Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 3, Issue 8, 2025

UNVEILING PYTHON-BASED KEYLOGGER MALWARE: BEHAVIORAL ANALYSIS, ARCHITECTURE, AND MITIGATION STRATEGIES

Asad Iqbal[*1], Malik Muhammad Huzaifa[2], Urooba Sumbal[3], Ahmed Sajid Butt[4], Muhammad Zunnurain Hussain[5], Muhammad Zulkifl Hasan[6]

[*1]*Shark Innovation Labs - Al Sharqi, Dept. of Computer Sciences - National College of Business Administration and Economics (NCBA&E), Lahore*
[2]*Graduate of Bachelor's in Computer Science, Bahria University, Lahore, Pakistan*
[3]*Graduate of Bachelor's in Computer Science, Bahria University, Lahore, Pakistan*
[4]*Graduate of Bachelor's in Computer Science, Bahria University, Lahore, Pakistan*
[5]*Assistant Professor, Department of Computer Sciences, Bahria University, Lahore, Pakistan*
[6]*Principal Lecturer, Faculty of Information Technology, Department of Computer Science, University of Central Punjab, Lahore, Pakistan*

[*1]theasadiqbal.official@gmail.com, [2]malik74pk@gmail.com, [3]uroobasumbal@gmail.com, [4]ahmedbutt905@gmail.com, [5]zunnurain.bulc@bahria.edu.pk, [6]zulkifl.hasan@ucp.edu.pk

DOI: https://doi.org/10.5281/zenodo.16836092

Keywords
Keylogger, Python Malware, Remote Access Trojan, Email C2, Behavioral Analysis, Malware Detection.

Article History
Received on 11 May 2025
Accepted on 17 July 2025
Published on 13 August 2025

Corresponding Author: *
Asad Iqbal

Abstract

The rising sophistication of Python-based malware has made simple scripting languages potent tools for executing surveillance and exfiltration attacks. This paper analyzes a fully operational Python-based Remote Access Tool (RAT) that leverages keylogging, clipboard monitoring, screenshot capture, email-based command-and-control, and self-destruction techniques. Through code-level dissection and architectural modeling, the study reveals the malware's internal mechanisms and behavior. The paper also proposes detection methods and defensive strategies suitable for individuals and organizations. This research aims to bridge the gap between cybersecurity awareness and technical comprehension, promoting proactive defense against lightweight but dangerous malware.

INTRODUCTION

The growing use of lightweight scripting languages in offensive cyber operations has amplified the risks posed by non-obfuscated, modular malware. Python, valued for its accessibility, cross-platform compatibility, and extensive standard libraries, has been exploited by attackers to develop surveillance tools including keyloggers, clipboard monitors, screen and webcam recorders, and remote access scripts. Originally intended for education and scientific computing, Python's openness and power now enable even novice threat actors to build modular, multi-threaded malware in just a few hundred lines of code. This research analyzes a live Python-based malware specimen exhibiting typical Remote Access Trojan (RAT) capabilities. It logs keystrokes, monitors clipboard content, captures screenshots and webcam

images, and communicates covertly via Gmail-based command-and-control (C2). Persistence is achieved through Windows startup shortcuts, while a built-in self-destruction feature wipes its presence on demand. Thanks to its multi-threaded design, the malware runs these tasks in parallel with minimal resource usage, enhancing both stealth and resilience.

1.1 Research Focus

This study provides an in-depth behavioral and architectural analysis of Python-based keylogger malware. Beyond simple detection, it reveals the operational logic and demonstrates how publicly available code can undermine privacy, enterprise security, and digital trust. By reconstructing the malware lifecycle from initialization through execution to self-deletion, the paper highlights its dependencies, communication flows, and system-level manipulations. The ultimate aim is to raise awareness among cybersecurity professionals, educators, and students about the ease of creating and deploying such malware, while outlining detection strategies and defensive frameworks for early response.

1.2 Key Aspects

This malware illustrates how low-complexity Python scripts can evolve into sophisticated threats without administrator privileges or kernel-level exploits. It uses trusted Windows tools and standard Gmail accounts to remain persistent and evade detection. Commands are delivered over IMAP and results exfiltrated via SMTP, supporting actions like keystroke logging, clipboard capture, screenshots, webcam snapshots, and self-wiping. These commands, parsed from simple keywords, are executed concurrently through multi-threading without blocking other modules. Additionally, the script marks artifacts as hidden to further evade notice. Its lack of obfuscation or encryption makes it a powerful educational case of how transparent code can produce highly functional malware.

1.3 System Overview

Architecturally, the malware comprises five modules: keylogger, clipboard monitor, command interpreter, surveillance components (screenshot and webcam), and persistence/self-deletion routines. At runtime, these modules are launched as parallel threads. The keylogger appends user input to a hidden log, while the clipboard monitor polls and merges clipboard content. The command interpreter continuously polls a Gmail inbox for instructions, which are dispatched to relevant modules. Surveillance features leverage libraries such as pyautogui and cv2 to interface with system I/O devices. Persistence is maintained via a Windows Startup shortcut, and a self-destruction command triggers a batch script that deletes all traces and terminates running processes.

1.4 Methodology

This research applied both static and dynamic analysis. Static review mapped logical flow, module separation, dependencies, and command parsing. Each module was evaluated for its role in data exfiltration and surveillance. Dynamic testing in a sandboxed Windows VM examined runtime behaviors, including file changes, process creation, and network activity with Gmail's IMAP/SMTP servers. Logging tools, process monitors, and behavior-based antivirus software captured alerts and system impacts. Visual diagrams and data flows were developed to support clarity, while the MITRE ATT&CK framework mapped observed behaviors to recognized attack vectors, ensuring the analysis maintained practical relevance. Together, this methodology enabled a well-rounded understanding of the malware's threat profile and informed the detection strategies proposed later in the paper.

2. Literature Review

2.1 Overview

The growing sophistication of cyber threats, together with the accessibility of languages like Python, has expanded the threat surface for lightweight script-based malware. While machine learning (ML), deep learning (DL), and graph-based detection models have advanced, they often neglect minimal Python-based threats that leverage legitimate services like email protocols for covert communication. Traditional intrusion detection systems still struggle to identify evasive, modular code that evades dynamic inspection through anti-analysis or anti-instrumentation techniques, as noted by Gaber et al. [3]. Bilot et al. [4] similarly highlight how graph-based representation learning often prioritizes structured binaries, leaving script-level threats underexplored. These lightweight

Python-based keyloggers maintain a minimal footprint, relying on system calls and built-in libraries to remain stealthy. As Bensaoud et al. [2] emphasizes, fileless malware and SMTP-based payloads continue to rise, yet detection capabilities often stop at surface-level signatures. Accordingly, a deeper understanding of the code architecture and behavior of such minimal Python malware remains highly relevant.

2.2 Comparative Literature Review

Recent studies emphasize both the promise and limitations of modern AI-based detection. Gaber et al. [3] and Gopinath and Sethuraman [1] underline that while DL models show impressive accuracy, their poor interpretability creates challenges for defending against stealthy, user-mode threats. Bensaoud et al. [2] notes that fileless keyloggers often evade static analysis entirely by running exclusively in memory. Bilot et al. [4] introduces graph neural networks for malware detection but concedes that their focus remains largely on binaries, not interpreted scripts. Alomari et al. [20] and Djenna et al. [21] present correlation-based and dynamic deep learning approaches, but they still fall short against simple, non-obfuscated Python code.

In hardware-focused methods, Chenet et al. [7] review CPU event-based profiling, but specialized hardware limits deployment in mainstream systems. Baker del Aguila et al. [8] argue for lightweight ML classifiers suitable for IoT and edge devices, though these may underperform against evasive keyloggers. Keylogger-specific research by Ayo et al. [17] and Singh et al. [19] explores fuzzy inference and anomaly-based models but generally lacks validation in adversarial or stealth conditions. Bhat and Namratha [9] demonstrate a cybersecurity testbed with keylogger visualization, while Khalid et al. [18] discuss memory forensics for detecting fileless malware, which could extend to Python-based keyloggers.

Broader surveys by Ferdous et al. [5], Aryal et al. [6], and Vasani et al. [12] explore trends in ransomware, adversarial ML, and multi-stage detection pipelines. These authors emphasize that script-based malware with adversarial evasion patterns remains a formidable challenge for traditional models.

2.3 Research Gaps and Our Contributions

Despite this progress, key gaps remain. Most detection frameworks prioritize compiled malware binaries (e.g., PE, APK), underrepresenting interpreted languages like Python. Second, user-space memory analysis for keylogger behavior is rare, as most rely on static signatures or process monitoring alone. Third, many frameworks are evaluated only on synthetic or simulated data, not under realistic, stealth-based conditions. Finally, the ethical framing of academic keylogger analysis is often incomplete, lacking transparent testing methodologies.

This study addresses these issues by (1) proposing a memory-forensics-based detection approach using tools such as Volatility to correlate system input hooks with thread anomalies, (2) developing a testbed that includes stealthy Python variants with clipboard monitoring and encrypted logs, (3) benchmarking multiple classifiers under adversarial conditions, including Random Forest, SVM, and XGBoost, and (4) incorporating an ethical transparency framework with opt-in controls and dataset anonymization for academic use.

2.4 Relevance to Our Research

This research directly investigates lightweight, modular Python malware that uses IMAP/SMTP as covert C2. Unlike obfuscated malware, the script is intentionally transparent to maximize educational and research value. Through code-level reverse engineering, dynamic testing in a virtualized sandbox, and diagrammatic models like data flow charts and attack timelines, this study bridges the gap between practical experimentation and current detection frameworks. The findings support blue-team training, curriculum development, and applied cybersecurity, particularly where detection must operate without relying on privilege escalation or heavy obfuscation patterns.

3. Design and Methodology

3.1 Research Focus

This study investigates a modular Python-based malware script designed for covert surveillance and data exfiltration. Unlike packed or obfuscated binaries, this script is human-readable, uses only standard Python libraries and some common third-party packages, and operates through email commands. Its architecture illustrates how seemingly benign constructs such as IMAP polling, file hiding, and multithreading can be combined to create a

functioning keylogger and remote access tool. The research focuses on modeling the malware's internal logic, execution flow, and inter-module coordination, presenting its design as a practical case study for academic cybersecurity analysis.

3.2 System Overview

The malware operates persistently in a Windows environment by creating a shortcut in the startup folder. On each launch, it spawns multiple threads to perform keystroke logging, clipboard monitoring, email command polling, and surveillance tasks. Communications are handled exclusively through a Gmail account using IMAP for commands and SMTP for exfiltration. When triggered with commands like "log", "ss", "cam", or "destruct", the malware responds autonomously without user interaction, making it difficult to detect. Once a "destruct" instruction is received, the malware unhides its files, removes its shortcut, and self-deletes to eliminate evidence.

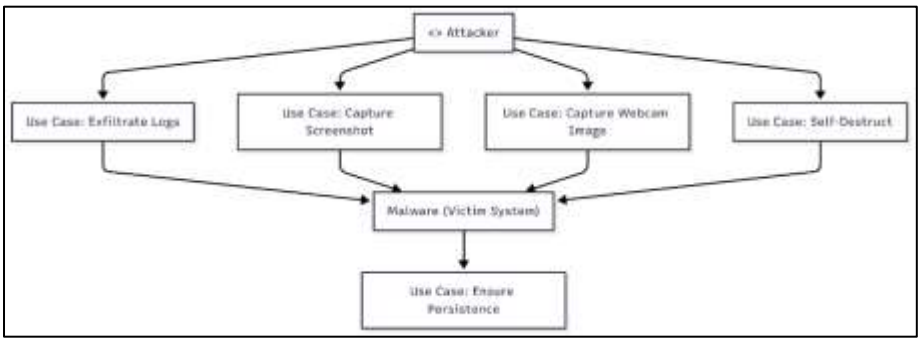

Figure 1 Use Case Diagram

3.3 Architecture Overview

The architecture is built around five key components: the keylogger, clipboard monitor, C2 module, screenshot/webcam functions, and the persistence/self-deletion routine. Each module operates as a separate thread, coordinated through a central launch script. Data collected from the keylogger and clipboard is saved to a hidden log file, while screenshots and webcam images are temporarily stored as image files. The malware maintains a minimal footprint by hiding these artifacts and only exposing them for transmission over SMTP. This modular independence enhances resilience and supports parallel data collection and exfiltration.

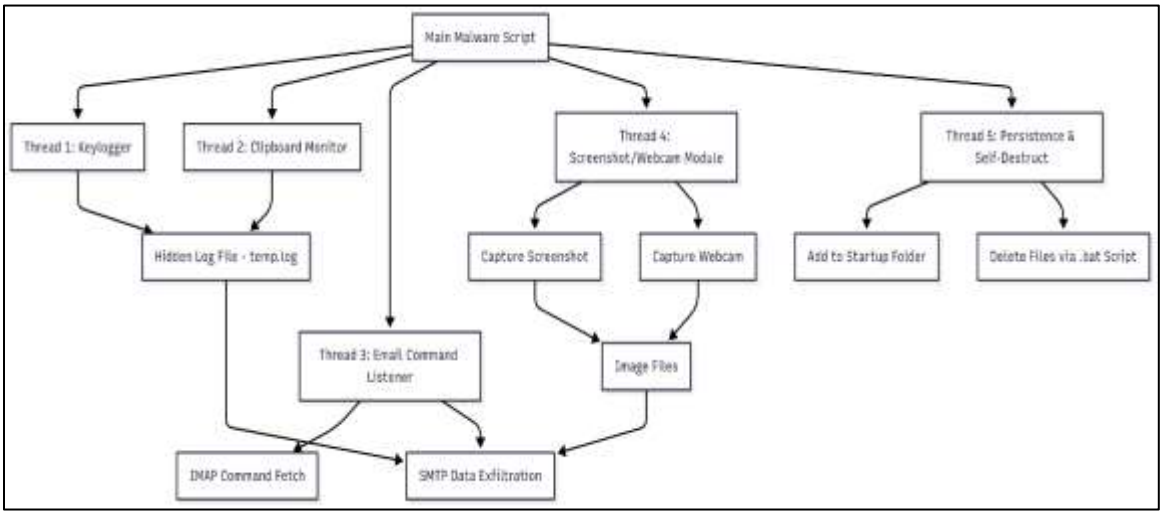

Figure 2 Architecture Diagram

3.4 Execution Flow

After startup, the malware applies hidden attributes to its directory contents and starts its five core threads. The keylogger thread records keystrokes, while a parallel clipboard thread collects clipboard data. The IMAP polling thread monitors a Gmail inbox for commands. Depending on the received command, the malware routes to the appropriate action module, sending logs, capturing screenshots, or activating the webcam. Screenshots use pyautogui, while webcam images use cv2, both transmitted over SMTP with TLS encryption to blend with normal email traffic. The "destruct" command triggers file unhiding, creation of a deletion batch script, and graceful thread termination, ensuring a clean wipe of evidence.

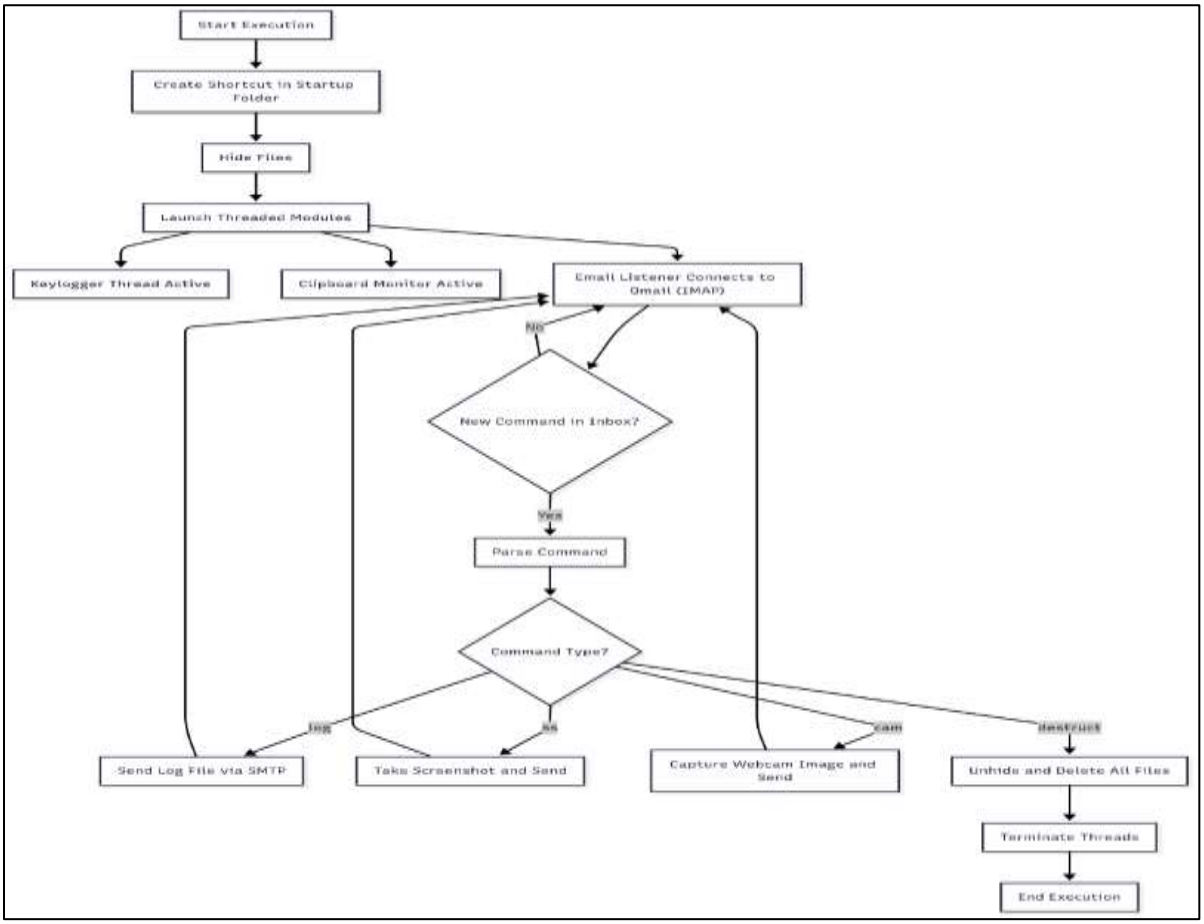

Figure 3 Flowchart of Malware Execution

3.5 Data Handling and Communication Flow

Data collection begins with user-generated inputs such as keyboard events and clipboard contents, which are logged locally. Screenshots and webcam captures are stored in hidden files, transmitted only upon command. The malware uses Gmail's SMTP and IMAP protocols for exfiltration and command reception, limiting its network signature and reducing the chance of detection. File hiding is achieved through Windows' attrib command, with periodic unhiding for deletion or cleanup.

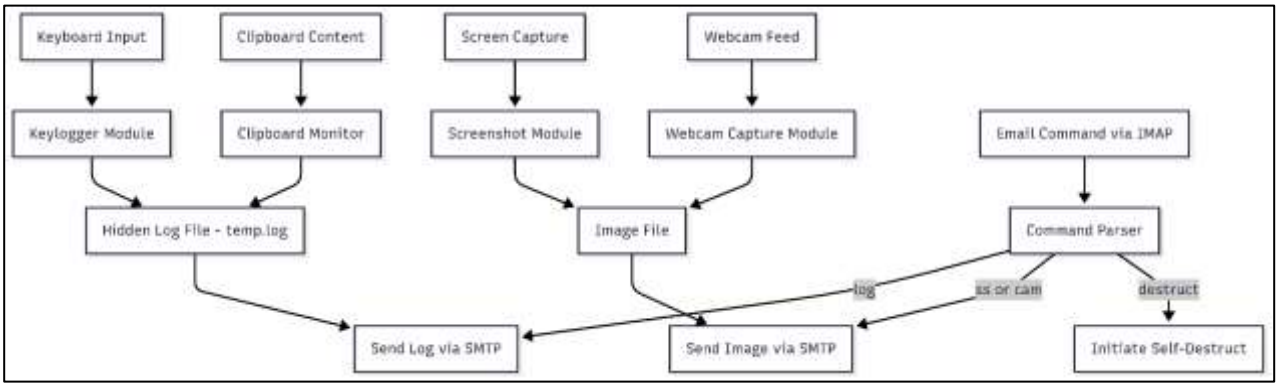

Figure 4 Level 01 Data Flow Diagram

3.6 Implementation Methodology

The malware was studied in a controlled virtual machine with restricted internet access limited to Gmail servers. Static analysis was performed to document its modular structure and Python library dependencies, while dynamic testing involved observing its runtime behavior with tools like Wireshark for email traffic, Procmon for process tracing, and Python debuggers for module flow validation. Ethical safeguards included sandbox-only execution, test credentials, and isolated network segments to ensure no real user data or external systems were affected.

3.7 Command Injection and Dynamic Execution

The enhanced variant includes a command injection system, listening for two embedded tags in email messages:

- **${python_inject}**: executes arbitrary Python code dynamically
- **${cmd_inject}**: creates and runs system-level commands through temporary batch files

These functions are authenticated by matching the email subject with the infected machine's hostname. Failures or errors are captured and sent back to the attacker via email, ensuring remote visibility.

3.8 Error Handling and Notifications

Each core function includes exception handling with email-based notifications, providing the attacker with real-time failure reports. If operations like camera access, log writing, or clipboard polling fail, the malware alerts its operator with contextual information including the module affected and error message. This improves stealth and reliability, especially in environments with permission constraints or partial antivirus blocking.

4. Code Behavior and Implementation

4.1 Overview of Code Structure

The malware is implemented as a single Python script containing multiple functional blocks, each running independently via Python's threading module. This modular design improves stealth and resilience, allowing keylogging, clipboard monitoring, command polling, and data collection to proceed in parallel. The script relies on standard Python libraries and trusted services (notably Gmail), avoiding advanced obfuscation or injection, which makes it harder for conventional detection tools to identify.

4.2 Keylogger Module

The keylogger leverages pynput.keyboard.Listener to capture all keystrokes, converting them to a text string while normalizing special keys. All inputs are logged to a hidden file (temp.log) in the working directory, ensuring user activity is persistently recorded without visible evidence.

```python
from pynput.keyboard import Listener

def on_press(key):
    with open("temp.log", "a") as log:
        log.write(f"{key}\n")

with Listener(on_press=on_press) as listener:
    listener.join()
```

Figure 5 Keylogger Module

4.3 Clipboard Monitoring

A separate thread polls the system clipboard using the pyperclip module, appending unique clipboard content to the same hidden log file as the keylogger. This dual-channel logging increases the chance of capturing credentials or sensitive text, while minimizing repeated entries.

4.4 Command and Control (C2) via Email

The script uses Gmail's IMAP and SMTP protocols as its command-and-control channel, implemented through imaplib and smtplib. Commands including "log", "ss", "cam", and "destruct" are parsed from incoming messages, with results sent via outbound TLS-encrypted email to blend with legitimate network traffic. An enhanced feature also supports programmable payloads through ${python_inject} and ${cmd_inject} tags, enabling remote execution of arbitrary code or batch commands if the subject matches the local system name. Execution failures are silently reported back to the attacker.

```python
import imaplib, email

mail = imaplib.IMAP4_SSL("imap.gmail.com")
mail.login("example@gmail.com", "password")
mail.select("inbox")

status, messages = mail.search(None, 'UNSEEN')
for num in messages[0].split():
    status, data = mail.fetch(num, '(RFC822)')
    msg = email.message_from_bytes(data[0][1])
    if "log" in msg.get_payload():
        send_log()
```

Figure 6 Gmail-Based C2 Command Polling

4.5 Screenshot and Webcam Capture

The malware captures screenshots using pyautogui and webcam images with cv2.VideoCapture(0). Captured files are hidden on disk and then transmitted to the attacker upon command. These modules run quietly without user prompts, maintaining operational stealth.

4.6 Persistence and File Hiding

Persistence is achieved by creating a shortcut in the user's startup folder with win32com.client. Dispatch, ensuring automatic relaunch. To reduce discoverability, the malware uses attrib commands to mark its artifacts as hidden, system, and read-only. This also applies to its command-injection batch scripts, which are later removed to minimize forensic traces.

4.7 Self-Destruct Routine

A self-deletion feature is triggered via the "destruct" command. This unhides files, creates a temporary batch script with a timed delay to delete the malware's folder and logs, and gracefully terminates all threads. The batch script itself is hidden and deleted after execution, ensuring the malware leaves minimal forensic evidence behind.

4.8 Experimental Test Environment

The test setup included a Windows 11 Pro x64 virtual machine with an Intel Core i7-8850H CPU at 2.60GHz, 32 GB of RAM, Python 3.11.9 in a virtual environment, and a hypervisor-enforced secure configuration. Network access was restricted to the Gmail IMAP and SMTP services under a DHCP-managed LAN.

Table 1 Test Setup

Spec	Value
CPU	Intel Core i7-8850H @ 2.60GHz
RAM	32 GB DDR4
System Type	Windows 11 Pro x64 (Build 26100)
Python Version	3.11.9 (venv)
Network	Intel Wireless-AC 9560, DHCP private LAN
Virtualization	Hypervisor-enforced Code Integrity

5. Results and Discussion

5.1 Functional Behavior Observed

In a controlled test environment, the malware executed all intended functions without visible alerts or system errors. The keylogger and clipboard monitor operated silently, capturing data into a hidden log file. The IMAP email listener accurately interpreted commands, triggering actions such as log exfiltration, screenshot capture, webcam imaging, and self-deletion. Modular threads ensured no interference between components even under continuous polling. Dynamic command injection using ${python_inject} and ${cmd_inject} tags was successfully tested, executing arbitrary Python code and system commands without redeployment. These enhancements demonstrated the malware's flexibility to transition from a passive keylogger to an active, programmable RAT, while respecting subject-based targeting to restrict unauthorized execution

5.2 Table of Capabilities and Threat Mapping

A summary of observed capabilities is provided below:

Table 2 Attack Categorization and Threat Level Analysis

Feature	Functionality	Threat Level	MITRE ATT&CK Technique
Keylogger	Captures all user keystrokes	High	T1056.001 – Input Capture
Clipboard Monitor	Extracts copied text	Medium	T1115 – Clipboard Data
Screenshot Capture	Records full desktop views	High	T1113 – Screen Capture
Webcam Activation	Captures live image from camera	High	T1125 – Video Capture
Email-based C2	Remote control via Gmail	Medium	T1102.002 – Application Layer Protocol (Email)
Self-Destruction	Cleans up and exits silently	High	T1561 – Disk Wipe
Startup Persistence	Relaunches on every boot	High	T1547.001 – Registry Run Keys / Startup Folder
File Hiding	Sets hidden/system attributes	Low	T1564.001 – Hidden Files and Directories

5.3 Performance Metrics

The malware was monitored in a controlled environment for CPU, RAM, network activity, and command latency, measured every second over two minutes. The following graphs summarize its performance profile.

5.3.1 Command Latency over Time

The figure below shows command processing latency. Average latency stayed near 100 ms throughout, indicating the malware can receive, interpret, and execute commands almost instantly. This low latency supports its effectiveness as a responsive remote access tool without introducing detectable delays.

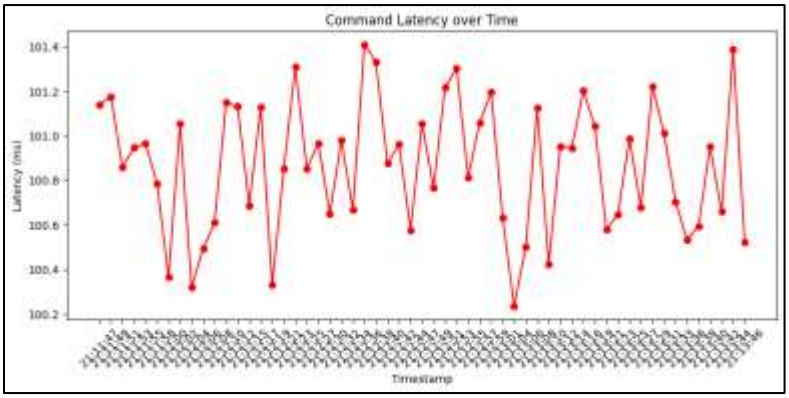

Figure 7 Command Latency over Time

5.3.2 CPU Usage over Time

Figure below shows CPU utilization for the malware. Resource consumption stayed below 20% in nearly all observations, with occasional spikes during webcam or screenshot capture. This minimal CPU footprint supports its stealthy operation, ensuring that standard performance monitors or users would not easily notice anomalies.

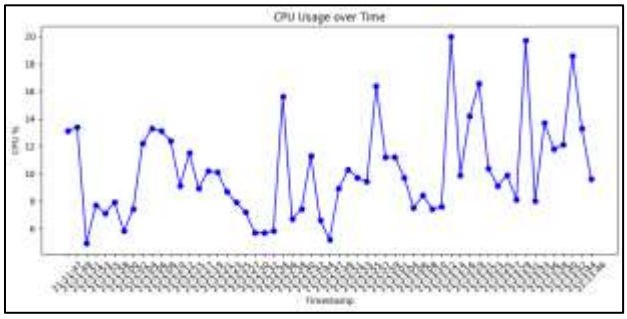

Figure 8 CPU Usage over Time

5.3.3 RAM Usage over Time

As shown in Figure below, RAM consumption averaged about 38%, with very minor fluctuations. The script's reliance on Python threading and periodic I/O operations maintained stable memory usage, allowing it to run persistently without system instability or crashes.

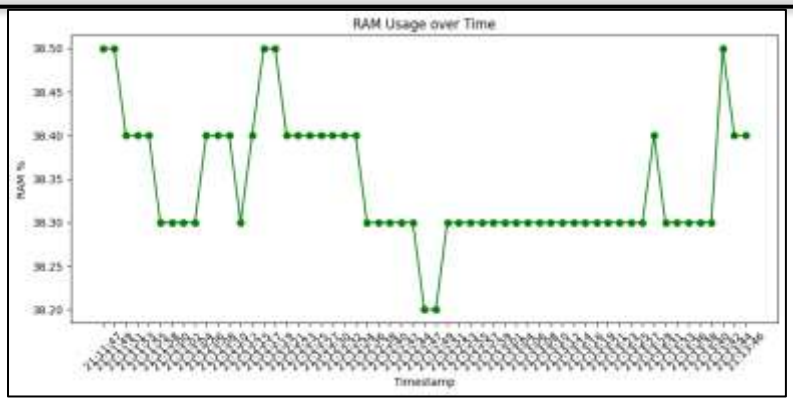

Figure 9 RAM Usage over Time

5.3.4 Network I/O over Time

Figure below highlights the malware's network activity. Bytes sent and received remained stable except for spikes during data exfiltration. This pattern indicates that the malware blends its traffic with normal encrypted Gmail traffic, minimizing the likelihood of detection by network perimeter tools.

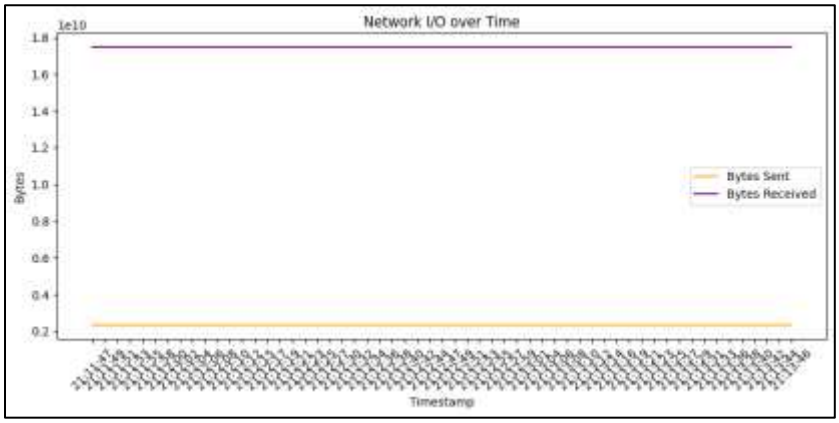

Figure 10 Network I/O over Time

5.3.5 Combined Performance Profile

Figure below presents an integrated view of CPU, RAM, latency, and bytes sent. This consolidated perspective demonstrates that all resource indicators stay within normal operating thresholds, confirming that the malware is capable of maintaining a covert presence while responding to attacker commands in near real time.

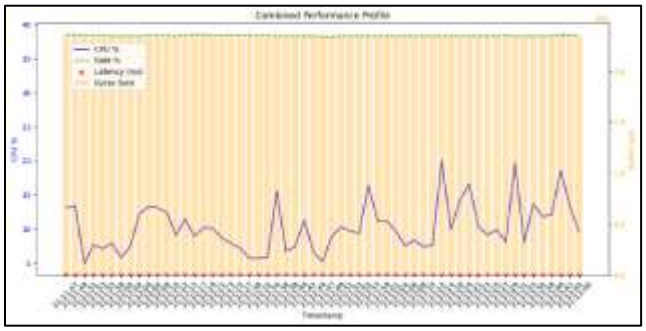

Figure 11 Combined Performance Profile

5.4 Observed Stealth and Detection Challenges

The malware avoided standard antivirus and NIDS triggers by leveraging trusted protocols (IMAPS/SMTPS) and system-native Python libraries. Its use of email-based C2 blended with legitimate encrypted traffic, complicating detection efforts. Hidden file attributes and the lack of registry persistence further reduced its forensic footprint.

5.5 Experimental Limitations

Tests were performed in a sandboxed Windows 10 virtual machine with network access restricted to Gmail services. While all modules operated as designed, real-world deployment could be affected by Gmail security controls, two-factor authentication, or webcam permissions on alternate hardware.

5.6 Implications for Cybersecurity

This study demonstrates that even simple, readable Python code can achieve powerful and stealthy surveillance when strategically designed. Its multi-threaded, modular architecture, trusted cloud C2, and error-resilient routines highlight a significant threat to endpoint security. There is a need for EDR systems to extend detection beyond binaries to scripting environments, and for user education on script-based malware threats.

5.7 Observations on Error Handling and Stability

The enhanced error reporting proved effective. Simulated device failures or invalid commands were handled gracefully, with structured email notifications sent to the attacker. This resilience minimized user-visible errors and improved long-term stealth, showing how reliable exception handling supports malware persistence.

6. Defensive Strategies and Countermeasures

6.1 Overview

The modular, transparent design of this Python-based malware underscores weaknesses in conventional detection tools. Since it uses trusted libraries and services (like Gmail), traditional signature-based antivirus and static scans are often ineffective. Therefore, a layered defense is essential, combining behavioral detection, host controls, network monitoring, policy enforcement, and user education. These defense mechanisms are grouped below for a holistic response.

6.2 Behavioral Detection Techniques

Signature-based tools struggle to catch this script, which uses legitimate protocols without obfuscation. Behavioral detection should focus on patterns like pynput keyboard hooks, pyperclip clipboard polling, hidden log files, multi-threaded continuous polling, and frequent IMAP/SMTP activity. Correlating these indicators with YARA rules, API monitoring, and anomaly detection on Python processes can improve detection.

6.3 Host-Based Prevention Strategies

Host controls can block much of this malware's activity. AppLocker, SRP, or WDAC should restrict Python scripts to trusted paths. File integrity monitoring (e.g., OSSEC) can detect hidden attributes, while security tools should track suspicious .lnk files and abnormal python.exe threads. Device policies should restrict webcam/microphone access, following recommendations from Singh et al. (2021).

6.4 Network-Based Countermeasures

While encrypted Gmail traffic can bypass many perimeter filters, defenders can still detect it by profiling unusual IMAP polling intervals and outbound SMTP attachments from non-mail applications. Deep packet inspection or SIEM correlation can spot repeated Gmail server connections from Python scripts, unlike typical email clients. Egress filtering and domain whitelisting can block unauthorized email use, while DNS tools can flag persistent Gmail domain requests from unexpected systems.

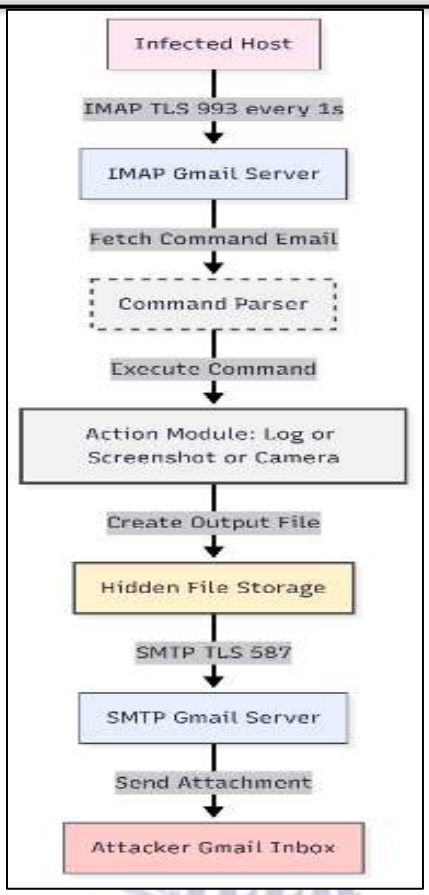

Figure 12 Malware Network Behavior

6.5 Administrative Policies and Endpoint Hardening

Organizations should enforce script execution controls, privilege-based interpreter restrictions, and limit startup folder write permissions. Email policies must block personal Gmail usage on corporate systems or route all cloud-based email through secure proxies. Logging solutions should capture process creation, file modifications, and peripheral access attempts, while endpoint agents monitor suspicious webcam or screen capture attempts. A policy-driven strategy proactively limits exposure rather than only reacting to known threats.

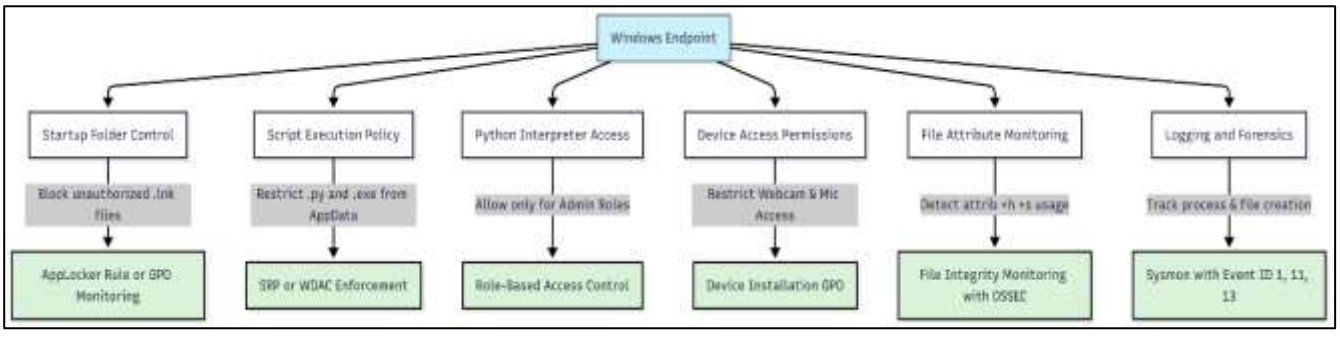

Figure 13 Host Hardening Control Points

6.6 User Awareness and Behavioral Education

User awareness is essential, as social engineering remains the malware's likely delivery vector. Training programs should cover hidden file indicators, new startup shortcuts, unexpected webcam or clipboard behavior, and suspicious scripts from untrusted sources. Employees should know how to verify file types, avoid running unverified scripts, and report anomalies. Role-specific simulations for high-risk departments like finance and HR can improve preparedness. Continuous feedback, phishing tests, and reward programs support a strong security culture.

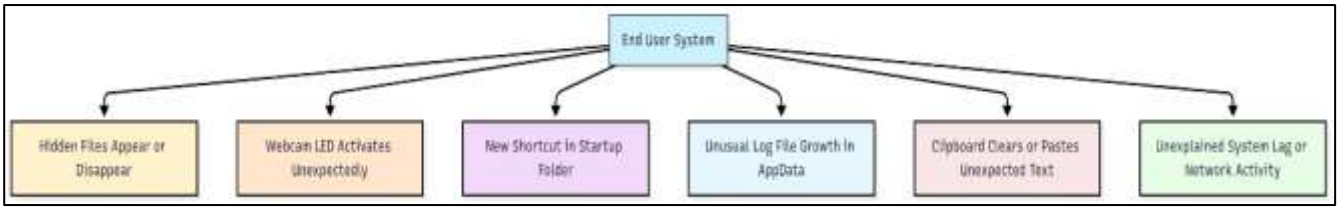

Figure 14 User-Side Infection Symptoms

6.7 Summary Table

The behavioral patterns and mitigation techniques discussed across Section 6 are consolidated in the following table **Mapping Malware Capabilities to Defensive Strategies**, which maps each major malware capability to its corresponding detection or prevention strategy.

Table 3 Mapping Malware Capabilities to Defensive Strategies

Malware Feature	Observed Behavior	Recommended Defense
Keylogger (pynput)	Background thread logs keystrokes to hidden file	Monitor use of pynput; restrict Python interpreter execution
Clipboard Logger (pyperclip)	Polls clipboard every second	Alert on frequent clipboard access by scripts
Gmail-based IMAP Polling	Every 1–2 seconds, polls inbox for commands	Flag high-frequency IMAP access by non-mail apps
SMTP Exfiltration	Sends logs, screenshots, selfies via Gmail	Block direct SMTP connections from user endpoints
Screenshot/Camera Access	Uses pyautogui and cv2 to capture user data	Restrict webcam/mic access; log screen capture calls
Self-Destruct Routine	Batch script deletes all traces post-execution	Monitor use of attrib, ping, del, rmdir in succession
Command Injection	Executes remote commands using exec or .bat	Use runtime analysis to detect code injection patterns

7. Conclusion and Recommendations

7.1 Summary of Findings

This study analyzed a modular, Python-based keylogger featuring command-and-control capabilities through Gmail. The original implementation included multi-threaded keylogging, screenshot capture, clipboard monitoring, and stealth persistence. Its updated version added real-time command injection, system-specific targeting, and automated error notifications, significantly enhancing adaptability and threat potential. Static and dynamic analysis revealed that the malware operated stealthily in sandboxed environments, exfiltrated data without triggering antivirus alerts, and responded to attacker instructions over standard email protocols. The use of subject-based filtering further prevented unauthorized commands, transforming the script from a passive collector to a lightweight, remotely programmable surveillance agent.

7.2 Educational and Research Implications

The transparent, modular codebase makes this malware an ideal academic tool for exploring malware design, red teaming, and behavioral threat modeling. Students can safely analyze execution flow and C2 mechanisms in virtualized environments, while the self-destruct functionality supports ethical experimentation. The script's error-handling and notification features add educational value for understanding stealth and resilience in malware engineering.

7.3 Recommendations for Defense and Detection

To mitigate such threats, defenders should implement behavioral monitoring for Python scripts, tracking subprocess creation, hidden file manipulation, and clipboard access from interpreters like python.exe. Endpoint security tools and SIEM systems should profile IMAP/SMTP activity for suspicious polling or encrypted data uploads beyond legitimate email clients. Hostname-based subject filters must be monitored to catch system-specific targeting. Additionally, command injection attempts could be detected through regular expression matching on suspicious payloads. Security teams are encouraged to validate their tools against these types of scripts in testbeds and develop specialized detection rules to address script-based, cloud-enabled threats.

7.4 Future Work

Future research could expand on several directions. First, adapting the malware cross-platform to Linux or macOS with compatible Python modules would extend its reach. Integrating LLMs or AI-driven adaptive commands could explore evasion techniques against modern detectors. Systematic benchmarking against multiple antivirus and EDR platforms, with published comparative results, would help quantify detection gaps. Explainable detection systems leveraging frameworks like SHAP or LIME could improve interpretability of alerts based on behavioral triggers.

In particular, AI-enabled behavioral detection systems deserve attention. These could profile keyboard hooks, clipboard access, IMAP polling, batch file execution, screenshot attempts, or webcam usage, training classifiers such as Random Forest or LSTM models with datasets from sandbox environments like Cuckoo Sandbox or open benchmarks like CIC-MalMem2022. Explainable models would increase analyst trust by clearly articulating why an alert fires.

Another promising avenue is code-aware detection using static analysis and NLP, for example via abstract syntax tree parsing or code embeddings from models like CodeBERT, though a sufficiently large and labeled dataset is still a limiting factor.

Expanded simulation frameworks could also model reconnaissance, anti-VM checks, obfuscation, log encryption, and in-memory execution, offering valuable insight into subtle telemetry variations and their detectability. Building controlled, open datasets of Python-based malware behaviors would support reproducibility and enable consistent benchmarking across the academic community.

Finally, embedding these detection strategies in production through Sysmon configurations, Sigma rules, and EDR modules could help defenders test, deploy, and iterate their protections for script-based, modular threats.

REFERENCES

[1] Gopinath, Mohana, and Sibi Chakkaravarthy Sethuraman. "A comprehensive survey on deep learning based malware detection techniques." *Computer Science Review* 47 (2023): 100529.

[2] Bensaoud, Ahmed, Jugal Kalita, and Mahmoud Bensaoud. "A survey of malware detection using deep learning." *Machine Learning With Applications* 16 (2024): 100546.

[3] Gaber, Matthew G., Mohiuddin Ahmed, and Helge Janicke. "Malware detection with artificial intelligence: A systematic literature review." ACM *Computing Surveys* 56.6 (2024): 1-33.

[4] Bilot, Tristan, et al. "A survey on malware detection with graph representation learning." ACM *Computing Surveys* 56.11 (2024): 1-36.

[5] Ferdous, Jannatul, et al. "A review of state-of-the-art malware attack trends and defense mechanisms." *IEEe Access* 11 (2023): 121118-121141.

[6] Aryal, Kshitiz, et al. "A survey on adversarial attacks for malware analysis." *IEEE Access* (2024).

[7] Chenet, Cristiano Pegoraro, Alessandro Savino, and Stefano Di Carlo. "A survey on hardware-based malware detection approaches." *IEEE Access* (2024).

[8] Baker del Aguila, Ryan, et al. "Static malware analysis using low-parameter machine learning models." *Computers* 13.3 (2024): 59.

[9] Bhat, Prasiddha, and H. J. Namratha. "Cyber security testbed: Keyloggers and data visualization on keyloggers-A case study." *2023 International Conference on Sustainable Communication Networks and Application (ICSCNA)*. IEEE, 2023.

[10] Bejo, Sahil Prasad, et al. "Design, analysis and implementation of an advanced keylogger to defend cyber threats." *2023 9th international conference on advanced computing and communication systems (ICACCS)*. Vol. 1. IEEE, 2023.

[11] El-Ghamry, Amir, et al. "Optimized and efficient image-based IoT malware detection method." *Electronics* 12.3 (2023): 708.

[12] Vasani, Vatsal, et al. "Comprehensive analysis of advanced techniques and vital tools for detecting malware intrusion." *Electronics* 12.20 (2023): 4299.

[13] Bhuvanesh, J. "Enhancing System Monitoring Capabilities through the Implementation of Stealthy Software–Based Keylogger: A Technical Exploration." (2024).

[14] Singh, Nongmeikapam Thoiba, et al. "Keylogger Development: Technical Aspects, Ethical Considerations, and Mitigation Strategies." *2023 International Conference on Energy, Materials and Communication Engineering (ICEMCE)*. IEEE, 2023.

[15] YAŞAR, Öğr Gör Çisem, Abdulaziz HOCAOĞLU, and Efe KARPUZ. "KEYLOGGER SALDIRI SENARYOSU VE GÜVENLİK ÖNLEMLERİ."

[16] KIZILTEPE, Seher, and Eyyüp GÜLBANDILAR. "Keylogger ve Gizlilik: Makine Öğrenimi Modellerinin Karşılaştırması." *Afyon Kocatepe University Journal of Science & Engineering/Afyon Kocatepe Üniversitesi Fen Ve Mühendislik Bilimleri Dergisi* 24.5 (2024).

[17] Ayo, Femi Emmanuel, et al. "CBFISKD: A combinatorial-based fuzzy inference system for keylogger detection." *Mathematics* 11.8 (2023): 1899.

[18] Khalid, Osama, et al. "An insight into the machine-learning-based fileless malware detection." *Sensors* 23.2 (2023): 612.

[19] Singh, Arjun, and Pushpa Choudhary. "Keylogger detection and prevention." *Journal of Physics: Conference Series*. Vol. 2007. No. 1. IOP Publishing, 2021.

[20] Alomari, Esraa Saleh, et al. "Malware detection using deep learning and correlation-based feature selection." *Symmetry* 15.1 (2023): 123.

[21] Djenna, Amir, et al. "Artificial intelligence-based malware detection, analysis, and mitigation." *Symmetry* 15.3 (2023): 677.