

Operation Turb00 — Part 3: Unmasking the SnappyClient RAT

 mrtiepolo.medium.com/operation-turb00-part-3-unmasking-the-snappyclient-rat-f9336de9400a

Gianluca Tiepolo

July 2, 2026

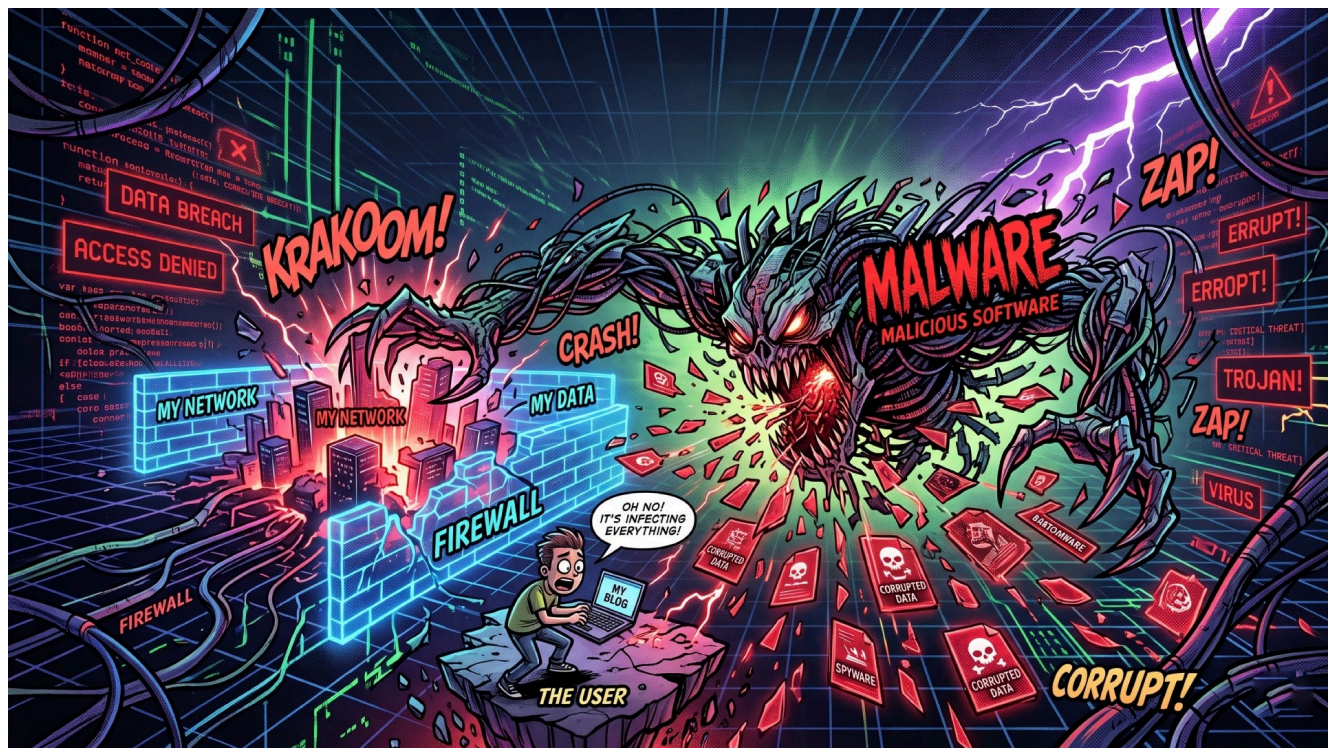


Table of Contents

1. Executive Summary
2. Initial Triage
3. Analyzing the Trojanized Opera DLL
4. Static Analysis of SnappyClient RAT
5. Dynamic Analysis
6. Dissecting SnappyClient's C2 Protocol
7. Closing Thoughts

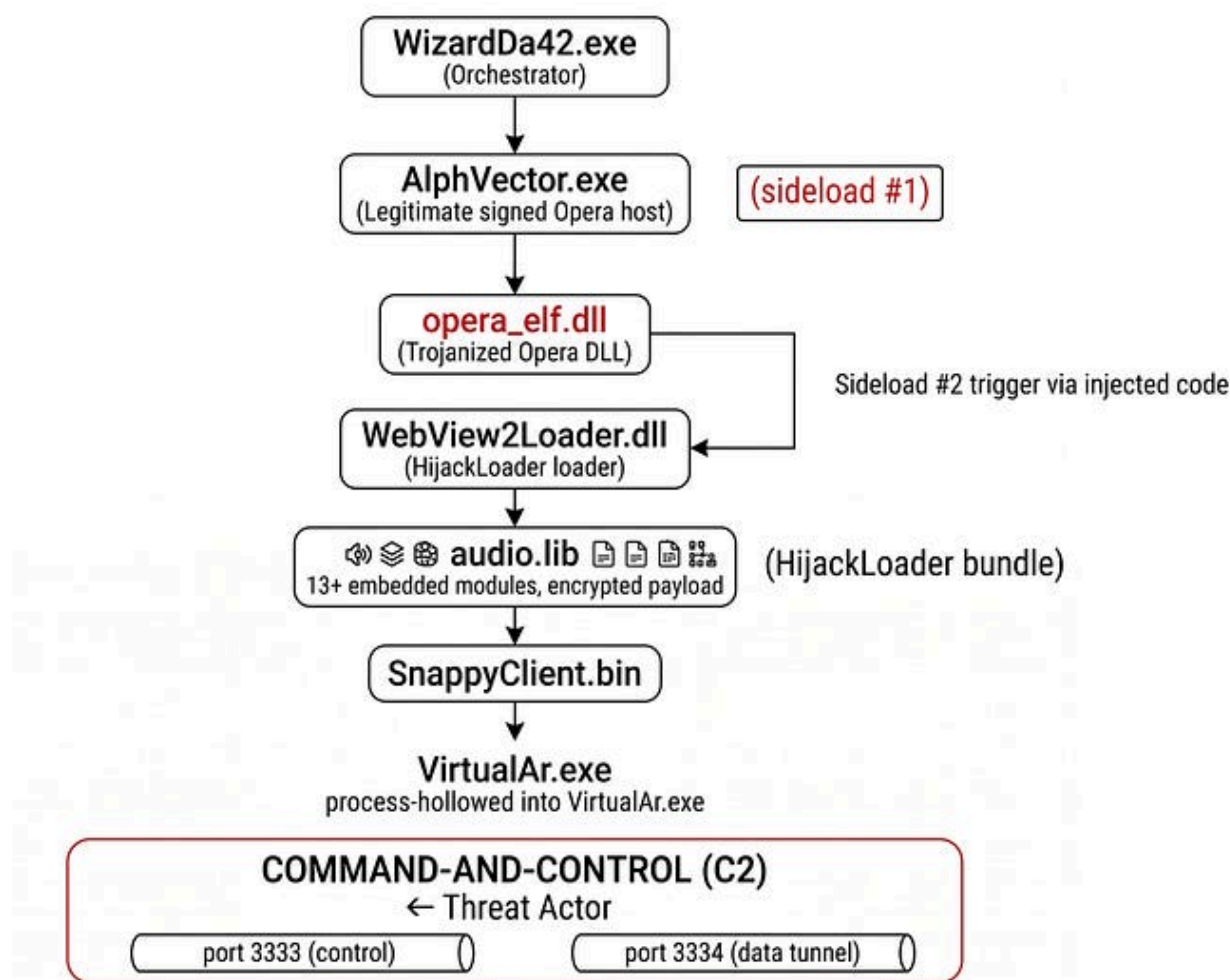
Executive Summary

The first two posts in this series examined the first stage of this campaign: a HijackLoader chain that stages and decrypts Vidar 2.1, then process-hollows it into a clean, signed binary to carry out cookie and credential theft. This post covers the campaign's second infection chain, which delivers a different payload entirely: **SnappyClient**.

The orchestrator behind the first chain, WizardDa42.exe (the legitimate Radmin VPN process), also launches this second one. Beyond that shared parent process, the two have nothing in common, as the second chain uses its own sideloading host, its own HijackLoader component, its own resource bundle, and its own final implant.

End to end, the second chain looks like this:

Press enter or click to view image in full size



SnappyClient is a full-featured, commercial-grade Remote Access Trojan (RAT). Where Vidar steals data once and exits, SnappyClient is built to be persistent and give the operator interactive control of the host. Its capabilities include remote desktop (VNC, and a hidden-desktop HVNC variant), an FTP server, a SOCKS5 reverse proxy, a remote command shell, a keylogger, and theft of browser credentials, cookies, and cryptocurrencies. Its command-and-control runs over a custom encrypted TCP protocol rather than HTTPS, and the commands are concealed behind several layers of compression and encryption that must be dissected before any data is visible.

The following sections walk through how I triaged this sample, what its capabilities are and what this infection chain tells us about the current threat landscape.

Initial Triage

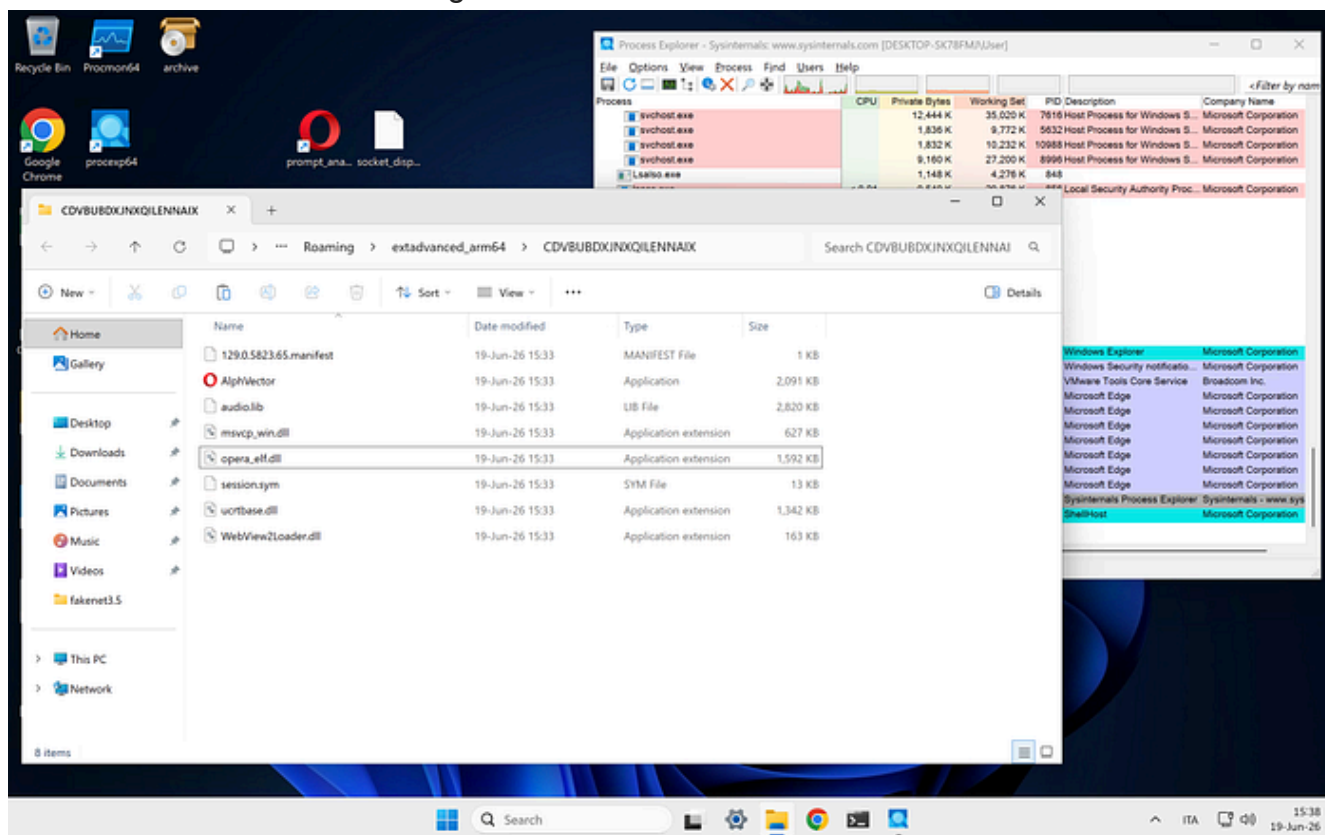
I started this analysis exactly where Part 2 left off. The Vidar infection chain ends when WizardDa42.exe, the legitimate Radmin VPN process, hollows Vidar 2.1 into NeuroManag.exe. The process tree captured during dynamic analysis had already shown me that the same parent spawns a second process chain in parallel, starting with AlphVector.exe.

WizardDa42.exe (orchestrator)

- └─ NeuroManag.exe ← Vidar 2.1
- └─ AlphVector.exe ← entry point for the second chain
 - └─ VirtualAr.exe ← payload host (SnappyClient runs here)
 - └─ chrome.exe
 - └─ Crisp.exe ← signed sideload host, persistence

For this second infection chain, WizardDa42.exe drops an entire toolkit into an AlphVector/ folder, and that folder is the starting point for this analysis:

Press enter or click to view image in full size



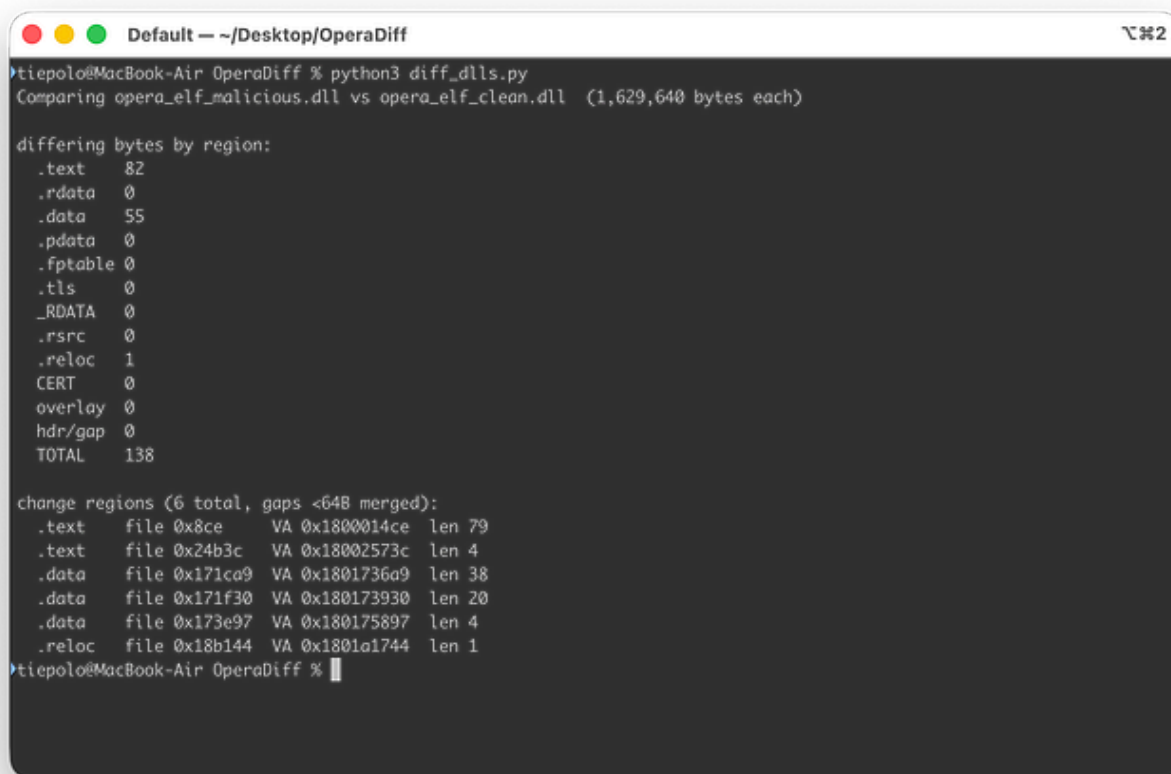
AlphVector.exe is the legitimate Opera binary, so I didn't analyze it because its only job is to do what a DLL search-order sideload always does, which is load a same-named DLL from its own directory without checking whether that DLL is the one Opera shipped with. The actual malicious code, including the SnappyClient PE itself, the C2 endpoints, the dropped-file names, and the campaign tag, is buried inside one of the other files.

A DLL-sideload chain has a deterministic execution order: the host file loads, it imports a sideloadable DLL, and that DLL's load triggers whatever comes next. That made opera_elf.dll the obvious first artifact to analyze.

Analyzing the Trojanized Opera DLL

I had a clean copy of opera_elf.dll to compare against, so rather than open the malicious file cold in a disassembler, I first ran a raw byte-level diff against the clean version:

Press enter or click to view image in full size

A terminal window titled "Default — ~/Desktop/OperaDiff" with a dark background. The prompt is "tiépolo@MacBook-Air OperaDiff %". The command "python3 diff_dlls.py" has been executed. The output shows a comparison of "opera_elf_malicious.dll" and "opera_elf_clean.dll", both 1,629,640 bytes. It lists differing bytes by region: .text (82), .rdata (0), .data (55), .pdata (0), .fptrtable (0), .tls (0), _RDATA (0), .rsrc (0), .reloc (1), CERT (0), overlay (0), hdr/gap (0), and TOTAL (138). Below this, it lists change regions (6 total, gaps <64B merged): .text (file 0x8ce, VA 0x1800014ce, len 79), .text (file 0x24b3c, VA 0x18002573c, len 4), .data (file 0x171ca9, VA 0x1801736a9, len 38), .data (file 0x171f30, VA 0x180173930, len 20), .data (file 0x173e97, VA 0x180175897, len 4), and .reloc (file 0x18b144, VA 0x1801a1744, len 1). The prompt is now "tiépolo@MacBook-Air OperaDiff %".

```
tiépolo@MacBook-Air OperaDiff % python3 diff_dlls.py
Comparing opera_elf_malicious.dll vs opera_elf_clean.dll (1,629,640 bytes each)

differing bytes by region:
.text      82
.rdata     0
.data      55
.pdata     0
.fptrtable 0
.tls       0
_RDATA     0
.rsrc      0
.reloc      1
CERT       0
overlay    0
hdr/gap    0
TOTAL     138

change regions (6 total, gaps <64B merged):
.text  file 0x8ce  VA 0x1800014ce len 79
.text  file 0x24b3c VA 0x18002573c len 4
.data  file 0x171ca9 VA 0x1801736a9 len 38
.data  file 0x171f30 VA 0x180173930 len 20
.data  file 0x173e97 VA 0x180175897 len 4
.reloc  file 0x18b144 VA 0x1801a1744 len 1
tiépolo@MacBook-Air OperaDiff %
```

Both files are exactly the same size, which already told me this was a binary patch rather than a complete replacement. The diff narrowed the entire attack surface down to six change regions spread across .text, .data, and .reloc, totaling roughly 144 bytes.

Having exact virtual addresses and lengths before opening Ghidra meant every subsequent query could be targeted at a specific location instead of scrolling through 1.6 MB of disassembly looking for something suspicious. So with both DLLs loaded side by side, I decided to analyze the raw bytes at each diff address to understand what was changed.

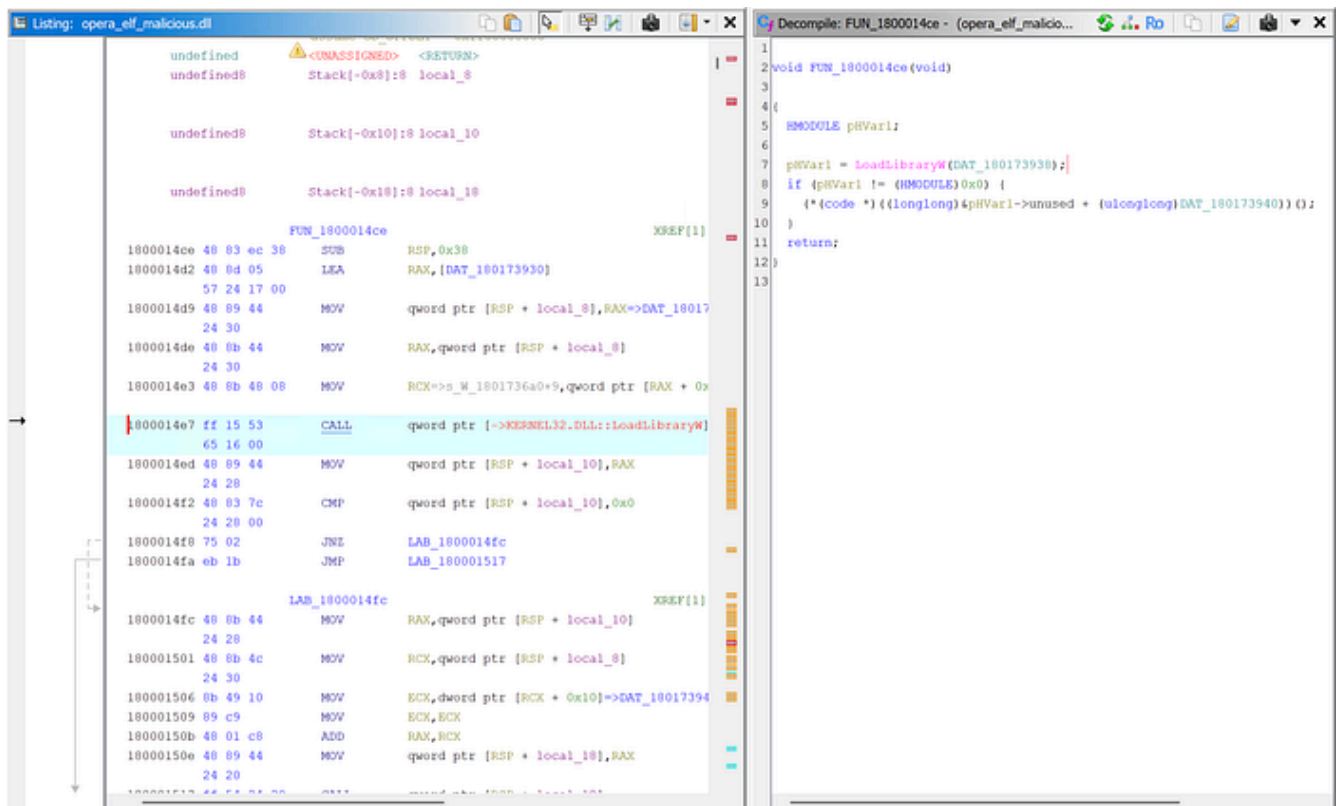
The 79-byte change at 0x1800014CE was the one that mattered most, as it showed me that the threat actor had written a complete, self-contained function into that region:

```
SUB  RSP, 0x38
LEA  RAX, [config]          ; load runtime config address
MOV  RCX, [RAX+0x8]         ; RCX = ptr to L"WebView2Loader.dll"
CALL [LoadLibraryW]        ; load second-stage DLL
CMP  RAX, 0
JZ   .done                 ; exit silently on failure
MOV  ECX, [config+0x10]     ; ECX = 0x2A40 (hardcoded RVA)
ADD  RAX, RCX              ; hModule + 0x2A40
CALL RAX                   ; execute payload
.done:
ADD  RSP, 0x38
RET
```

The second .text change, just 4 bytes at 0x18002573C, sat inside a byte sequence where the clean DLL called an ordinary function elsewhere in the binary, while the malicious DLL called 0x1800014CE — the exact address of the stub I'd just found. The mechanism was now clear: a self-contained loader stub had been written into the DLL's .text section.

I disassembled FUN_1800014CE in full, and its logic is short and deliberate. It reads a small configuration structure from .data, takes from it a pointer to the wide string WebView2Loader.dll, and passes that to LoadLibraryW to load the DLL:

Press enter or click to view image in full size



The screenshot displays a debugger window with two panes. The left pane shows the assembly code for the function `FUN_1800014ce`. The right pane shows the decompiled C++ code for the same function.

Assembly Code (Left Pane):

```
1800014ce 48 03 ec 30 SUB RSP,0x30
1800014d2 48 04 05 LEA RAX,[DAT_180173930]
57 24 17 00 MOV qword ptr [RSP + local_8],RAX=>DAT_180173930
1800014d9 48 09 44 MOV RAX,qword ptr [RSP + local_8]
24 30 MOV RAX,qword ptr [RSP + local_8]
1800014de 48 0b 44 MOV RAX,qword ptr [RSP + local_8]
24 30 MOV RAX,qword ptr [RSP + local_8]
1800014e3 48 0b 08 MOV RAX,qword ptr [RAX + 0x8]
1800014e7 ff 15 53 CALL qword ptr [RAX + 0x1553]
65 16 00 MOV RAX,qword ptr [RAX + 0x651600]
1800014ed 48 09 44 MOV RAX,qword ptr [RSP + local_8]
24 28 MOV RAX,qword ptr [RSP + local_8]
1800014f2 48 03 7c CMP qword ptr [RSP + local_10],0x0
24 28 00 MOV RAX,qword ptr [RAX + 0x242800]
1800014f8 75 02 JNE LAB_1800014fc
1800014fa eb 1b JMP LAB_180001517

LAB_1800014fc
1800014fc 48 0b 44 MOV RAX,qword ptr [RSP + local_10]
24 28 MOV RAX,qword ptr [RSP + local_10]
180001501 48 0b 4c MOV RAX,qword ptr [RSP + local_8]
24 30 MOV RAX,qword ptr [RSP + local_8]
180001506 0b 49 10 MOV ECX,dword ptr [RAX + 0xb4910]
180001509 89 c9 MOV ECX,ECX
18000150b 48 01 c8 ADD RAX,ECX
18000150e 48 09 44 MOV RAX,qword ptr [RSP + local_8],RAX
24 20 MOV RAX,qword ptr [RSP + local_8],RAX
```

Decompiled Code (Right Pane):

```
void FUN_1800014ce(void)
{
    HMODULE pHVar1;
    pHVar1 = LoadLibraryW(DAT_180173930);
    if (pHVar1 != (HMODULE)0x0) {
        (*code *)((longlong)&pHVar1->unused + (ulonglong)DAT_180173940))();
    }
    return;
}
```

If the load fails, the function returns immediately. If it succeeds, it reads a hardcoded offset (0x2A40) from the same structure, adds it to the base address of the freshly loaded module, and calls the result — handing execution to a fixed location inside `WebView2Loader.dll`. Following the stub's indirect call through the import table confirmed the imported function is `LoadLibraryW`, which confirms the mechanism: load a DLL by name, then jump to a fixed offset inside it.

That left one question: who actually runs the stub? Its only caller is a single function, and decompiling it explains the whole design. It is a `libc++` static initializer, the routine that constructs the global C++ stream objects. wrapped in the `_Init_thread_header/_Init_thread_footer` pair the MSVC runtime uses for thread-safe, one-time initialization. The CRT runs this kind of initializer automatically the first time the DLL is loaded, before `DllMain` and without any export being called. The threat actor replaced one call inside it with a call to the stub. The original constructor is still invoked elsewhere in the same function, so the DLL's stream setup completes normally and Opera's startup looks entirely ordinary, the only difference is that the function now also runs the loader stub once, automatically, the moment Opera maps the DLL.

Knowing that the trojanized Opera DLL loads `WebView2Loader.dll`, the next stop was that DLL itself.

Analyzing WebView2Loader.dll

The first question was whether this was the real Microsoft DLL or a modified copy, and file size answered it before a single instruction was examined. The genuine `WebView2Loader.dll` is 141,600 bytes and scores 0/75 on VirusTotal; this sample is 165,968 bytes, a difference of 24 KB that no binary patching exercise can account for, particularly when the entire footprint of the Opera DLL modification was roughly 150 bytes. This file was compiled from scratch and then dressed up with Microsoft's real version metadata to pass a file-property check.

Opening it in Ghidra returned 568 functions and over 400 strings, most of them exactly what you'd expect from a real WebView2 library: public API exports

(`CreateCoreWebView2EnvironmentWithOptions`, `CompareBrowserVersions`, etc.), CRT boilerplate, locale tables. The legitimate WebView2 code is there to keep Opera functional and to make the DLL look credible in any directory listing or static scan. Among all those strings, one broke the pattern — `audio.lib` at `0x180023296`.

From the Opera DLL analysis we already knew where execution would land: RVA `0x2A40`, or `0x180002A40` as a virtual address. Asking Ghidra to decompile that address returned `FUNCTION_NOT_FOUND`, which was itself informative. The function immediately before it, `FUN_180002A0F`, carried a decompiler warning about overlapping instructions at `0x180002A10`, a single junk byte placed so that a disassembler starting at the wrong offset decodes garbage and fails to recognize the real function boundary 49 bytes later. Reading the raw bytes at `0x180002A40` directly worked around this, revealing the actual prologue and control flow:

```
PUSH R15/R14/R13/R12/RBP/RDI/RSI/RBX    ; save registers
SUB  RSP, 0xA8                          ; 168-byte stack frame
CALL 0x180002940                         ; timing delay
MOV  RCX, [0x1800231ED]                  ; config pointer
CALL [LoadLibraryW via IAT]              ; load next stage
MOV  RBX, RAX                           ; save handle
MOVSXD RAX, DWORD [RAX+0x3C]             ; e_lfanew
MOV  EAX, [RBX+RAX+0x88]                  ; export directory RVA
ADD  RAX, RBX                           ; -> IMAGE_EXPORT_DIRECTORY
MOV  EDX, [RAX+0x24]                     ; AddressOfNameOrdinals
```

The first call out of the entry point, to `0x180002940`, turned out to be a deliberate stalling tactic. Decompiling `FUN_180002a0f` showed a loop terminating on `param_1 != 1700000000`, 1.7 billion iterations, with a XOR-shift sequence running over a stack array on each pass. The array-based XOR-shift means the loop body can't be optimized away, and it simply burns wall-clock time before anything interesting happens. Sandboxes running on an execution time budget will often time out and report a clean verdict before this loop even finishes.

I then went after the `audio.lib` string directly. A cross-reference query for everything reading from `0x180023296` returned exactly two hits, both inside `FUN_180002ed0`, and no legitimate `WebView2` code path touches that address, confirming the string belongs to a config block rather than sitting there as a stray literal. The decompile was noisy because the function inherits live register state from its callers in a way Ghidra can't fully reconstruct, leaving many variables as `unaff_*` placeholders, but the logic underneath was clear. A loop iterates the target module's export name table, computing a rolling hash over each character with `param_1 = in_EAX + param_1 * 2` and comparing the result against a stored value — another instance of hash-based export resolution that avoids any traceable API call.

```
do {
    unaff_R12[lVar10] = (short)(char)PTR_s_audio_lib_180023239[lVar10];
    lVar10++;
} while (PTR_s_audio_lib_180023239[lVar10] != '\0');
```

This converts `audio.lib` into a wide string on the stack and writes it into a struct that gets passed into the final indirect call:

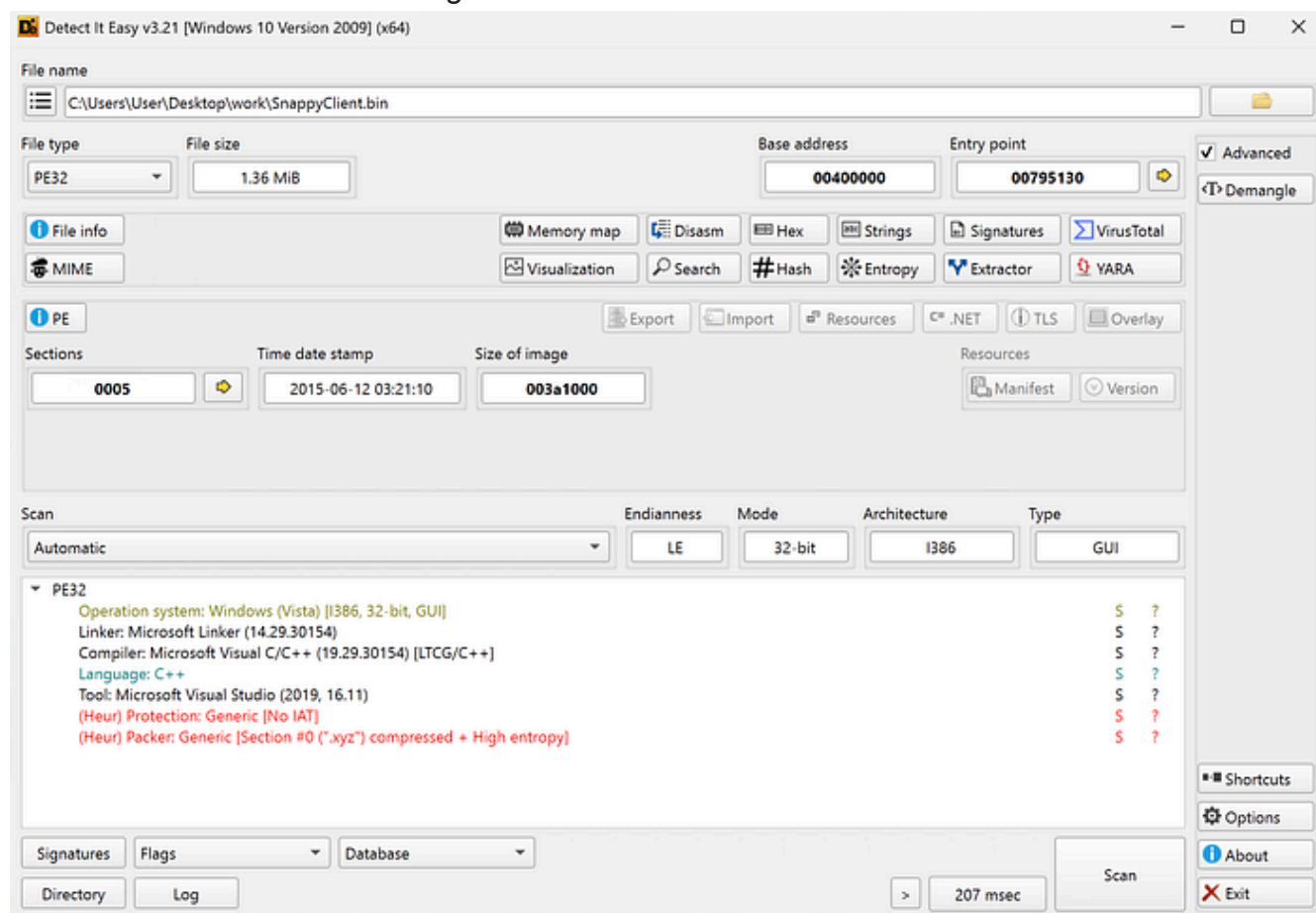
```
(*(code *) (lVar5 + (ulonglong)uVar2))(&stack0x00000088);
```

With the full chain mapped — `opera_elf.dll` redirecting into `WebView2Loader.dll`, which resolves and calls into `audio.lib` — the next step was to open the bundle and see what was inside it.

Extracting the SnappyClient PE from audio.lib

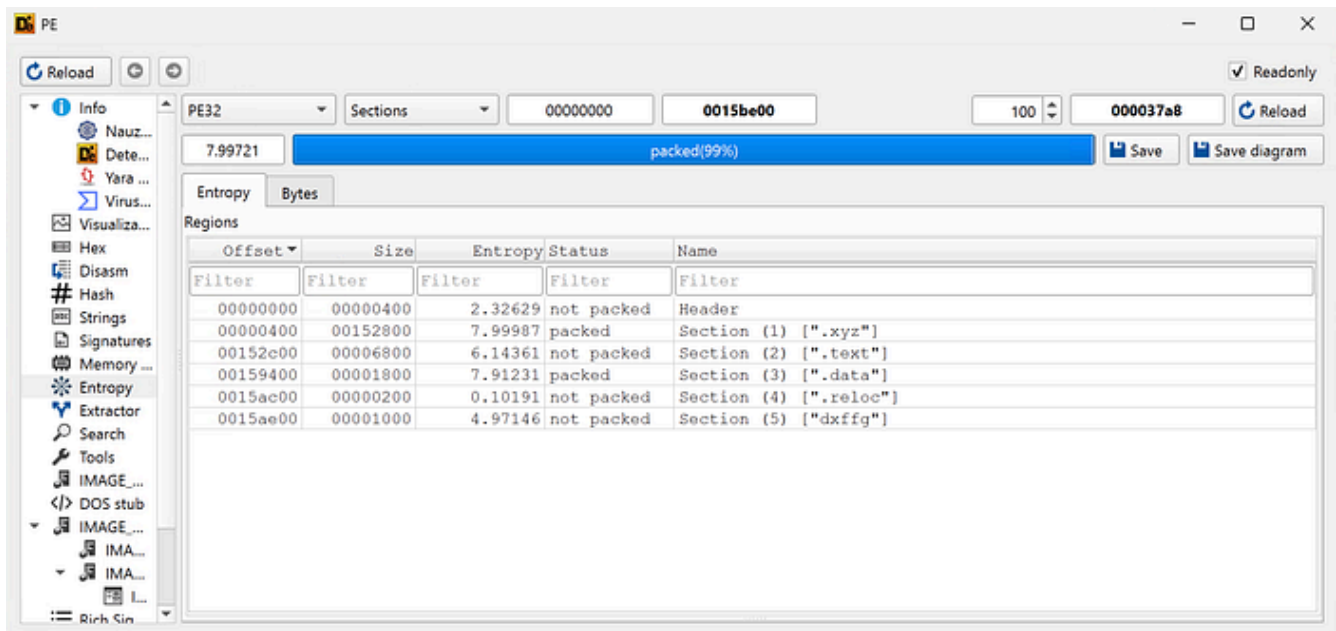
`audio.lib` is the `HijackLoader` resource bundle, and I parsed it using the same procedure I'd already built for the `Vidar` branch in the previous post, same bundle format, same extraction script, just a different set of embedded modules and a different final payload. Running the script against `audio.lib` recovered a bundle of `HijackLoader` components (exactly like in the first infection chain) and a final payload, a PE32 GUI executable.

Press enter or click to view image in full size



A look at the PE header and section table told me this was packed before I'd disassembled a single instruction. Whole-file entropy came back at 7.9 (a normal MSVC PE sits in the 5.5–6.8 range), so anything close to the theoretical maximum of 8 means most of the file is compressed or encrypted. As usual, the import directory was empty, which is structurally impossible for a real Windows binary that needs to call any API at all, and means whatever this PE does, it has to resolve its own imports at runtime through an unpacker stub.

Press enter or click to view image in full size



The section table had five entries, two of them with names that don't belong to any MSVC default layout: .xyz and dxffg:

- The .xyz section sat at VA 0x401000 with a virtual size of 0x394000 (3.7 MB), entropy at a flat 8.000, and characteristics marking it read-write-execute. The .text section, by contrast, had a virtual size of only 30 KB, far too small to hold a real malware body, which told me it was the unpacker stub and that the entry point would live inside it.
- dxffg was 4 KB of pointer-shaped DWORDs, scratch space the unpacker would populate as a runtime IAT once it had resolved its imports. And SizeOfImage came out to 0x3A1000, about 2.7 times the size of the file on disk, consistent with a section that's going to decompress substantially once it's mapped.

The structural model practically wrote itself: .xyz holds an encrypted or compressed PE, and .text is a small stub whose job is to decode it.

I opened the entry point in Ghidra, at 0x00795130, inside the 30 KB .text stub. Decompiling it at a high level showed the unpacking logic laid out plainly:

```
local_c = FUN_00798ff0(600, 4); // allocate control struct
FUN_00797bc0(local_c);         // populate its vtable
local_10 = FUN_00795020();      // get section-table address
local_14 = FUN_00795050(0x7461642e); // find section starting with ".dat"
local_1c = FUN_00795050(0x7865742e); // find section starting with ".tex"
local_24 = FUN_007950c0(local_c, local_10, &local_18); // decode section
local_28 = FUN_007950c0(local_c, local_14, &local_20); // decode section
local_8 = (*(code **)(local_c + 0x24))(0x70); // alloc 0x70 byte struct
```

Two things came out of reading this. First, the unpacker locates sections by the first four bytes of their name rather than by a full string match: 0x7461642e is ".dat" little-endian, matching .data, and 0x7865742e is ".tex", matching .text. The first decode call passes local_10, the section table base, which structurally starts at the first section header, so that call is decoding whatever the first section happens to be, and for this PE that's .xyz. Second, the actual decode work happens inside FUN_007950c0, which is where I went looking next.

Decompiling FUN_007950c0 gives us this code:

```
local_c = (byte*)(module_base + section_header->VirtualAddress);
local_8 = FUN_00799ac0(&local_c);
local_10 = alloc(local_8);
*param_3 = local_8;
local_14 = local_8 - 5;
FUN_00799ad0(local_10, &local_8, local_c + 5, &local_14, local_c, 5);
return local_10;
```

The call signature on FUN_00799ad0, (dst, &dst_len, src, &src_len, props, prop_size=5) with the property size hardcoded to 5, is the canonical decode signature from the LZMA SDK's LzmaDecode family, and I recognized it immediately because it's a signature I've seen often enough in unpackers that it's basically diagnostic on its own. Stepping into FUN_00799ac0 confirmed it does nothing more interesting than read a 32-bit value and advance the pointer past it, which is exactly what reading an uncompressed-size header looks like.

With the format fully known, decoding it was a short Python script. lzma's raw LZMA1 mode with explicit filters handles it directly:

```
import struct, lzma
d = open('SnappyClient.bin', 'rb').read()
xyz = d[0x400 : 0x400 + 0x152800]
uncomp_size = struct.unpack('<I', xyz[0:4])[0]
props = xyz[4:9]
comp_data = xyz[9:]
filt = [{
    "id": lzma.FILTER_LZMA1,
    "dict_size": struct.unpack('<I', props[1:5])[0],
    "lc": props[0] % 9,
    "lp": (props[0] // 9) % 5,
    "pb": props[0] // 45,
}]
dec = lzma.LZMADecompressor(format=lzma.FORMAT_RAW, filters=filt)
out = dec.decompress(comp_data, max_length=uncomp_size)
open('snappy_unpacked_xyz.bin', 'wb').write(out)
```

That produced exactly 3,710,976 bytes, opening with 4D 5A 90 00, a valid MZ header. The obvious next step was to upload the unpacked PE to VirusTotal, which confirmed the obvious: this is SnappyClient RAT.

File: snappy_unpacked_xyz.bin

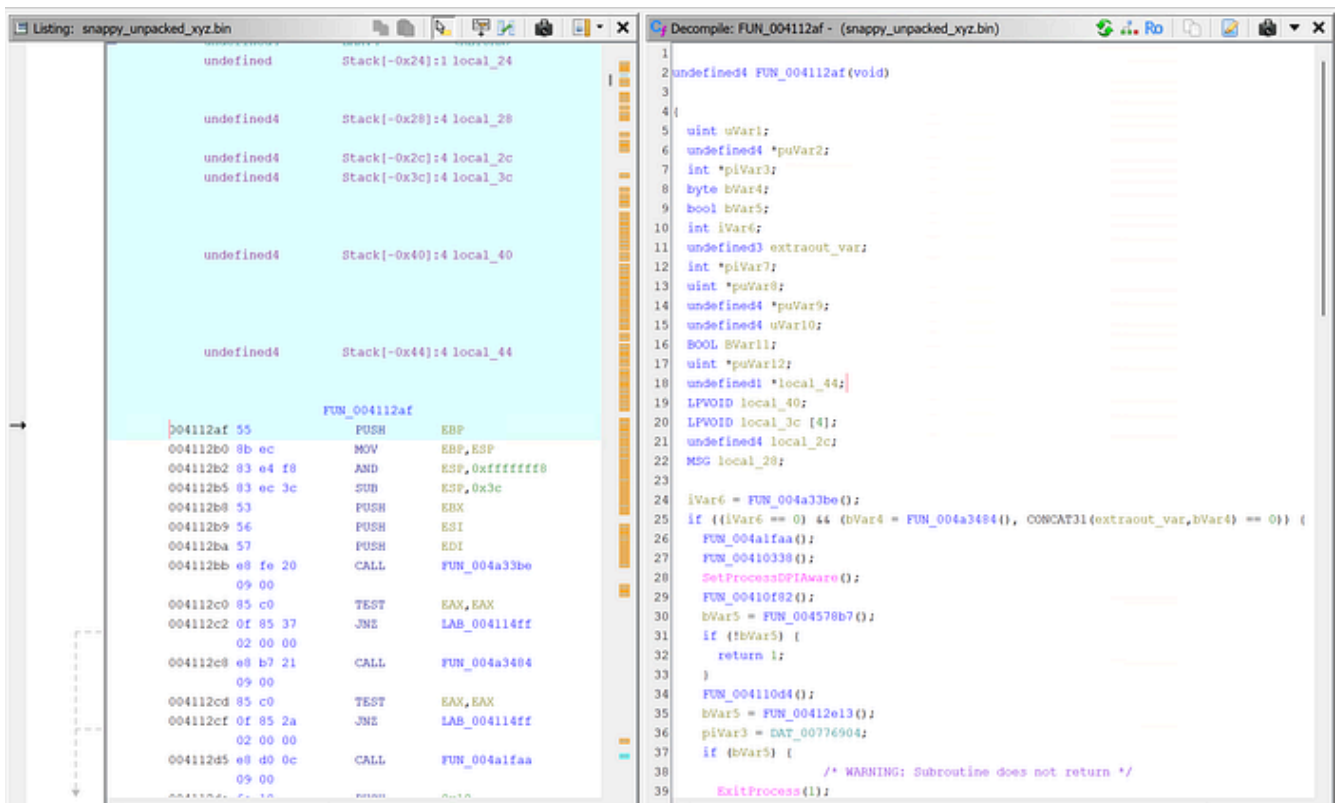
Size: 3,710,976

SHA-256: 81c4da1203d1b492924c83e8f20afed84cb647983ebf7e2ba4e6895d53994615

Static Analysis of SnappyClient

With a clean, unpacked PE in hand, I could finally start reverse-engineering SnappyClient itself. The entry point sits at 0x005ce3f3, and after the usual MSVC runtime initialization, execution lands in FUN_004112af, which acts as SnappyClient's main startup routine.

Press enter or click to view image in full size



This function performs early process setup, loads the embedded configuration, applies local runtime settings, checks whether the host should be banned/exited, initializes malware subsystems, decrypts the network bootstrap, starts the main client object, and then enters a normal Windows message loop:

```
FUN_004112af()  
{  
    early_init_1();  
    early_init_2();
```

```

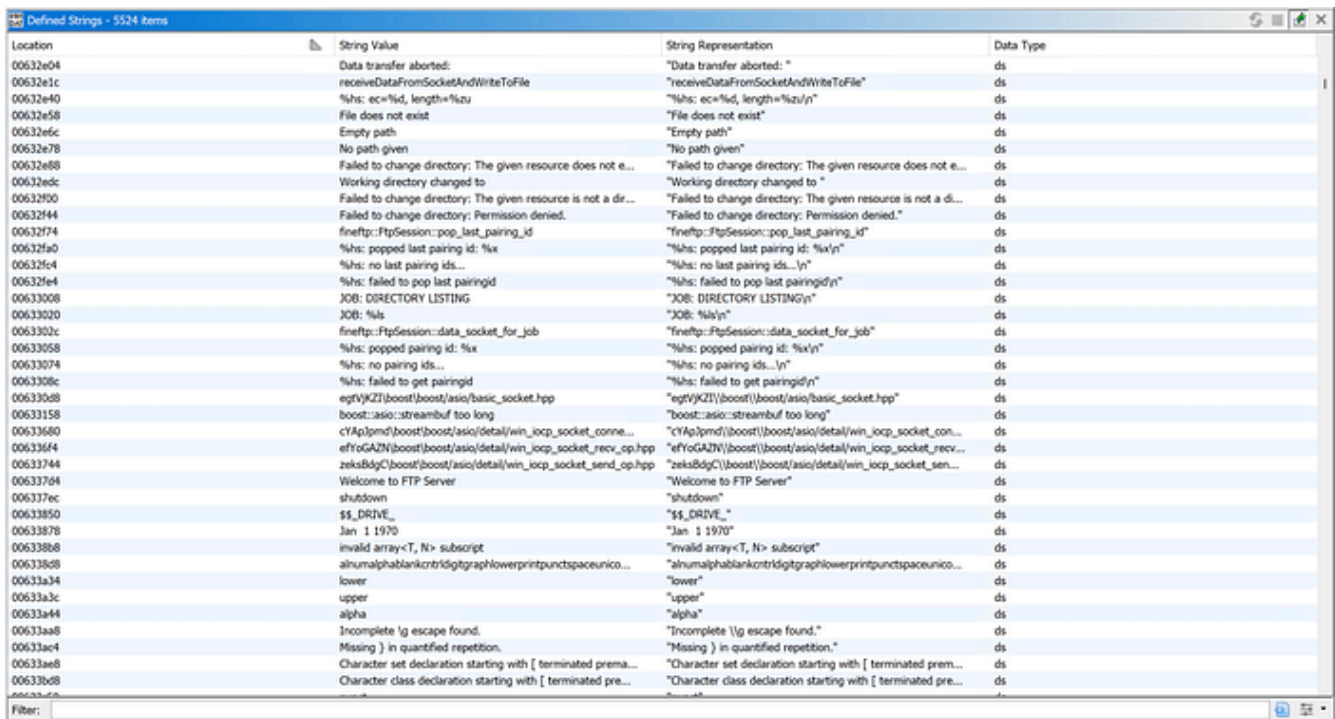
process/runtime setup;
SetProcessDPIAware();
parse_embedded_config();
if config parse failed:
    return 1;
apply_local_paths_and_config();
if banned_host_or_banned_marker_check():
    ExitProcess(1);
if config["e"] exists:
    initialize latest-version/single-instance logic;
if config["m"] exists:
    initialize mutex / instance guard;
    if mutex/init failed:
        return 1;
initialize core subsystems;
initialize event/software DB handlers;
initialize filesystem/keylog/appbound paths;
decrypt/load network bootstrap config;
start main async client object;
while GetMessageW():
    TranslateMessage();
    DispatchMessageW();
}

```

The control flow gives you the skeleton, but the more useful signal came from strings, RTTI, and cross-references into the main initialization code. Several statically-linked libraries immediately stood out:

- Boost.Asio and Windows IOCP networking code
- Snappy compression
- ChaCha20-Poly1305 and hashing code
- 7-Zip archive handling
- libcurl
- Crashpad
- FineFTP
- JPEG and RFB/VNC-related code
- Browser credential and profile handling code

Press enter or click to view image in full size



Location	String Value	String Representation	Data Type
00632e04	Data transfer aborted:	"Data transfer aborted: "	ds
00632e1c	receivedDataFromSocketAndWriteToFile	"receivedDataFromSocketAndWriteToFile"	ds
00632e40	%hs: ec=%id, length=%zu	"%hs: ec=%id, length=%zu/n"	ds
00632e58	File does not exist	"File does not exist"	ds
00632e6c	Empty path	"Empty path"	ds
00632e78	No path given	"No path given"	ds
00632e88	Failed to change directory: The given resource does not e...	"Failed to change directory: The given resource does not e..."	ds
00632e9c	Working directory changed to	"Working directory changed to "	ds
00632f00	Failed to change directory: The given resource is not a dir...	"Failed to change directory: The given resource is not a di..."	ds
00632f44	Failed to change directory: Permission denied.	"Failed to change directory: Permission denied."	ds
00632f74	finetp::FtpSession::pop_last_pairing_id	"finetp::FtpSession::pop_last_pairing_id"	ds
00632fa0	%hs: popped last pairing id: %ix	"%hs: popped last pairing id: %ix/n"	ds
00632fc4	%hs: no last pairing ids...	"%hs: no last pairing ids.../n"	ds
00632fe4	%hs: failed to pop last pairingid	"%hs: failed to pop last pairingid/n"	ds
00633008	JOB: DIRECTORY LISTING	"JOB: DIRECTORY LISTING/n"	ds
00633020	JOB: %ls	"JOB: %ls/n"	ds
0063302c	finetp::FtpSession::data_socket_for_job	"finetp::FtpSession::data_socket_for_job"	ds
00633058	%hs: popped pairing id: %ix	"%hs: popped pairing id: %ix/n"	ds
00633074	%hs: no pairing ids...	"%hs: no pairing ids.../n"	ds
0063308c	%hs: failed to get pairingid	"%hs: failed to get pairingid/n"	ds
00633088	egvYKZl\\boost\\asio\\basic_socket.hpp	"egvYKZl\\boost\\asio\\basic_socket.hpp"	ds
00633158	boost::asio::streambuf too long	"boost::asio::streambuf too long"	ds
00633680	cYAp\\pmd\\boost\\asio\\detail\\win_iocp_socket_conne...	"cYAp\\pmd\\boost\\asio\\detail\\win_iocp_socket_conne..."	ds
006336f4	efYcGAZl\\boost\\asio\\detail\\win_iocp_socket_recv_op.hpp	"efYcGAZl\\boost\\asio\\detail\\win_iocp_socket_recv..."	ds
00633744	zeksBdgC\\boost\\asio\\detail\\win_iocp_socket_send_op.hpp	"zeksBdgC\\boost\\asio\\detail\\win_iocp_socket_sen..."	ds
00633764	Welcome to FTP Server	"Welcome to FTP Server"	ds
006337ec	shutdown	"shutdown"	ds
00633850	\$\$_DRIVE_	"\$\$_DRIVE_"	ds
00633878	Jan 1 1970	"Jan 1 1970"	ds
00633888	invalid array<T, N> subscript	"invalid array<T, N> subscript"	ds
00633898	alnumalphanblankcntrldgtgraphlowerprintpunctspaceunico...	"alnumalphanblankcntrldgtgraphlowerprintpunctspaceunico..."	ds
00633a34	lower	"lower"	ds
00633a3c	upper	"upper"	ds
00633a44	alpha	"alpha"	ds
00633a98	Incomplete \\g escape found.	"Incomplete \\g escape found."	ds
00633ac4	Missing) in quantified repetition.	"Missing) in quantified repetition."	ds
00633ae8	Character set declaration starting with [terminated prema...	"Character set declaration starting with [terminated prem..."	ds
00633b08	Character class declaration starting with [terminated pre...	"Character class declaration starting with [terminated pre..."	ds

That already gives us a strong hint about the shape of the malware. This is not a simple stealer, it is a full C2 implant with credential theft, file transfer, proxying, remote desktop, and tasking support.

Mapping Capabilities

Large C++ malware can be painful to reverse because the real behavior is spread across classes, handlers, callbacks, and request builders. Fortunately, this sample still had enough RTTI and strings to map the capability surface quickly. The first thing I wanted to answer was simple: What high-level components does this binary actually contain? The answer was visible from class names and nearby strings.

Browser and credential theft

The binary carries a full set of handlers and builders for browser data:

- Chromium browser passwords
- Chromium cookies
- Browser extensions
- Browser profile cloning
- Mozilla/Firefox credential handling
- Generic software credential collection
- Chromium App-Bound Encryption handling

Relevant class names included:

BrowserCredentialsResponseBuilder
BrowserCookiesResponseHandler
BrowserPasswordsResponseHandler
BrowserExtensionResponseHandler
BrowserClonerResponseHandler
MultiBrowserCredentialsResponseHandler
MozillaCredentialsResponseBuilder
CAppBoundDynamo
CAppBoundExecutor
CDynamoDBReader
CBrowserCloner
CBrowserProfileCopier

Taken together, that's a complete browser-collection subsystem.

Remote desktop and hidden VNC

The sample also contains a substantial remote-desktop stack:

CHVnc
HVncResponseBuilder
HVncResponseHandler
CVncServer
CSharedVnc
RfbClient
RfbDispatcher
ScreenDriver
WinD3D11Device
WinDxgi
CursorShapeDetector
WindowsClipboard
CtrlAltDelSimulator
WindowsInputBlocker

The presence of RFB/VNC classes, DXGI/D3D11 screen capture, clipboard support, cursor handling, and input simulation points to interactive remote control, not just screenshots.

File, process, shell, and proxy features

The binary also contains:

- File browsing and recursive file operations
- File search and file scanning
- Archive creation/extraction through 7-Zip
- Process listing and process control

- Remote command shell support
- Download and execute logic
- FTP server functionality
- SOCKS5 proxy functionality
- RLOGIN-style console functionality

Examples of useful strings and class names:

```
CFileDownloader
CFileScanner
CFileSearcher
FileTransferRequestHandler
RemoteFileBrowserResponseBuilder
CProcessList
ProcessManagerResponseBuilder
CFtpServer
CSocks5Server
CRloginServer
CConsoleHost
cmd.exe /q /u
cmd.exe /c start "" "%s" %s
```

The goal at this stage wasn't to reverse every command in full, but to pin down the capability map. As we've just seen, SnappyClient is a general-purpose RAT and its C2 decides which parts of that arsenal to switch on for any given infection. That raised the obvious next question: What is *this* campaign actually configured to do? Answering it meant recovering the malware's configuration and C2.

Recovering the Build Configuration

The RTTI mapping told me what the binary could do, but not what it was actually configured to do on this campaign: the config keys (tag, dd, ef, sf, and the rest) needed values, and the C2 endpoint itself was nowhere in plaintext. The first real configuration artifact was easy to find with a string search. At virtual address 0x0065f2d0, the binary contains this JSON object:

```
{
  "tag": "TIGRAN",
  "buildId": "main",
  "dd": "%ALLUSERSPROFILE%\\Mayanex",
  "ef": "Jifexabo",
  "sf": "Mitonico",
  "kf": "Kapuq",
  "c": "Nata",
  "ab": "naijaco",
  "e": true
}
```

That JSON is essentially the local build/runtime configuration, and each key has the following meaning:

Press enter or click to view image in full size

Key	Meaning
tag	Campaign/operator tag sent during registration.
buildId	Build identifier sent during registration.
dd	Base directory for malware data.
ef	Encrypted EventsDB filename.
sf	Encrypted SoftwareDB filename.
kf	Keylogger filename.
c	Directory used for a per-host banned marker.
ab	Directory used for Chromium App-Bound Encryption key material.
e	Single-instance/latest-version enforcement flag.

The *tag* and *buildId* values are especially useful because they later appear in the malware's registration message to C2.

Recovering the C2 Configuration

Sitting close to the plaintext config was a non-printable blob at VA 0x0065f6d2. Ghysa showed startup code calling `FUN_00504d10(&DAT_0065f6d2, 0x2c3, output)`, and `FUN_00504d10` matched a raw Snappy decompressor, which, given the malware family's name, is obvious in retrospect. After Snappy decompression, the blob expands to 1,316 bytes with this layout:

Press enter or click to view image in full size

Offset	Size	Meaning
0x0000	2	Encrypted payload size: 138 bytes
0x0002	2	Context/archive size: 518 bytes
0x0004	138	Encrypted first-stage payload plus authentication tag
0x008e	518	Context data used for key derivation and archive reconstruction
after context	656	Zero padding

To make it easier to follow, I will call the two important pieces **seg1** (the encrypted first-stage payload) and **seg2** (the 518-byte context/archive data). Between them, they're wrapped in several layers of protection: Snappy compression, Base58 encoding, ChaCha20-Poly1305 encryption, a modified RIPEMD-160 routine, and a password-protected 7z archive. It sounds excessive, but the flow is logical once it's laid out:

1. Use seg2 to derive crypto material
2. Use that crypto material to decrypt seg1
3. Use the decrypted seg1 value as both a password and encrypted data
4. Rebuild a hidden 7z archive from seg2
5. Extract a key/nonce from the archive
6. Use that key/nonce to decrypt the final host list

Before reconstructing all of that by hand, I first attempted to automatically extract the C2 config using a Python script from Zscaler — ThreatLabz's public SnappyClient research. However, that tool was built for a different version of SnappyClient and didn't work on my sample, but it still gave me enough pointers to work out the rest.

Join Medium for free to get updates from this writer.

Remember me for faster sign in

Stage 1: derive a key from the context block The 518-byte seg2 block is used as the seed for the first key derivation. Conceptually:

```
key0    = SHA256(seg2)
nonce0  = repeated SHA1(seg2), truncated to 12 bytes
```

From here, the malware Snappy-compresses seg2, encrypts it with ChaCha20-Poly1305, Base58-encodes the result, and runs the modified RIPEMD-160/Base58/SHA-256 derivation chain over that data. The final result is the real key and nonce used to decrypt seg1:

```
key1    = 1d058c4a2b84a6a5df47270a01f71794481d10d6ce8a44c5426e8a87ce9ec7ce
nonce1  = ec78cfed78bae5f15306cbb8
```

Stage 2: decrypt the first-stage payload seg1 is 138 bytes total, 122 bytes of ciphertext, plus a 16-byte authentication tag. Decrypting it with key1 and nonce1 produces Snappy-compressed data, which then results in this Base58 string:

```
7smfQP3NdFYRziL9Ym8tFUC9Q9sGH27Zf35FaxVCVMYiCtK3561BXhZL8qfg4ZTpEbdWQMjggq5j8x5NTW53buS
Q9EC3bzLE4L2HqcvNzP4Rn2JqQA5RVBZp
```

This string is the reason the extraction feels confusing at first, because it does two jobs at once: it's the password for an embedded 7z archive, and it's also a Base58-encoded, encrypted host list. The same recovered value gets used in two completely different ways.

Stage 3: rebuild the embedded 7z archive The 518-byte seg2 block is almost a 7z archive, but not quite. It starts with 8 bytes that need to be replaced with the normal 7z signature. The reconstruction is:

```
archive = 37 7A BC AF 27 1C 00 04 || seg2[8:]
```

The rebuilt archive then opens with the Base58 password recovered from stage 2. Inside the archive are several Base58-looking filenames, and one of them contains the key material needed to decrypt the final host list:

```
467Urzbh66QBsqdH98EAYkGBEDdu
```

That file contains 44 bytes:

```
32-byte key:
ae49a31ccd0711ce7063ab486984a504cb9ca5690851a92b6e19b0d41addfa88
12-byte nonce:
2b263f15105a07bad47c48fc
```

The remaining files hold the port configuration, which in this sample resolves to:

```
{"p":3333,"dp":3334}
```

where p is the control port and dp is the data port.

Stage 4: decrypt the host list The last step reuses the Base58 string from stage 2. Decoded to bytes, its final 16 bytes are the Poly1305 tag and everything before them is ciphertext.

Decrypting that with the archive-derived key and nonce leaves one more layer of Snappy compression. After decompression, the host list is finally visible:

```
66.163.113.238;yoda-tube.sbs;yoda-line.site;yoda-vac.online;yoda-tape.sbs;yoda-pipe.site
```

Which assembles into the final network configuration:

```
{
  "hosts": [
    "66.163.113.238",
    "yoda-tube.sbs",
    "yoda-line.site",
    "yoda-vac.online",
    "yoda-tape.sbs",
    "yoda-pipe.site"
  ],
  "control_port": 3333,
  "data_port": 3334
}
```

Once I had the entire chain working, I turned it into a Python script so the next sample from this family won't require the same manual effort. The extractor does the following:

1. Reads the PE
2. Finds the plaintext build config
3. Locates the Snappy-compressed network bootstrap
4. Decompresses the bootstrap
5. Recreates the modified RIPEMD-160 and ChaCha20-Poly1305 key derivation
6. Rebuilds the embedded 7z archive
7. Extracts the archive key/nonce and port records
8. Decrypts and parses the host list
9. Writes the final config to a JSON file

At this point the static picture was complete. I knew what SnappyClient was built to do and, more usefully for tracking this campaign, exactly what this particular build was pointed at: the TIGRAN tag, the Mayanex data directory, and a primary C2 of 66.163.113[.]238 with five yoda-* fallback domains on ports 3333 and 3334.

Dynamic Analysis

With the static analysis complete, the next step was to confirm it against the malware's actual behavior. I detonated the sample in an isolated VM, with Procmon and Wireshark running, and traced the full execution from the initial sideload through to live C2 tasking. I've organized

this section around MITRE ATT&CK tactics, which suits a behavioral sequence better than a flat chronological account.

The run produced the following process tree, with the parent-child relationships it revealed:

WizardDa42.exe	← parent orchestrator
└─ AlphVector.exe	← first-stage Opera sideload host
└─ AlphVector.exe	← orchestrator child
└─ VirtualAr.exe	← SnappyClient process-hollow target
└─ chrome.exe	← Spawned for AppBound key extraction
└─ Crisp.exe	← persistence cover
elevation_service.exe	← Chrome IElevator
elevation_service.exe	← second invocation

In practice, the first-stage `AlphVector.exe` launches from the desktop, copies the entire loader chain into `C:\ProgramData\service`, and relaunches itself from that persistent location. From there it side-loads the trojanized `opera_elf.dll`, whose patched initializer loads `WebView2Loader.dll`. That component reads the `audio.lib` bundle, decrypts the SnappyClient PE, and writes it to a randomly-named `.tmp` file in `%Temp%`. It then creates the signed process `VirtualAr.exe` in a suspended state, maps SnappyClient into it through process hollowing, and resumes it.

With the implant live, `Crisp.exe` installs persistence and drops out of the picture, and SnappyClient begins its own work. It first tries its primary C2, fails, resolves a fallback address, registers over the control port, opens its parallel data sessions, and starts data collection by spawning Chrome to extract its App-Bound keys and staging stolen data into the Mayanex directory. From first execution to active C2 tasking, the whole run completes in a little over two minutes, with no human interaction at any point.

Execution

The sideload chain I'd reconstructed statically shows up in the image-load events exactly as predicted:

```
AlphVector.exe Load Image C:\ProgramData\service\opera_elf.dll
AlphVector.exe Load Image C:\ProgramData\service\WebView2Loader.dll
```

`opera_elf.dll` loads through Windows' default DLL search order, its patched static initializer fires `LoadLibraryW("WebView2Loader.dll")`, and that second component goes on to read `audio.lib` in a single pass. The most diagnostic single event in the entire trace, though, is what happens right after `VirtualAr.exe` starts:

Press enter or click to view image in full size

VirtualAr.exe	1588	Process Start		SUCCESS	Parent PID: 10068, Command line: C:\Users\User\AppData\Local\Temp\5733CD6.tmp
VirtualAr.exe	1588	Thread Create		SUCCESS	Thread ID: 7176
VirtualAr.exe	1588	Load Image	C:\Users\User\AppData\Local\Temp\5733CD6.tmp	SUCCESS	Image Base: 0x500000, Image Size: 0x3a9000
VirtualAr.exe	1588	Load Image	C:\Users\User\AppData\Local\Temp\VirtualAr.exe	SUCCESS	Image Base: 0xdb0000, Image Size: 0x11c000
VirtualAr.exe	1588	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x77fb4ee000, Image Size: 0x266000
VirtualAr.exe	1588	Load Image	C:\Windows\SysWOW64\ntdll.dll	SUCCESS	Image Base: 0x77390000, Image Size: 0x1bf000

Two details here are unambiguous evidence of process hollowing. First, 5733CD6.tmp gets mapped as an executable image before VirtualAr.exe's own image is loaded. Normal process startup never loads anything before the executable's own image. Seeing a .tmp file mapped first means the parent wrote SnappyClient's bytes into the suspended child's address space, redirected its entry point with SetThreadContext, and resumed the thread before VirtualAr.exe got a chance to run on its own. Second, there's a roughly four-second gap between Process Start and any further activity. A normal process maps ntdll.dll and gets its main thread running within tens of milliseconds, not seconds, so that idle window is exactly the time the parent needed to finish the hollowing operation externally before resuming the thread.

Persistence

Crisp.exe exists to plant the chain's persistence and then steps out of the picture. It drops two artifacts, both chosen for low visibility. The first is a logon Startup shortcut, prompt_analyzer_debug.lnk, written into the user's ... \Start Menu\Programs\Startup\ folder, which relaunches the chain at every interactive logon. The second is a scheduled task:

```
Crisp.exe CreateFile C:\Windows\Tasks\socket_dispatcher_v2.job
Crisp.exe WriteFile C:\Windows\Tasks\socket_dispatcher_v2.job
```

The task name, socket_dispatcher_v2, reads plausibly enough in taskschd.msc to pass a casual glance, as it sounds like exactly the kind of name a developer would give a legitimate background networking service. More interesting than the task itself is the format, as this is legacy Task Scheduler 1.0, the binary .job format, not the modern XML-based scheduler. The implementation goes through mstask.dll, which exposes a COM interface for AT-style jobs, meaning there's no schtasks.exe process spawn anywhere in this sequence, no command-line interpreter involved at all, just a direct COM call. That matters because it routes around the standard detection signal of schtasks.exe launching under a suspicious parent.

What ties the two choices together is that neither is where modern detection tends to look. Run-key autostarts and the XML-based scheduled tasks under \Microsoft\Windows\... are the heavily-instrumented paths, and this chain deliberately avoids both, reaching instead for a Startup-folder .lnk and a legacy Task Scheduler 1.0 .job under C:\Windows\Tasks\ — two older, lower-traffic mechanisms that a defender can easily miss.

Defense Evasion

The evasion behavior is spread through the run rather than front-loaded at startup. VirtualAr.exe, for example, checks for the VMware Tools daemon nine separate times across its lifetime:

```
VirtualAr.exe QueryNameInformationFile C:\Program Files\VMware\VMware Tools\vmtoolsd.exe
```

These checks are interleaved with active C2 traffic and credential collection, which tells us the implant is repeatedly re-evaluating its environment as it runs rather than just deciding once whether it's in a sandbox. Early on, VirtualAr.exe also queries HKLM\System\CurrentControlSet\Control\WMI\Restrictions six times in quick succession, a key consulted when ETW providers are about to register, which from a user-mode process with no obvious WMI need reads as a probe for whether ETW restrictions are active on the host.

Credential Access

The actual data stealing begins with Chrome, opening with full profile enumeration:

Press enter or click to view image in full size

Process Name	PID	Operation	Path	Result	Detail
VirtualAr.exe	1588	QueryAttribute...	C:\Users\User\AppData\Local\Google\Chrome\User Data\WasmiTtsEngine	SUCCESS	Attributes: D, ReparseTag: 0x0
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\WasmiTtsEngine	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\WidevineCdm	SUCCESS	Desired Access: Read EA, Read Attributes, t
VirtualAr.exe	1588	QueryAttribute...	C:\Users\User\AppData\Local\Google\Chrome\User Data\WidevineCdm	SUCCESS	Attributes: D, ReparseTag: 0x0
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\WidevineCdm	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\ZxcvbnData	SUCCESS	Desired Access: Read EA, Read Attributes, t
VirtualAr.exe	1588	QueryAttribute...	C:\Users\User\AppData\Local\Google\Chrome\User Data\ZxcvbnData	SUCCESS	Attributes: D, ReparseTag: 0x0
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\ZxcvbnData	SUCCESS	
VirtualAr.exe	1588	QueryDirectory	C:\Users\User\AppData\Local\Google\Chrome\User Data	NO MORE FILES	FileInformationClass: FileExtDtdDirectoryInfo
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	Desired Access: Read EA, Read Attributes, t
VirtualAr.exe	1588	QueryAttribute...	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	Attributes: A, ReparseTag: 0x0
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	Desired Access: Read Attributes, Synchroniz
VirtualAr.exe	1588	QueryStandard...	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	AllocationSize: 147,456, EndOfFile: 144,711
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	Desired Access: Read Data/List Directory, R
VirtualAr.exe	1588	QueryStandard...	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	AllocationSize: 147,456, EndOfFile: 144,711
VirtualAr.exe	1588	ReadFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	Offset: 0, Length: 144,711, Priority: Normal
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Local State	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\ProgramData\Mayanex\Natale61c376b7804d047aed6b6e9f48c8520774e301e	NAME NOT FOUND	Desired Access: Generic Read, Disposition:
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	Desired Access: Read EA, Read Attributes, t
VirtualAr.exe	1588	QueryAttribute...	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	Attributes: A, ReparseTag: 0x0
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	Desired Access: Read Attributes, Synchroniz
VirtualAr.exe	1588	QueryStandard...	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	AllocationSize: 32,768, EndOfFile: 31,849, N
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	Desired Access: Read Data/List Directory, R
VirtualAr.exe	1588	QueryStandard...	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	AllocationSize: 32,768, EndOfFile: 31,849, N
VirtualAr.exe	1588	ReadFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	Offset: 0, Length: 31,849, Priority: Normal
VirtualAr.exe	1588	CloseFile	C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Preferences	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\ProgramData\Mayanex\Natale61c376b7804d047aed6b6e9f48c8520774e301e	SUCCESS	Desired Access: Generic Write, Read Attribu
VirtualAr.exe	1588	WriteFile	C:\ProgramData\Mayanex\Natale61c376b7804d047aed6b6e9f48c8520774e301e	SUCCESS	Offset: 0, Length: 26, Priority: Normal
VirtualAr.exe	1588	CloseFile	C:\ProgramData\Mayanex\Natale61c376b7804d047aed6b6e9f48c8520774e301e	SUCCESS	
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\CocCoc\Browser\User Data	PATH NOT FOUND	Desired Access: Read Attributes, Dispositio
VirtualAr.exe	1588	CreateFile	C:\Users\User\AppData\Local\Microsoft\Edge\User Data	SUCCESS	Desired Access: Read Attributes, Dispositio
VirtualAr.exe	1588	QueryBasicInfo	C:\Users\User\AppData\Local\Microsoft\Edge\User Data	SUCCESS	CreationTime: 11-Jun-26 00:47:19, LastAccess

Local State carries the encrypted Chrome AppBound key, Preferences tells the implant which profiles and extensions exist, and the Cookies database gets opened later with Generic Read/Write and No Buffering, the SQLite WAL-aware open pattern stealers use specifically to read a live database file without risking a stale snapshot.

The most distinctive sequence in this whole section is the on-demand chrome.exe spawn and the Chrome elevation service activations that follow it:

Press enter or click to view image in full size

Process Name	PID	Operation	Path	Result	Detail
VirtualAr.exe	1588	Process Create	C:\Program Files\Google\Chrome\Application\chrome.exe	SUCCESS	PID: 5528, Command line: "C:\Program Files\Go
chrome.exe	5528	Process Start		SUCCESS	Parent PID: 1588, Command line: "C:\Program F
chrome.exe	5528	Thread Create		SUCCESS	Thread ID: 1884
VirtualAr.exe	1588	Load Image	C:\Windows\SysWOW64\ntdll.dll	SUCCESS	Image Base: 0x146e0000, Image Size: 0x1bf000
chrome.exe	5528	Load Image	C:\Program Files\Google\Chrome\Application\chrome.exe	SUCCESS	Image Base: 0x7ff7376f0000, Image Size: 0x42f
chrome.exe	5528	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x7ffab4ee0000, Image Size: 0x26
chrome.exe	5528	Load Image	C:\Windows\System32\kernel32.dll	SUCCESS	Image Base: 0x7ffab3c40000, Image Size: 0xc9f
chrome.exe	5528	Load Image	C:\Windows\System32\KernelBase.dll	SUCCESS	Image Base: 0x7ffab2170000, Image Size: 0x3ff
chrome.exe	5528	Load Image	C:\Windows\System32\apphelp.dll	SUCCESS	Image Base: 0x7ffaaf2b0000, Image Size: 0xa0f
chrome.exe	5528	Thread Create		SUCCESS	Thread ID: 1436
chrome.exe	5528	Load Image	C:\Program Files\Google\Chrome\Application\149.0.7827.155\chrome_elf.dll	SUCCESS	Image Base: 0x7ffa59530000, Image Size: 0x2b
chrome.exe	5528	Load Image	C:\Windows\System32\version.dll	SUCCESS	Image Base: 0x7ffa38a00000, Image Size: 0xb0
chrome.exe	5528	Load Image	C:\Windows\System32\msvcrt.dll	SUCCESS	Image Base: 0x7ffab3a20000, Image Size: 0xa9
chrome.exe	5528	Thread Create		SUCCESS	Thread ID: 5216
chrome.exe	5528	Load Image	C:\Windows\System32\bcryptprimitives.dll	SUCCESS	Image Base: 0x7ffab2a40000, Image Size: 0xa8
chrome.exe	5528	Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x7ffab4d80000, Image Size: 0x11
chrome.exe	5528	Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x7ffab3740000, Image Size: 0x1c
chrome.exe	5528	Load Image	C:\Windows\System32\win32u.dll	SUCCESS	Image Base: 0x7ffab2af0000, Image Size: 0x27f
chrome.exe	5528	Load Image	C:\Windows\System32\gdi32.dll	SUCCESS	Image Base: 0x7ffab4d50000, Image Size: 0x2b
chrome.exe	5528	Load Image	C:\Windows\System32\gdi32full.dll	SUCCESS	Image Base: 0x7ffab2b20000, Image Size: 0x12
chrome.exe	5528	Load Image	C:\Windows\System32\msvcp_win.dll	SUCCESS	Image Base: 0x7ffab2990000, Image Size: 0xa3
chrome.exe	5528	Load Image	C:\Windows\System32\ucrtbase.dll	SUCCESS	Image Base: 0x7ffab2570000, Image Size: 0x14
chrome.exe	5528	Load Image	C:\Windows\System32\imm32.dll	SUCCESS	Image Base: 0x7ffab48d0000, Image Size: 0x32
chrome.exe	5528	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x2238c430000, Image Size: 0x26
chrome.exe	5528	Load Image	C:\Windows\System32\ole32.dll	SUCCESS	Image Base: 0x7ffab4b20000, Image Size: 0x19
chrome.exe	5528	Load Image	C:\Windows\System32\combase.dll	SUCCESS	Image Base: 0x7ffab3d10000, Image Size: 0x38
chrome.exe	5528	Load Image	C:\Windows\System32\oleaut32.dll	SUCCESS	Image Base: 0x7ffab3590000, Image Size: 0xd9
chrome.exe	5528	Load Image	C:\Windows\System32\crypt32.dll	SUCCESS	Image Base: 0x7ffab2780000, Image Size: 0x17
chrome.exe	5528	Load Image	C:\Windows\System32\shlwapi.dll	SUCCESS	Image Base: 0x7ffab4a90000, Image Size: 0x67
chrome.exe	5528	Load Image	C:\Windows\System32\kernel.appcore.dll	SUCCESS	Image Base: 0x7ffab0ec0000, Image Size: 0x1b
chrome.exe	5528	Load Image	C:\Windows\System32\uxtheme.dll	SUCCESS	Image Base: 0x7ffaaf3d0000, Image Size: 0xaaf
chrome.exe	5528	Load Image	C:\Windows\System32\clbcatq.dll	SUCCESS	Image Base: 0x7ffab3b00000, Image Size: 0xae
chrome.exe	5528	Load Image	C:\Windows\System32\sechost.dll	SUCCESS	Image Base: 0x7ffab40a0000, Image Size: 0xaa
chrome.exe	5528	Thread Create		SUCCESS	Thread ID: 6372
chrome.exe	5528	Thread Create		SUCCESS	Thread ID: 4732
chrome.exe	5528	Thread Create		SUCCESS	Thread ID: 6996
elevation_service.exe	4924	Process Start		SUCCESS	Parent PID: 828, Command line: "C:\Program Fil
elevation_service.exe	4924	Thread Create		SUCCESS	Thread ID: 5864
elevation_service.exe	4924	Load Image	C:\Program Files\Google\Chrome\Application\149.0.7827.155\elevation_service.exe	SUCCESS	Image Base: 0x7ff727fd0000, Image Size: 0x3ec
elevation_service.exe	4924	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x7ffab4ee0000, Image Size: 0x26
elevation_service.exe	4924	Load Image	C:\Windows\System32\kernel32.dll	SUCCESS	Image Base: 0x7ffab3c40000, Image Size: 0xc9f
elevation_service.exe	4924	Load Image	C:\Windows\System32\KernelBase.dll	SUCCESS	Image Base: 0x7ffaaf2b0000, Image Size: 0xa0f

Both elevation_service.exe instances are parented by services.exe, the standard Windows COM activation parent, and they're launched in response to CoCreateInstance calls against the IElevator COM interface, the same API surface Chrome itself uses to decrypt the AppBound-protected key in Local State. The paths touched confirm the purpose directly: a RegOpenKey against HKCR\CLSID\{00020424-0000-0000-C000-000000000046}\Elevation, followed by reads of DPAPI.dll from both the Chrome application directory and C:\Windows\System32. The double invocation lines up with two separate AppBound decryption operations: one for the cookies key, one for the password key.

Wallet targeting runs independently of the browser theft path, probing a fixed list of directories (Bitcoin, Coinomi, Electrum, Exodus, WalletWasabi) plus the Chrome extension storage for the Google Wallet extension ID nmmhkkegccagdldgiimedpiccmgmieda. These are presence probes rather than reads, as the implant opens the directory to check whether it exists and would only proceed to read files if something were actually installed there, and on this particular VM none of the wallets were present, so the probes came back empty. Finally, Telegram theft is a single CreateFile against ... \Telegram Desktop\tdata, which is enough on its own to hand an operator a live, already-authenticated Telegram session without touching 2FA. Edge gets the same treatment as Chrome, profile-for-profile, against its near-identical Chromium layout.

Discovery

Host fingerprinting pulls the obvious set of registry values:

- ComputerName
- MachineGuid and its WOW6432Node mirror
- The user profile list

File and directory discovery under Chrome and Edge is exhaustive: cache, code cache, extensions, extension state, session storage, local storage, IndexedDB, sessions, all enumerated, totaling 1,775 distinct Chrome-related operation/path pairs in this run alone. Software discovery walks more than 25 top-level vendor keys under HKLM\SOFTWARE, building a real-time inventory of installed credential-bearing software before the implant decides what's worth exfiltrating, and the repeated VMware checks noted earlier double as ongoing security-software discovery rather than a one-time startup gate.

Command and Control

The most operationally useful thing the dynamic analysis revealed is exactly how the fallback resolution works. Two HTTP probes to a Cloudflare-hosted IP show up sandwiched between failed and successful C2 connection attempts:

Press enter or click to view image in full size

Process Name	PID	Operation	Path	Result	Detail
VirtualAr.exe	1588	TCP Reconnect	NaughtyKoala:58787 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Reconnect	NaughtyKoala:58787 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Reconnect	NaughtyKoala:58787 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Reconnect	NaughtyKoala:58787 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Disconnect	NaughtyKoala:58787 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Connect	NaughtyKoala:58791 -> 172.66.147.243:http	SUCCESS	Length: 0, mss: 1360, sackopt: 1, tsopt: 0, wsop
VirtualAr.exe	1588	TCP Send	NaughtyKoala:58791 -> 172.66.147.243:http	SUCCESS	Length: 51, starttime: 7234755, endtime: 723475
VirtualAr.exe	1588	TCP TCPCopy	NaughtyKoala:58791 -> 172.66.147.243:http	SUCCESS	Length: 273, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Receive	NaughtyKoala:58791 -> 172.66.147.243:http	SUCCESS	Length: 273, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Disconnect	NaughtyKoala:58791 -> 172.66.147.243:http	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Reconnect	NaughtyKoala:58792 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Reconnect	NaughtyKoala:58792 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Reconnect	NaughtyKoala:58792 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Reconnect	NaughtyKoala:58792 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Disconnect	NaughtyKoala:58792 -> 238-113-163-66.clients.gghost.com:3333	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Connect	NaughtyKoala:58793 -> 172.66.147.243:http	SUCCESS	Length: 0, mss: 1360, sackopt: 1, tsopt: 0, wsop
VirtualAr.exe	1588	TCP Send	NaughtyKoala:58793 -> 172.66.147.243:http	SUCCESS	Length: 51, starttime: 7237059, endtime: 723706
VirtualAr.exe	1588	TCP TCPCopy	NaughtyKoala:58793 -> 172.66.147.243:http	SUCCESS	Length: 273, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Receive	NaughtyKoala:58793 -> 172.66.147.243:http	SUCCESS	Length: 273, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Disconnect	NaughtyKoala:58793 -> 172.66.147.243:http	SUCCESS	Length: 0, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Connect	NaughtyKoala:58794 -> 193.24.123.38:3333	SUCCESS	Length: 0, mss: 1360, sackopt: 1, tsopt: 0, wsop
VirtualAr.exe	1588	TCP TCPCopy	NaughtyKoala:58794 -> 193.24.123.38:3333	SUCCESS	Length: 48, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Receive	NaughtyKoala:58794 -> 193.24.123.38:3333	SUCCESS	Length: 48, seqnum: 0, connid: 0
VirtualAr.exe	1588	TCP Send	NaughtyKoala:58794 -> 193.24.123.38:3333	SUCCESS	Length: 48, starttime: 7237585, endtime: 723759
VirtualAr.exe	1588	TCP Send	NaughtyKoala:58794 -> 193.24.123.38:3333	SUCCESS	Length: 38, starttime: 7237585, endtime: 723759
VirtualAr.exe	1588	TCP Send	NaughtyKoala:58794 -> 193.24.123.38:3333	SUCCESS	Length: 621, starttime: 7237585, endtime: 72376

This is the implant's dead-drop resolver lookup, translating one of the five yoda-*.sbs/.site/.online fallback domains the static config exposed into a usable IP at runtime. The full resolution sequence observed in this run was:

- Try the primary IP from the embedded config (66.163.113[.]238:3333)
- Fails after a 20-second retry timeout
- HTTP probe the Cloudflare resolver
- Try the primary IP again (fails again, same timeout)

- Probe the resolver again (identical response)
- Connect to the resolver-supplied fallback IP (193.24.123[.]38:3333, succeeds)
- And finally open three parallel data sessions to 193.24.123[.]38:3334

The primary IP's reverse DNS, 238-113-163-66.clients.gthost[.]com, is the PTR record for 66.163.113[.]238, the exact same primary address recovered statically from the encrypted config in the previous section, which is a clean confirmation that the static config-extraction pipeline produces operationally accurate results.

Once the control connection to 3333/tcp is established, it's followed within about a second by three parallel data-session connecting to 3334/tcp, matching the {"p":3333,"dp":3334} port pair recovered statically and giving the operator a control plane that stays free to issue commands while bulk data moves concurrently across independent sockets.

Collection and Exfiltration

Everything the implant gathers gets staged into the Mayanex drop directory before it goes anywhere. These paths map exactly onto the config keys recovered statically: dd to C:\ProgramData\Mayanex\, kf to Kapuq, c to a Nata subdirectory holding SHA-1-named files (one per collected artifact, functioning as a local de-duplication cache that never leaks what's been stolen through its filenames), and ab to a naijaco subdirectory holding the autostart record. Once staged, three parallel data sessions start exchanging traffic almost immediately, which matches the statically-observed design. The data plane is built to carry multiple concurrent transfers (browser data, wallet data, Telegram data all in flight together) rather than serializing everything over one socket.

End to end, this second infection chain goes from first execution to live, multi-session C2 with credential collection already underway in 2 minutes 16 seconds, entirely automated, with no human interaction anywhere in the sequence.

Dissecting SnappyClient's C2 Protocol

The dynamic analysis confirmed that SnappyClient reaches C2, registers the victim host, receives data-session setup commands, and gets actively tasked to collect browser data, crypto-wallet data, Telegram data, and seed-phrase material. However, I really wanted to understand how this custom TCP protocol worked, so in this section I describe how I pulled that protocol apart packet by packet.

As we've seen in the previous section, SnappyClient opens one control connection and three data-session connections:

Press enter or click to view image in full size

Role	Flow	Port
Control session	10.0.30.2:58794 -> 193.24.123[.]38:3333	3333/tcp
Data session	10.0.30.2:58796 -> 193.24.123[.]38:3334	3334/tcp
Data session	10.0.30.2:58795 -> 193.24.123[.]38:3334	3334/tcp
Data session	10.0.30.2:58798 -> 193.24.123[.]38:3334	3334/tcp

The protocol itself is a custom encrypted TCP design layered over plain sockets, and both the control and data channels share the same handshake shape. For the control channel:

1. The client connects to 3333/tcp
2. The server immediately sends 48 raw bytes (a 32-byte ChaCha20-Poly1305 key, a 12-byte nonce, and a 4-byte session ID)
3. The client replies with an encrypted key-confirmation packet
4. Normal message exchange begins

Data sessions follow the same pattern with one addition: after connecting to 3334/tcp and receiving the same 48-byte key/nonce/session packet, the client first sends a 12-byte cleartext preamble identifying the control session ID, the new data session ID, and a flags byte (always set to 1 in my analysis), and only then sends the encrypted key-confirmation packet before normal encrypted messages start.

Once the handshake completes, every message follows the same two-part shape. First SnappyClient transmits an encrypted header concatenated with its Poly1305 tag:

```
encrypted header || Poly1305 tag
```

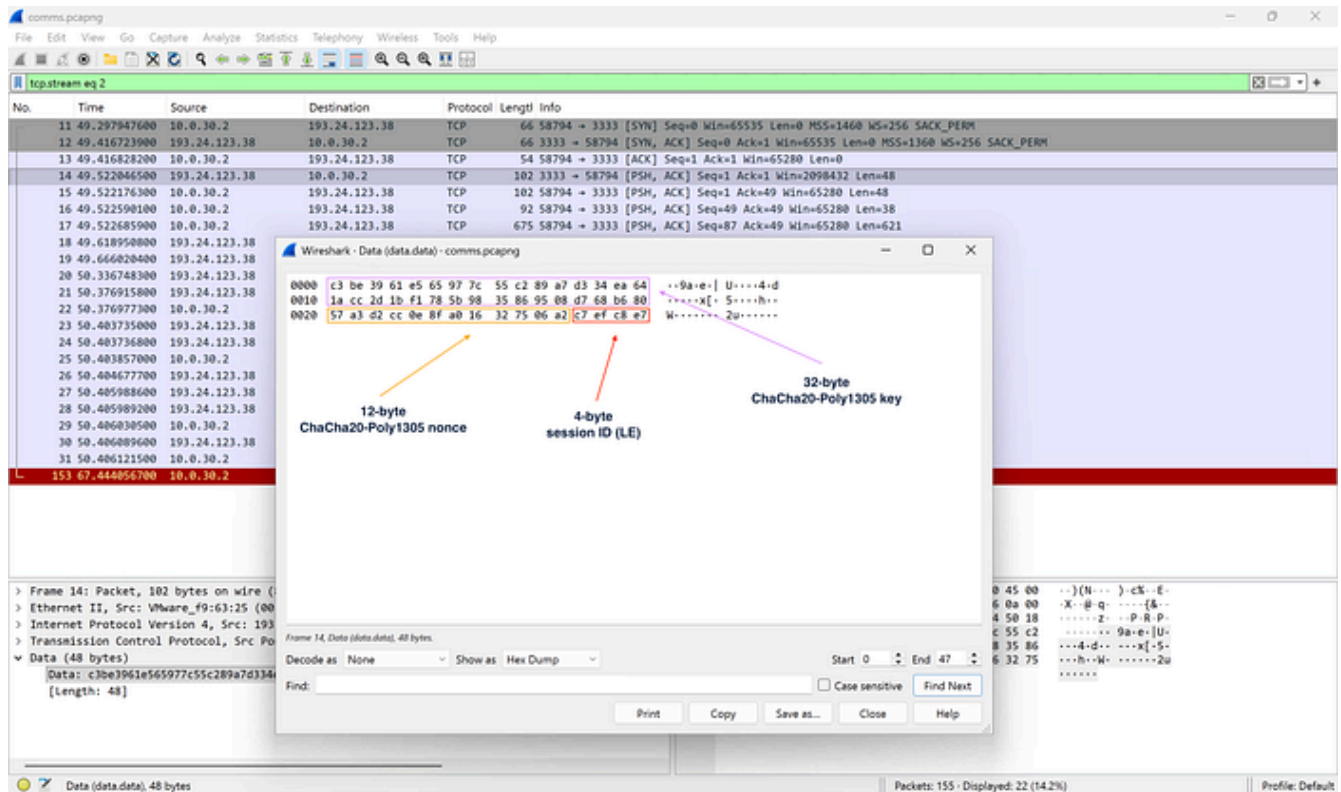
Then, the body itself is structured as a 16-bit little-endian encrypted size field, a single tag-flag byte, the encrypted Snappy-compressed JSON payload, and a trailing Poly1305 tag. Large bodies can be chunked, and the tag-flag byte tells you which kind of chunk you're looking at: 0 for an intermediate encrypted chunk, 1 for the final chunk carrying the Poly1305 tag:

```
uint16_le encrypted body size
uint8      tag flag
bytes      encrypted Snappy-compressed JSON document
bytes      poly1305 tag
```

As you can see in the following screenshot, after the TCP three-way handshake the very first thing the C2 sends is 48 bytes of raw session setup material, split as: 32 bytes of ChaCha20-

Poly1305 key, 12 bytes of nonce, and 4 bytes of little-endian control session ID:

Press enter or click to view image in full size



In plain terms, that one packet is the C2 telling the implant: use this key and this nonce for the rest of this TCP session, and refer to this session as 3888705479 going forward.

The malware's reply is a 48-byte key-confirmation packet: the same 32-byte session key, re-encrypted under itself with the session key and nonce via ChaCha20-Poly1305, producing 32 bytes of ciphertext plus a 16-byte tag. So the opening exchange is just:

```
C2 -> malware:  key[32] || nonce[12] || session id[4]
```

```
malware -> C2: ChaCha20-Poly1305(key, nonce).encrypt(key)
```

After that, real C2 traffic starts. Packet 16 from the malware to the C2 server is a 38-byte encrypted message header. The header data contains a 22-byte ciphertext and a 16-byte trailing tag. Decrypting it with the session key and nonce from the C2's first packet gives the following raw command: ffff49499bf8010000006502000000000000000000000000.

Press enter or click to view image in full size

Input Content: 9c23ebe88639be45223af30fbb21a3f74cfd4170906

Algorithm: ChaCha20Poly1305 | Charset: UTF-8

Password Format: hex | Password: c3be3961e565977c55c289a (password length is 32 bytes (256 bits))

Nonce Format: hex | Nonce: 57a3d2cc0e8fa016327506a (nonce length is 12 bytes (96 bits))

Input Format: hex | Output Format: hex

Tag Format: hex | Authenticate Tag: a6693f91f37a3c6b46ae009a (authenticate tag length is 16 bytes (128 bits))

Buttons: Open, Decrypt, Exchange, Clear

Output Result(Click to Copy): fff49499bf801000000650200000000000000000000

That 22-byte plaintext header can be parsed like this:

ff ff	command_id
49 49 9b f8	message_id
01 00 00 00	unknown / constant
65 02 00 00	plaintext body length
00 00 00 00 00 00 00 00	zero marker

Which decodes to:

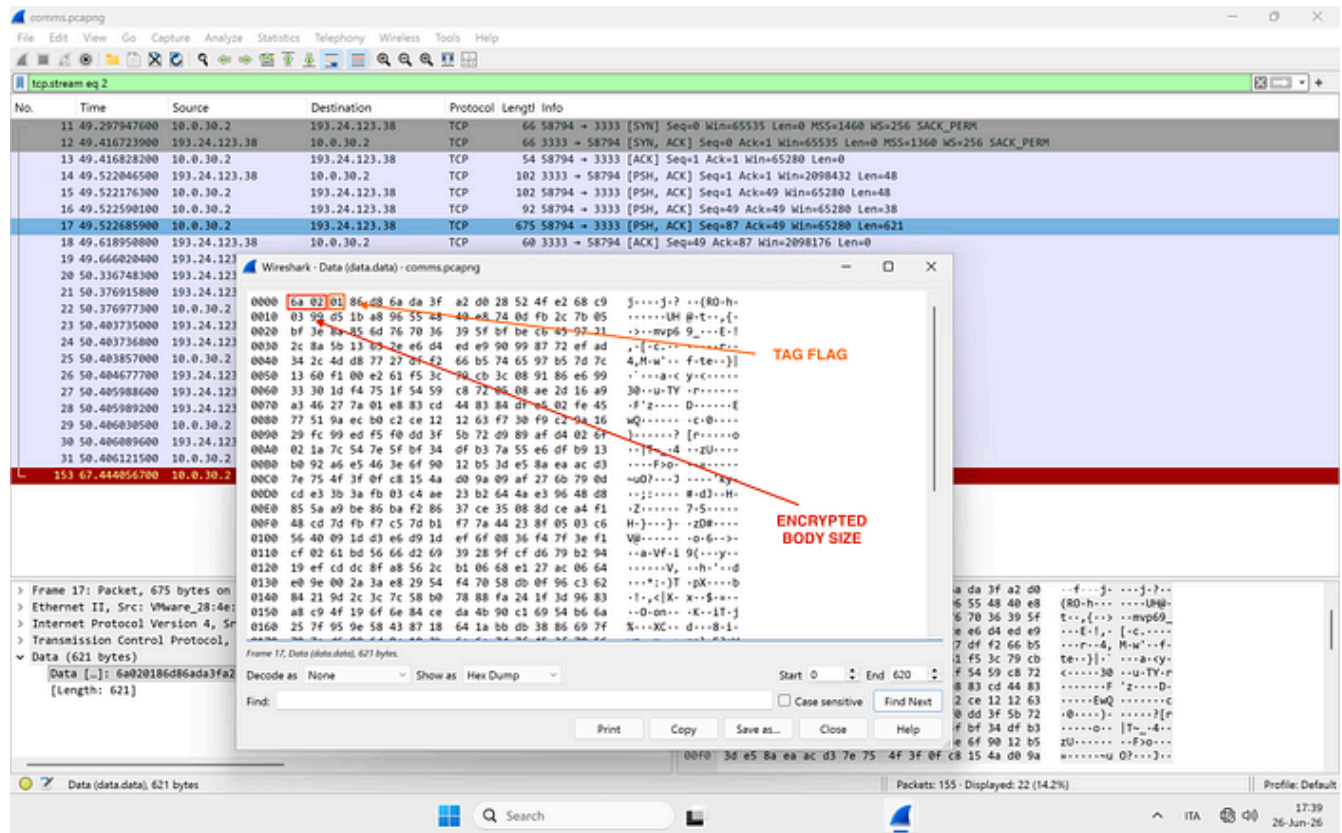
command_id	= 0xFFFF -> BEACON REGISTRATION / CHECK-IN
message_id	= 4170926409
unknown	= 1
message_body_length	= 613 bytes
zero marker	= 8 null bytes

Meaning that this is the header for a client-to-C2 registration message.

The following packet (n. 17), is the encrypted Snappy-compressed registration body. Message 17's own body starts with a 3-byte prefix, 6a 02 01, decoding to an encrypted body size of 0x026a (618 bytes) and a tag flag of 1, meaning this is a final chunk, with the trailing 16 bytes

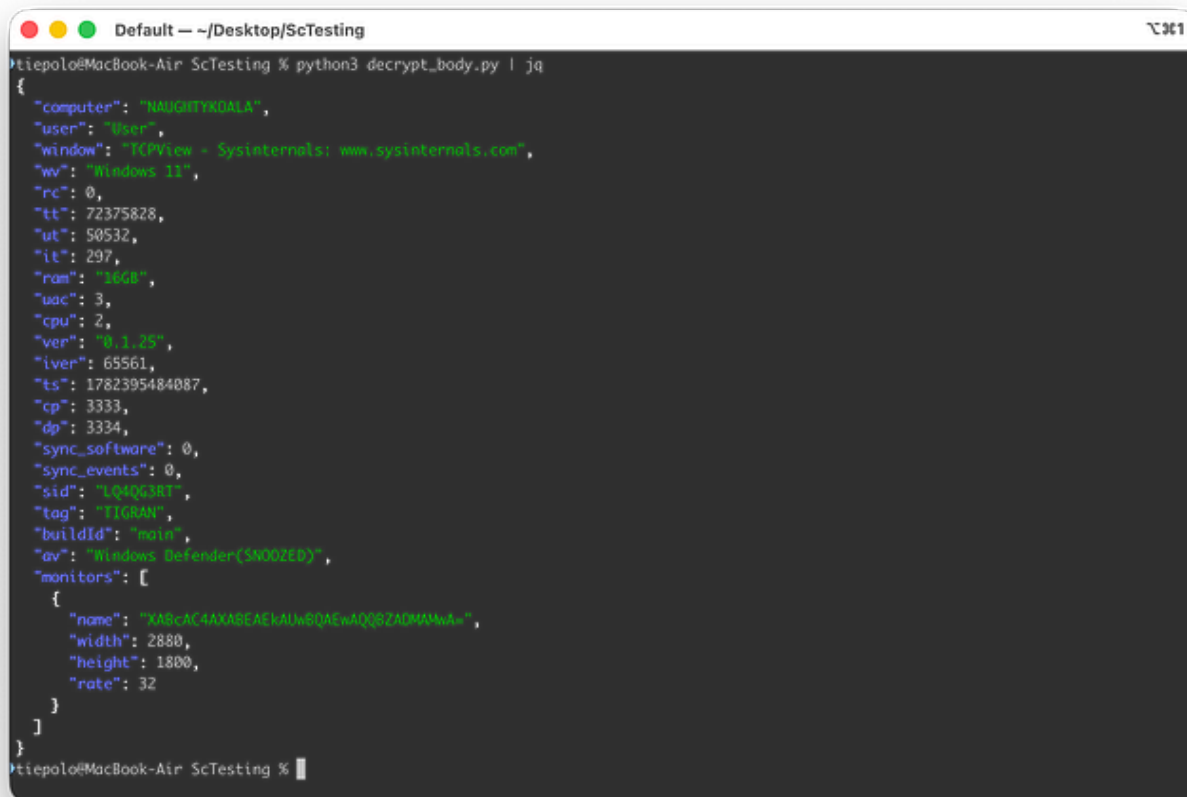
being the Poly1305 tag and the preceding 602 bytes being the actual ciphertext, starting at offset 4:

Press enter or click to view image in full size



Decrypting that ciphertext with the same session key and nonce, then Snappy-decompressing the result, produces the full registration JSON the implant sends on first contact:

Press enter or click to view image in full size

A terminal window titled "Default — ~/Desktop/ScTesting" on a Mac. The prompt is "tiempolo@MacBook-Air ScTesting %". The command entered is "python3 decrypt_body.py | jq". The output is a JSON object representing system information. The JSON is formatted with syntax highlighting: strings are green, numbers are yellow, and booleans are blue. The output shows fields like "computer", "user", "window", "wv", "rc", "tt", "ut", "it", "ram", "uac", "cpu", "ver", "lver", "ts", "cp", "dp", "sync_software", "sync_events", "sid", "tag", "buildId", "av", and "monitors". The "monitors" array contains one object with fields "name", "width", "height", and "rate".

```
tiempolo@MacBook-Air ScTesting % python3 decrypt_body.py | jq
{
  "computer": "HAUGHTYREDALA",
  "user": "User",
  "window": "TCPView - Sysinternals: www.sysinternals.com",
  "wv": "Windows 11",
  "rc": 0,
  "tt": 72375828,
  "ut": 50532,
  "it": 297,
  "ram": "38GB",
  "uac": 3,
  "cpu": 2,
  "ver": "0.1.25",
  "lver": 65561,
  "ts": 1782395484087,
  "cp": 3333,
  "dp": 3334,
  "sync_software": 0,
  "sync_events": 0,
  "sid": "LQ4Q63RT",
  "tag": "TIGRAN",
  "buildId": "main",
  "av": "Windows Defender(SNOOZER)",
  "monitors": [
    {
      "name": "XABcAC4AXABEAEkAUwBQREwAQ6SZADMANwA=",
      "width": 2880,
      "height": 1800,
      "rate": 32
    }
  ]
}
```

The next TCP packets show the malware doing exactly what its architecture implies: using the control channel to orchestrate, and pushing the actual work onto data sessions. The server replies first with 0xBABA, an empty acknowledgment, then issues three 0xDDDD (open data session) commands in sequence, each carrying a new session ID and a shared frameId of 3005772351:

Press enter or click to view image in full size

Direction	Command	Meaning	Key fields
Server → client	0xBABA	Empty acknowledgment-like response	{}
Server → client	0xDDDD	Open data session	sid=3736169770 , frameId=3005772351
Server → client	0xDDDD	Open data session	sid=4030284061 , frameId=3005772351
Server → client	0xDDDD	Open data session	sid=3404666342 , frameId=3005772351

All three reference the same control session ID, 3888705479. The C2 never sends large tasking payloads over the control connection itself, as it uses control purely to spin up data sessions, then moves all the substantive configuration onto data port 3334.

The first data session carried the browser-theft configuration. It targets a long list of Chromium-family browsers (360Browser, 360ChromeX, 360Extreme, Opera, Chrome, CocCoc, Edge, QQBrowser, Slimjet, UCBrowser, Vivaldi, Brave) and two Mozilla-family browsers (Firefox, Waterfox), each with an explicit data-path string, alongside a set of targeted browser-extension wallets (Coinbase, Metamask, OKXWallet, Phantom, TronLink, TrustWallet). Brave gets one extra file-based rule beyond the standard set, targeting encrypted_mnemonic inside files matching (\Preferences)(?=\$) under the BraveWallet name.

Press enter or click to view image in full size

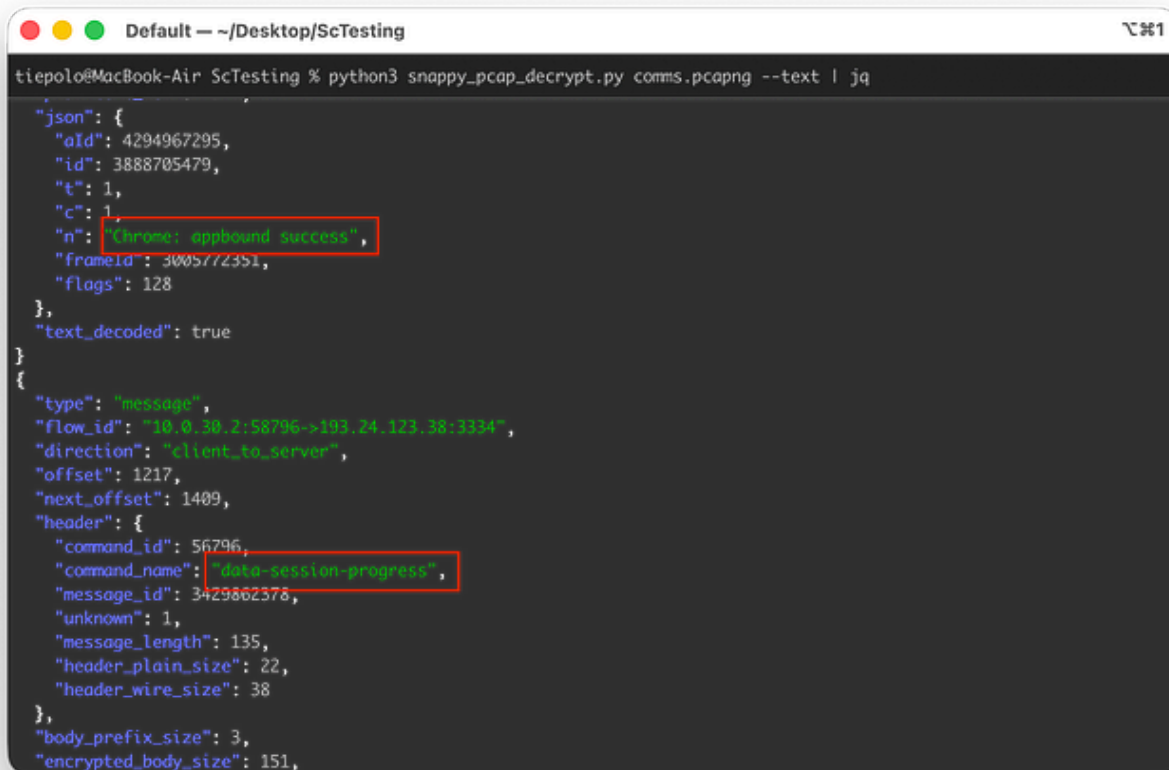
```
Default — ~/Desktop/ScTesting
```

```
tiepolo@MacBook-Air ScTesting % python3 snappy_pcap_decrypt.py comms.pcapng --text | jq
```

```
"action": 3,  
"data": {  
  "browsers": [  
    {  
      "u8name": "360Browser",  
      "name": "360Browser",  
      "process_name": "360se.exe",  
      "data_path": "%AppData%\360se6\User Data",  
      "profile_regex": "(((Default|Profile[\\w _]*)$)",  
      "logins_file": "(\\\\\\\\Login Data\\\\\\\\Login Data For Account1\\\\\\\\Web Data)$",  
      "cookies_file": "(\\\\\\\\Cookies$)",  
      "engine": "Chromium",  
      "ext_regex": "(\\\\\\\\\\hnfanknocfeofbddgcijnmhfnkdnaad\\\\\\\\nkbihfbeogaeaoehlefnkodbefgpgknn\\\\\\\\mcohilncbfahbm  
djkbpemcciiolgce\\\\\\\\bnafnaeiameimhlpngjaiohhpkkoljpa\\\\\\\\ibnejdfjmknpcnlpebklnkoenihofeci\\\\\\\\egjidjbpglichdrondcb  
nbeeppgdph)(?=$)",  
      "extensions": [  
        {  
          "name": "Coinbase",  
          "id": "\hnfanknocfeofbddgcijnmhfnkdnaad"  
        },  
        {  
          "name": "Metamask",  
          "id": "nkbihfbeogaeaoehlefnkodbefgpgknn"  
        },  
        {  
          "name": "OKXWallet",  
          "id": "mcohilncbfahbm djkbpemcciiolgce"  
        }  
      ]  
    }  
  ]  
}
```

The bot reported its own progress back over the same data stream in plain status strings, which confirms the AppBound handling I'd seen in the credential-access trace wasn't theoretical; the C2 explicitly tasked it and the implant attempted it live:

Press enter or click to view image in full size



```
tiepolo@MacBook-Air ScTesting % python3 snappy_pcap_decrypt.py comms.pcapng --text | jq
{
  "json": {
    "oId": 4294967295,
    "id": 3888705479,
    "t": 1,
    "c": 1,
    "n": "Chrome: opphaund success",
    "frameId": 3005772351,
    "flags": 128
  },
  "text_decoded": true
}
{
  "type": "message",
  "flow_id": "10.0.30.2:58796->193.24.123.38:3334",
  "direction": "client_to_server",
  "offset": 1217,
  "next_offset": 1409,
  "header": {
    "command_id": 56796,
    "command_name": "data-session-progress",
    "message_id": 3429862378,
    "unknown": 1,
    "message_length": 135,
    "header_plain_size": 22,
    "header_wire_size": 38
  },
  "body_prefix_size": 3,
  "encrypted_body_size": 151,
}
```

The second data session carried task name exSeeds, with options {"min":1,"max":20000,"flags":3,"job":1} and three filters:

- An include rule for *.txt files
- A match rule applying a 12-word BIP-39-style mnemonic wordlist pattern
- A size or bound limit of 20000

I'm intentionally not reproducing the full BIP-39-style regular expression here since it's both very large and not the point; what matters operationally is that the C2 explicitly instructed SnappyClient to search text files for wallet seed phrases, and the client's progress messages during this task (C:\, C:\Program Files\7-Zip, C:\...\7-Zip\Lang) confirm the file-scanning capability was genuinely activated:

```
{
  "type": "message",
  "flow_id": "10.0.30.2:58795->193.24.123[.]38:3334",
  "direction": "client_to_server",
  "offset": 572,
  "next_offset": 775,
  "header": {
```

```
    "command_id": 56796,  
    "command_name": "data-session-progress",  
    "message_id": 3702421930  
  },  
  "json": {  
    "aId": 408945804,  
    "id": 3888705479,  
    "t": 0,  
    "c": 0,  
    "n": "C:\\\\...\\7-Zip\\Lang"  
  }  
}
```

The third data session carried command 0xDACC, targeting software wallets and Telegram directly:

Press enter or click to view image in full size

Target	Search path	Include	Exclude
Atomic	%AppData%\atomic	*	*cache*
BitcoinCore	%AppData%\Bitcoin \\\ %LocalAppData%\Bitcoin\	*wallet.dat	
Coinomi	%LocalAppData% \Coinomi\Coinomi\wallets	*	
Electrum	%AppData% \Electrum\wallets	*	
Exodus	%AppData% \Exodus\exodus.wallet	*	
LedgerLive	%AppData%\Ledger Live	*	*cache*
Telegram	%AppData%\Telegram Desktop\tdata	^(?!.*(cache ebwebview)).*\$	
TrezorSuite	%AppData%@trezor\suite- desktop	*	*cache*
Wasabi	%AppData% \WalletWasabi\Client\Wallet s	*	

The implant's response to this particular task was simply nothing found, since none of these wallet applications were installed on the analysis host, but the task structure itself confirms the operator-configurable targeting.

In summary, the pattern across all three data sessions is the same: the C2 is using SnappyClient as a modular, task-driven credential and wallet collector. Browser paths, wallet extension IDs, file-search filters, and seed-phrase rules all arrive from the server side, which means the operator can update the targeted data without ever touching the implant binary itself, and it also means static analysis alone will only ever show you the shape of what's possible, while the live capture shows you what a specific operator actually chose to task on a specific run.

No HVNC, SOCKS proxy, reverse shell, screenshot, or payload-download command appeared in this particular execution. Those capabilities are all statically present in the binary, but in this specific run the threat actor simply chose not to deploy them.

Closing Thoughts

What makes this campaign effective isn't any single payload or technique, but a handful of design choices that recur across both chains. The most consequential is that the malicious code never ran as a process of its own. Radmin VPN, Opera, dBpoweramp, the signed Qihoo 360 component, Crisp: every executable in this campaign was a legitimate, validly signed binary. The malicious logic ran only inside those trusted processes, loaded through DLL side-loading and process hollowing, and the implant touched disk only as randomly-named .tmp files deleted moments after the hollow completed. Signature checks, reputation scoring, and ordinary AV all key off what a file *is*, and here every file is a signed application from a reputable vendor.

The network layer follows the same logic. SnappyClient skips HTTPS entirely and runs ChaCha20-Poly1305 over Snappy-compressed frames directly on TCP ports 3333/3334, so there's no TLS handshake to fingerprint, no certificate or SNI to check, no URL or user-agent to profile. None of the controls designed to inspect web traffic apply, because the traffic is not HTTP-based. Still, high-entropy binary traffic on a non-standard port with no TLS handshake is anomalous in most enterprise environments, and the protocol's structure yields usable indicators: the fixed 48-byte server-first handshake, and a single control connection on 3333 alongside multiple concurrent data connections on 3334. In practice, abandoning HTTPS shifts the detection surface from certificate and SNI analysis toward identifying encrypted, non-TLS traffic on ports that are not normally inspected.

It's also worth noting how cleanly this campaign lines up with what Zscaler ThreatLabz independently documented for SnappyClient — delivery via HijackLoader, targeting of German-speaking users, a focus on cryptocurrency wallets, and configuration-driven tasking suited to long-term access rather than one-pass theft. Zscaler also note code-level overlaps between SnappyClient and HijackLoader and suggest the two may share developers.

Running a stealer and a RAT together like this is characteristic of how the access economy now works. Infostealers were the most common initial-access vector reported in 2025, precisely because the credentials, cookies, and session tokens they harvest are what feed the initial-access-broker (IAB) market that ransomware crews buy from. That progression — stealer infection, then brokered access, then ransomware — is normally split across separate actors and separated in time. This campaign performs the first two stages at once: Vidar does the stealing, and SnappyClient is the persistent, sellable foothold a broker would otherwise

have to establish. A single Radmin VPN sideload therefore produces both the stolen data and the durable access in one intrusion. The practical consequence for defenders is that an infostealer detection should not be treated as self-contained, as the same loader may have delivered a second, persistent payload in parallel, and the host should be treated as a potential remote-access compromise until that is excluded.

That closes out Operation Turb00! What started as one unremarkable sample turned into a multi-stage infection chain from a single HijackLoader delivery. Full IOCs, YARA, and Sigma rules are in the accompanying [repository](#).