0 communicating files, hosting unrelated shared-hosting tenants. They tie to the actor only through the shared DNS resolver banner. The OVH org field is assigned to "jalil abdul", and a Shodan pivot on this indicator pulled back a fleet of ~15+ servers — most likely an abuse-tolerant hosting reseller, with our actor as one tenant. Useful as a monitoring surface for the hoster, but overall a low-confidence indicator.

Pivot C — The self-signed TLS certificate fleet

Another useful infra-related pivot point is a domain's TLS certificate. On the confirmed C2 backend `65.21.96[.]134`, VT's HTTPS certificate panel showed a self-signed certificate — subject CN and issuer CN both `135.181.77[.]214`, which means it was minted on that host:

```
Serial: b7d2d897adfb46cdebfa997d03bb4841
Thumbprint: ae87b4dab40e1d42fe3558bab959204a915be547
CN: 135.181.77[.]214 (self-signed)
Validity: 2026-06-01 → 2036-05-29 (10-year cert)
JARM: 40d40d40d0000000043d40d40d43d684d61a135bd962c8dd9c541ddbaefa8
```

A 10-year self-signed certificate is immediately suspicious, as it means someone manually ran `openssl req -x509 -days 3650` by hand. Pivoting on the cert serial in Shodan returned 4 machines, plus the cert's origin machine. All hosts are on the same Hetzner Finland ASN, AS24940, and share an identical JARM. After a bit more manual analysis:

Press enter or click to view image in full size

IP	Cert-linked	Malicious evidence	Our Vidar campaign
65.21.96[.]134	Yes	Yes, Vidar-family matches	Confirmed
65.21.96[.]131	Yes	Yes, Vidar-family matches	Likely, not confirmed
135.181.224[.]79	Yes	Yes, trojan loaders	Possible, not confirmed
65.21.96[.]135	Yes	Yes, trojan loaders	Possible, not confirmed
135.181.77[.]214	Yes (origin)	Yes, crypto-miner/torrent	Different activity

What made this TLS certificate particularly interesting is that the identical private key appeared on heterogeneous operating systems — `65.21.96[.]131` running *Windows/IIS 10.0*, `135.181.224[.]79` running *Debian/OpenSSH 9.2p1*. A default certificate generator produces a fresh key per host and cannot produce the same serial on both Windows and Linux. Only a human operator copying a cert+key file across their fleet can.

In practice, this was an **operator artifact**, a hand-rolled certificate deliberately replicated across the C2 backend fleet. The cert serial appears on only a small number of hosts, near-unique, and a strong tracking anchor. The JARM, on the other side, was totally useless; in fact, JARM is an *active* TLS fingerprint: it fires a series of crafted TLS Client Hellos at a server and hashes how the server responds, so it characterizes the TLS stack and its configuration — not the operator. The value we saw (`40d40d40d` ...) is one of the most common JARMs on the internet: a stock configuration shared by 185k hosts on Shodan at the time of analysis. Pivoting on it would return a vast, undifferentiated slice of the internet, the same dead end as pivoting on a Cloudflare IP. The certificate *serial*, by contrast, encodes a private key the operator generated once and copied by hand; it appeared on only a handful of hosts. That's the real lesson of this pivot: a fingerprint is only useful if it's rare. JARM measured something genuine, but far too common to attribute.

Pivot D — Finding new attacker infrastructure

Up to this point every pivot had been *reactive*, as I was following indicators a sample had already touched. But the highest-value infrastructure intelligence is *predictive*, such as finding the actor's domains and servers before they're used against a victim. So for this final pivot the goal shifted: not to catalogue what the actor had already used, but to catch what they were *about to*.

The idea is to turn one of the actor's operational habits into a search. Across the confirmed infrastructure, I'd noticed the operator consistently pre-creates two specific subdomains — `ggt.` and `puz.` — on each new C2 root, often well before the domain goes live. That habit is distinctive enough to hunt on directly.

The VT query below asks for any domain modified recently (a tight recency window keeps it focused on freshly-staged infrastructure) whose name matches exactly `ggt.*.top` or `puz.*.top`.

```
entity:domain last_modification_date:2026-06-03+ (  
  domain_regex:"^ggt\.[^.]+\top$" OR domain_regex:"^puz\.[^.]+\top$"  
)
```

Broadening the TLD coverage and then writing a quick Python script requiring the intersection of roots with *both* a `ggt.` and a `puz.` subdomain, cleaned the results perfectly, showing only domains where the actor has pre-staged both subdomains. This query can become a durable infrastructure monitoring anchor for this actor. Run as a saved VT Intelligence search on a short recency window, any new root domain appearing with both a `ggt.` and a `puz.` subdomain is a candidate for new actor infrastructure.

To confirm this isn't purely theoretical: as an example, this monitoring query surfaced `goturbo88[.]top` — a pre-staged candidate with the `ggt.` subdomain and the turbo naming theme, but with zero communicating files at discovery time (12th June). The day after discovery, the `ggt.goturbo88[.]top` domain started showing malicious communicating files!

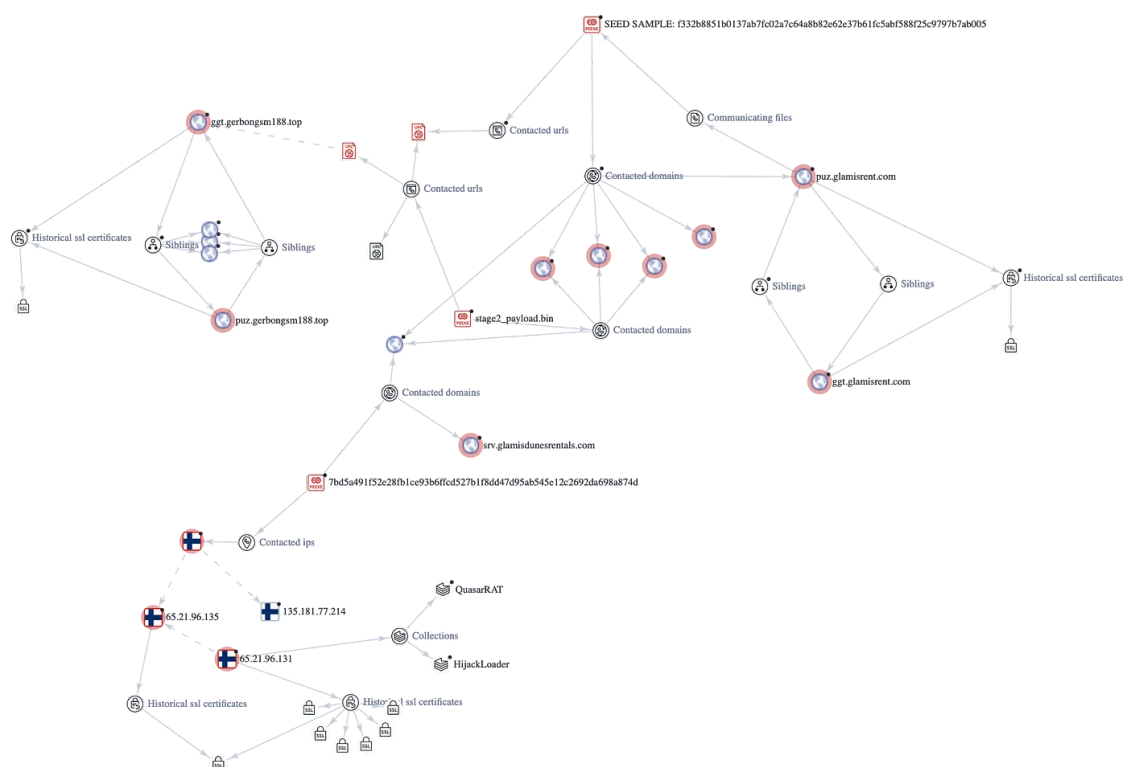
Press enter or click to view image in full size

Communicating Files (7) 🔍			
Scanned	Detections	Type	Name
2026-06-13	14 / 67	ZIP	Bootstrapper_v2.7.5.zip
2026-06-14	19 / 71	Win32 EXE	IDM_6.4x_Crack_v20.6.exe
2026-06-13	16 / 67	ZIP	Bootstrapper_v1.0.6.zip
2026-06-14	23 / 70	Win32 EXE	7747af7cc821341d62e88d21c71cf518bc2e9dd890328c89e5d16357eace1a14.exe
2026-06-13	16 / 67	ZIP	Bootstrapper_v3.5.6.zip
2026-06-14	20 / 66	Win32 EXE	fc70g.exe
2026-06-14	36 / 69	Win32 EXE	31x3xqr.exe

Pivoting further and harvesting communicating_files from live ggt./puz. subdomains, surfaced 33 additional samples beyond the original cluster, and pulling their contacted_domains produced a new confirmed C2 (`aml[sentry[.]online`), which is Cloudflare-fronted, has malicious communicating files with at least one in our validated cluster.

So, we started off with just one sample and a domain it used as a C2, and through analysis, clustering and pivoting, we now have a [larger picture](#) of the campaign and its infrastructure, with dozens of samples and domains that can each become further pivot points:

Press enter or click to view image in full size



6. Summary: Closing The CTI Loop

A long weekend, one anonymous binary, and a friend's ping on a Friday morning. That's where this started! What it turned out to be — a multi-stage Vidar operation — is what happens when you follow the thread all the way through instead of stopping at the first answer. I want to use this final section to make explicit something that ran

underneath every step of this research, because it's the thing that makes CTI work actually useful rather than just intellectually satisfying: the feedback loop.

Analysis produced tooling: Reversing the sample's cipher through static analysis didn't just tell me what the loader was doing, it also produced a reusable Python decoder that could extract the Vidar payload from any loader built with the same cipher. Tracing the config decryption produced a Python script which could recover C2, User-Agents, campaign ID, and the full capability map from any sample of this family without hardcoding offsets. These tools unlock future research and pivoting.

Clustering produced detection rules: The initial 55-sample validated cluster gave me something I couldn't have had from one sample: the ability to distinguish family-stable features from campaign-specific ones. The opaque-predicate constant `0x2b5626` appeared only in the turbo88ku-campaign builds, not the *glamisrents* ones — so it was a campaign IOC, not a family anchor. The S-box, the FNV-1a constants, the dead-drop behavioral pattern — those appeared across all 55. Only by having the cluster could I tell the difference, and this is what helps us create effective detection rules.

Detection rules created a hunting surface: The YARA rules were deployed in client's environments and used as Livehunt rules on VT. New samples arriving that match the family patterns trigger alerts automatically, feeding new samples back into the analysis pipeline. The loop closes: analysis → detection → new samples → analysis.

Infrastructure tracking enables proactive detection: The ggt./puz. subdomain monitoring query catches new actor domains while they're **pre-staged** — before any sample contacts them. In practice, this means defenders can have blocklist entries for `goturbo88[.]top` in place *before the actor starts using it*, and can flag `amlstory[.]online` as C2 infrastructure *before* a victim's endpoint reaches out to it. The actor has to build infrastructure before they can use it; a monitoring query that catches them during construction reverses the asymmetry.

From just one sample, this is what the weekend produced: the loader fully reverse-engineered, the payload extracted, decrypted, and attributed to Vidar, 101 APIs mapped, 30 internal strings decoded, the config decrypted, three reusable Python tools, a 55-sample validated cluster across multiple concurrent campaigns, the distribution vector, an infrastructure map, a three-rule YARA detection pack, network and host rules, and a durable monitoring query that catches new actor infrastructure while it's being staged.

This is the CTI loop working as intended! The research doesn't end when the report is written — it seeds the next detection, and the next detection seeds the next research question. The actor will re-key the crypter, rotate the C2 domains, and rebuild the infrastructure. When they do, the behavioral hunts will catch the new samples, the monitoring query will catch the new domains, and the cert serial will catch the new servers. The specific IOCs expire. The methodology doesn't.

Detection Opportunities

All detection rules are available in [this repository](#) on my GitHub.

File / YARA:

- [vidar turb00 payload](#): Custom FNV-1a constants, VM init constant, S-box, XOR key, API-hash cluster. Fires on exposed/unpacked payloads and memory dumps.
- [vidar turb00 go loader](#): Broad-reach static rule on the delivered loader (PE32+/AMD64, Go build-id, ~3–3.5 MB, high-entropy .data region, one surviving family code sequence). Expect some FPs.
- [vidar turb00 loader behaviour](#): VT Livehunt only; behaviour-gated on the C2 theme + Telegram-and-Steam dead-drop pair. Survives sample rotation.

🔗Network (Suricata/Snort):

- [Snort rule](#) for exact-match on the impossible User-Agent (Firefox/156.0) — durable across domain rotation.
- FQDN/SNI blocklisting on the family domains (never IP-block, as all C2 is Cloudflare-fronted).
- Community signature “[ET MALWARE Vidar Stealer Form Exfil](#)” fires on the live exfil POST pattern.

🖥️Host / Behavioral (Sigma):

- [Sigma rule](#) that tracks Vidar service creation for Persistence

🏗️Infrastructure Monitoring:

- New domains carrying both a ggt. and a puz. subdomain.
- Shodan monitor on cert serial `b7d2d897adfb46cdebfa997d03bb4841` for new backend nodes.

IOCs

A full list of indicators is available in [this repository](#) on my GitHub.

OPERATION TURB00 – KEY INDICATORS OF COMPROMISE

— SAMPLES (SHA-256) —

Seed Go loader

`f332b8851b0137ab7fc02a7c64a8b82e62e37b61fc5abf588f25c9797b7ab005`

Extracted Vidar payload

`725344564a8c9b0a033e9a5d41369fae5b8503d36e87002e7c064eface927e7b`

Cert-pivot sibling loader

`7bd5a491f52e28fb1ce93b6ffcd527b1f8dd47d95ab545e12c2692da698a874d`

— NETWORK – C2 & DEAD-DROPS —

Primary C2 (hardcoded) `srv.turbo88ku[.]top`

Family C2 domains `ggt.glamisrents[.]com`
 `srv.glamisdunesrentals[.]com`
 `glamisrent[.]com`
 `ggt.gerbongsm188[.]top`

	amlentry[.]online
	goturbo88[.]top
Telegram dead-drops	telegram[.]me/d77xtr
	telegram[.]me/turb00m
Steam dead-drop	steamcommunity[.]com/profiles/76561198694566254
Dead-drop marker string	gr0u4

— NETWORK – SECONDARY PAYLOAD DELIVERY —

netblokirovka[.]asia/files/install.exe
netblokirovka[.]asia/files/telemetriawork/telepuz.dll
scaleyou.com[.]br/3.exe
scaleyou.com[.]br/144.exe

— BACKEND INFRASTRUCTURE —

Confirmed C2 backend IP	65.21.96[.]134
Code-signing cert	serial 37C54CED9319769C, issuer action[.]com
Operator TLS cert serial	b7d2d897adfb46cdebfa997d03bb4841

Source: <https://mrtiepolo.medium.com/operation-turb00-analyzing-and-hunting-a-vidar-campaign-b45481d8cd74>