Chasing Eddies: New Rust- based InfoStealer used in CAPTCHA campaigns

elastic.co/security-labs/eddiestealer

Subscribe

Preamble

Elastic Security Labs has uncovered a novel Rust-based infostealer distributed via Fake CAPTCHA campaigns. This malware is hosted on multiple adversary-controlled web properties. This campaign leverages deceptive CAPTCHA verification pages that trick users into executing a malicious PowerShell script, which ultimately deploys the infostealer, harvesting sensitive data such as credentials, browser information, and cryptocurrency wallet details. We are calling this malware EDDIESTEALER.

This adoption of Rust in malware development reflects a growing trend among threat actors seeking to leverage modern language features for enhanced stealth, stability, and resilience against traditional analysis workflows and threat detection engines. A seemingly simple infostealer written in Rust often requires more dedicated analysis efforts compared to its C/C++ counterpart, owing to factors such as zero-cost

abstractions, Rust's type system, compiler optimizations, and inherent difficulties in analyzing memory-safe binaries.

Key takeaways

- Fake CAPTCHA campaign loads EDDIESTEALER
- EDDIESTEALER is a newly discovered Rust infostealer targeting Windows hosts
- EDDIESTEALER receives a task list from the C2 server identifying data to target

Intial access

Overview

Fake CAPTCHAs are malicious constructs that replicate the appearance and functionality of legitimate CAPTCHA systems, which are used to distinguish between human users and automated bots. Unlike their legitimate counterparts, fake CAPTCHAs serve as gateways for malware, leveraging social engineering to deceive users. They often appear as prompts like "Verify you are a human" or "I'm not a robot," blending seamlessly into compromised websites or phishing campaigns. We have also encountered a similar campaign distributing GHOSTPULSE in late 2024.

From our telemetry analysis leading up to the delivery of EDDIESTEALER, the initial vector was a compromised website deploying an obfuscated React-based JavaScript payload that displays a fake "I'm not a robot" verification screen.

Mimicking Google's reCAPTCHA verification interface, the malware uses the `document.execCommand("copy")` method to copy a PowerShell command into the user's clipboard, next, it instructs the user to press Windows + R (to open the Windows run dialog box), then Ctrl + V to paste the clipboard contents, and finally Enter to execute the malicious PowerShell command.

This command silently downloads a second-stage payload (`gverify.js`) from the attacker-controlled domain `hxxps://llll.fit/version/` and saves it to the user's `Downloads` folder.

Finally, the malware executes `gverify.js` using `cscript` in a hidden window.

`gverify.js` is another obfuscated JavaScript payload that can be deobfuscated using open-source tools. Its functionality is fairly simple: fetching an executable (EDDIESTEALER) from `hxxps://llll.fit/io` and saving the file under the user's `Downloads` folder with a pseudorandom 12-character file name.

EDDIESTEALER

Overview

EDDIESTEALER is a novel Rust-based commodity infostealer. The majority of strings that give away its malicious intent are encrypted. The malware lacks robust anti-sandbox/VM protections against behavioral fingerprinting. However, newer variants suggest that the anti-sandbox/VM checks might be occurring on the server side. With relatively straightforward capabilities, it receives a task list from the C2 server as part of its configuration to target specific data and can self-delete after execution if specified.

Stripped Symbols

EDDIESTEALER samples featured stripped function symbols, likely using Rust's default compilation option, requiring symbol restoration before static analysis. We used `rustbinsign`, which generates signatures for Rust standard libraries and crates based on specific Rust/compiler/dependency versions. While `rustbinsign` only detected `hashbrown` and `rustc-demangle`, suggesting few external crates being used, it failed to identify crates such as `tinyjson` and `tungstenite` in newer variants. This occurred due to the lack of clear string artifacts. It is still possible to manually identify crates by finding unique strings and searching for the repository on GitHub, then download, compile and build signatures for them using the `download_sign` mode. It is slightly cumbersome if we don't know the exact version of the crate being used. However, restoring the standard library and runtime symbols is sufficient to advance the static analysis process.

String Obfuscation

EDDIESTEALER encrypts most strings via a simple XOR cipher. Decryption involves two stages: first, the XOR key is derived by calling one of several key derivation functions; then, the decryption is performed inline within the function that uses the string.

The following example illustrates this, where `sub_140020fd0` is the key derivation function, and `data_14005ada8` is the address of the encrypted blob.

Each decryption routine utilizes its own distinct key derivation function. These functions consistently accept two arguments: an address within the binary and a 4-byte constant value. Some basic operations are then performed on these arguments to calculate the address where the XOR key resides.

Binary Ninja has a handy feature called `User-Informed Data Flow` (UIDF), which we can use to set the variables to known values to trigger a constant propagation analysis and simplify the expressions. Otherwise, a CPU emulator like Unicorn paired with a scriptable binary analysis tool can also be useful for batch analysis.

There is a general pattern for thread-safe, lazy initialization of shared resources, such as encrypted strings for module names, C2 domain and port, the sample's unique identifier - that are decrypted only once but referenced many times during runtime. Each specific getter function checks a status flag for its resource; if uninitialized, it calls a shared, low-level synchronization function. This synchronization routine uses atomic operations and OS wait primitives (`WaitOnAddress`/`WakeByAddressAll`) to ensure only one thread executes the actual initialization logic, which is invoked indirectly via a function pointer in the vtable of a `dyn Trait` object.

API Obfuscation

EDDIESTEALER utilizes a custom WinAPI lookup mechanism for most API calls. It begins by decrypting the names of the target module and function. Before attempting resolution, it checks a locally maintained hashtable to see if the function name and address have already been resolved. If not found, it dynamically loads the required module using a custom `LoadLibrary` wrapper, into the process's address space, and invokes a well-known implementation of GetProcAddress to retrieve the address of the exported function. The API name and address are then inserted into the hashtable, optimizing future lookups.

Mutex Creation

EDDIESTEALER begins by creating a mutex to ensure that only one instance of the malware runs at any given time. The mutex name is a decrypted UUID string `431e2e0e-c87b-45ac-9fdb-26b7e24f0d39` (unique per sample), which is later referenced once more during its initial contact with the C2 server.

Sandbox Detection

EDDIESTEALER performs a quick check to assess whether the total amount of physical memory is above ~4.0 GB as a weak sandbox detection mechanism. If the check fails, it deletes itself from disk.

Self-Deletion

Based on a similar self-deletion technique observed in LATRODECTUS, EDDIESTEALER is capable of deleting itself through NTFS Alternate Data Streams renaming, to bypass file locks.

The malware uses `GetModuleFileName` to obtain the full path of its executable and `CreateFileW` (wrapped in `jy::ds::OpenHandle`) to open a handle to its executable file with the appropriate access rights. Then, a `FILE_RENAME_INFO` structure with a new stream name is passed into `SetFileInformationByHandle` to rename the default stream `$DATA` to `:metadata`. The file handle is closed and reopened, this time using `SetFileInformationByHandle` on the handle with the `FILE_DISPOSITION_INFO.DeleteFile` flag set to `TRUE` to enable a "delete on close handle" flag.

Additional Configuration Request

The initial configuration data is stored as encrypted strings within the binary. Once decrypted, this data is used to construct a request following the URI pattern: `<C2_ip_or_domain>/<resource_path>/<UUID>`. The `resource_path` is specified as `api/handler`. The `UUID`, utilized earlier to create a mutex, is used as a unique identifier for build tracking.

EDDIESTEALER then communicates with its C2 server by sending an HTTP GET request with the constructed URI to retrieve a second-stage configuration containing a list of tasks for the malware to execute.

The second-stage configuration data is AES CBC encrypted and Base64 encoded. The Base64-encoded IV is prepended in the message before the colon (`:`).

```
Base64(IV):Base64(AESEncrypt(data))
```

The AES key for decrypting the server-to-client message is stored unencrypted in UTF-8 encoding, in the `.rdata` section. It is retrieved through a getter function.

The decrypted configuration for this sample contains the following in JSON format:

- Session ID
- List of tasks (data to target)
- AES key for client-to-server message encryption
- Self-delete flag

```
{
    "session": "<unique_session_id>",
    "tasks": [
        {
            "id": "<unique_task_id>",
            "prepare": [],
            "pattern": {
                "path": "<file_system_path>",
                "recursive": <true/false>,
                "filters": [
                    {
                        "path_filter": <null/string>,
                        "name": "<file_or_directory_name_pattern>",
                        "entry_type": "<FILE/DIR>"
                    },
                    ...
                ]
            },
            "additional": [
                {
                    "command": "<optional_command>",
                    "payload": {
                        "<command_specific_config>": <value>
                    }
                },
                ...
            ]
        },
        ...
    ],
    "network": {
        "encryption_key": "<AES_encryption_key>"
    },
    "self_delete": <true/false>
}
```

For this particular sample and based on the tasks received from the server during our analysis, here are the list of filesystem-based exfiltration targets:

- Crypto wallets
- Browsers
- Password managers
- FTP clients
- Messaging applications

Crypto Wallet	Target Path Filter
Armory	`%appdata%\\Armory\\*.wallet`
Bitcoin	`%appdata%\\Bitcoin\\wallets\\*`
WalletWasabi	`%appdata%\\WalletWasabi\\Client\\Wallets\\*`
Daedalus Mainnet	`%appdata%\\Daedalus Mainnet\\wallets\\*`
Coinomi	`%localappdata%\\Coinomi\\Coinomi\\wallets\\*`
Electrum	`%appdata%\\Electrum\\wallets\\*`
Exodus	`%appdata%\\Exodus\\exodus.wallet\\*`
DashCore	`%appdata%\\DashCore\\wallets\\*`
ElectronCash	`%appdata%\\ElectronCash\\wallets\\*`
Electrum-DASH	`%appdata%\\Electrum-DASH\\wallets\\*`
Guarda	`%appdata%\\Guarda\\IndexedDB`
Atomic	`%appdata%\\atomic\\Local Storage`

Browser	Target Path Filter
Microsoft Edge	`%localappdata%\\Microsoft\\Edge\\User Data\\` `[Web Data,History,Bookmarks,Local Extension Settings\\...]`

Browser	Target Path Filter
Brave	`%localappdata%\\BraveSoftware\\Brave-Browser\\User Data\\[Web Data,History,Bookmarks,Local Extension Settings\\...]`
Google Chrome	`%localappdata%\\Google\\Chrome\\User Data\\[Web Data,History,Bookmarks,Local Extension Settings\\...]`
Mozilla Firefox	`%appdata%\\Mozilla\\Firefox\\Profiles\\[key4.db,places.sqlite,logins.json,cookies.sqlite,formhistory.sqlite,webappsstore.sqlite,*+++*]`

Password Manager	Target Path Filter
Bitwarden	`%appdata%\\Bitwarden\\data.json`
1Password	`%localappdata%\\1Password\\[1password.sqlite,1password_resources.sqlite]`
KeePass	`%userprofile%\\Documents\\*.kdbx`

FTP Client	Target Path Filter
FileZilla	`%appdata%\\FileZilla\\recentservers.xml`
FTP Manager Lite	`%localappdata%\\DeskShare Data\\FTP Manager Lite\\2.0\\FTPManagerLiteSettings.db`
FTPbox	`%appdata%\\FTPbox\\profiles.conf`
FTP Commander Deluxe	`%ProgramFiles(x86)%\\FTP Commander Deluxe\\FTPLIST.TXT`
Auto FTP Manager	`%localappdata%\\DeskShare Data\\Auto FTP Manager\\AutoFTPManagerSettings.db`
3D-FTP	`%programdata%\\SiteDesigner\\3D-FTP\\sites.ini`
FTPGetter	`%appdata%\\FTPGetter\\servers.xml`
Total Commander	`%appdata%\\GHISLER\\wcx_ftp.ini`

Messaging App	Target Path Filter
Telegram Desktop	`%appdata%\\Telegram Desktop\\tdata\\*`

A list of targeted browser extensions can be found here.

These targets are subject to change as they are configurable by the C2 operator.

EDDIESTEALER then reads the targeted files using standard `kernel32.dll` functions like `CreateFileW`, `GetFileSizeEx`, `ReadFile`, and `CloseHandle`.

Subsequent C2 Traffic

After successfully retrieving the tasks, EDDIESTEALER performs system profiling to gather some information about the infected system:

- Location of the executable (`GetModuleFileNameW`)
- Locale ID (`GetUserDefaultLangID`)
- Username (`GetUserNameW`)
- Total amount of physical memory (`GlobalMemoryStatusEx`)
- OS version (`RtlGetVersion`)

Following the same data format (`Base64(IV):Base64(AESEncrypt(data))`) for client-to-server messages, initial host information is AES-encrypted using the key retrieved from the additional configuration and sent via an HTTP POST request to `<C2_ip_or_domain>/<resource_path>/info/<session_id>`. Subsequently, for each completed task, the collected data is also encrypted and transmitted in separate POST requests to `<C2_ip_or_domain>/<resource_path><session_id>/<task_id>`, right after each task is completed. This methodology generates a distinct C2 traffic pattern characterized by multiple, task-specific POST requests. This pattern is particularly easy to identify because this malware family primarily relies on HTTP instead of HTTPS for its C2 communication.

Our analysis uncovered encrypted strings that decrypt to panic metadata strings, disclosing internal Rust source file paths such as:

- `apps\bin\src\services\chromium_hound.rs`
- `apps\bin\src\services\system.rs`
- `apps\bin\src\structs\search_pattern.rs`
- `apps\bin\src\structs\search_entry.rs`

We discovered that error messages sent to the C2 server contain these strings, including the exact source file, line number, and column number where the error originated, allowing the malware developer to have built-in debugging feedback.

Chromium-specific Capabilities

Since the introduction of Application-bound encryption, malware developers have adapted to alternative methods to bypass this protection and gain access to unencrypted sensitive data, such as cookies. ChromeKatz is one of the more well-received open source solutions that we have seen malware implement. EDDIESTEALER is no exception—the malware developers reimplemented it in Rust.

Below is a snippet of the browser version checking logic similar to COOKIEKATZ, after retrieving version information from `%localappdata%\<browser_specific_path>\\User Data\\Last Version`.

COOKIEKATZ signature pattern for detecting COOKIEMONSTER instances:

CredentialKatz signature pattern for detecting CookieMonster instances:

Here is an example of the exact copy-pasted logic of COOKIEKATZ's `FindPattern`, where `PatchBaseAddress` is inlined.

The developers introduced a modification to handle cases where the targeted Chromium browser is not running. If inactive, EDDIESTEALER spawns a new browser instance using the command-line arguments `--window-position=-3000,-3000 https://google.com`. This effectively positions the new window far off-screen, rendering it invisible to the user. The objective is to ensure the malware can still read the memory (`ReadProcessMemory`) of the necessary child process - the network service process identified by the `--utility-sub-type=network.mojom.NetworkService` flag. For a more detailed explanation of this browser process interaction, refer to our previous research on MaaS infostealers.

Differences with variants

After analysis, more recent samples were identified with additional capabilities.

Information gathered on victim machines now include:

- Running processes
- GPU information
- Number of CPU cores
- CPU name
- CPU vendor

The C2 communication pattern has been altered slightly. The malware now preemptively sends host system information to the server before requesting its decrypted configuration. In a few instances where the victim machine was able to reach out to the C2 server but received an empty task list, the adjustment suggests an evasion tactic: developers have likely introduced server-side checks to profile the client environment and withhold the main configuration if a sandbox or analysis system is detected.

The encryption key for client-to-server communication is no longer received dynamically from the C2 server; instead, it is now hardcoded in the binary. The key used by the client to decrypt server-to-client messages also remains hardcoded.

Newer compiled samples exhibit extensive use of function inline expansion, where many functions - both user-defined and from standard libraries and crates - have been inlined directly into their callers more often, resulting in larger functions and making it difficult to isolate user code. This behavior is likely the result of using LLVM's inliner. While some functions remain un-inlined, the widespread inlining further complicates analysis.

In order to get all entries of Chrome's Password Manager, EDDIESTEALER begins its credential theft routine by spawning a new Chrome process with the `--remote-debugging-port=<port_num>` flag, enabling Chrome's DevTools Protocol over a local WebSocket interface. This allows the malware to interact with the browser in a headless fashion, without requiring any visible user interaction.

After launching Chrome, the malware queries `http://localhost:<port>/json/version` to retrieve the `webSocketDebuggerUrl`, which provides the endpoint for interacting with the browser instance over WebSocket.

Using this connection, it issues a `Target.createTarget` command with the parameter `chrome://password-manager/passwords`, instructing Chrome to open its internal password manager in a new tab. Although this internal page does not expose its contents to the DOM or to DevTools directly, opening it causes Chrome to decrypt and load stored credentials into memory. This behavior is exploited by EDDIESTEALER in subsequent steps through CredentialKatz lookalike code, where it scans the Chrome process memory to extract plaintext credentials after they have been loaded by the browser.

Based on decrypted strings `os_crypt`, `encrypted_key`, `CryptUnprotectData`, `local_state_pattern`, and `login_data_pattern`, EDDIESTEALER variants appear to be backward compatible, supporting Chrome versions that still utilize DPAPI encryption.

We have identified 15 additional samples of EDDIESTEALER through code and infrastructure similarities on VirusTotal. The observations table will include the discovered samples, associated C2 IP addresses/domains, and a list of infrastructure hosting EDDIESTEALER.

A Few Analysis Tips

Tracing

To better understand the control flow and pinpoint the exact destinations of indirect jumps or calls in large code blocks, we can leverage binary tracing techniques. Tools like `TinyTracer` can capture an API trace and generate a `.tag` file, which maps any selected API calls to be recorded to the executing line in assembly. Rust's standard library functions call into WinAPIs under the hood, and this also captures any code that calls `WinAPI` functions directly, bypassing the standard library's abstraction. The tag file can then be imported into decompiler tools to automatically mark up the code blocks using plugins like `IFL`.

Panic Metadata for Code Segmentation

Panic metadata - the embedded source file paths (.rs files), line numbers, and column numbers associated with panic locations - offers valuable clues for segmenting and understanding different parts of the binary. This, however, is only the case if such metadata has not been stripped from the binary. Paths like `apps\bin\src\services\chromium.rs`, `apps\bin\src\structs\additional_task.rs` or any path that looks like part of a custom project typically points to the application's unique logic. Paths beginning with `library<core/alloc/std>\src\` indicates code from the Rust standard library. Paths containing crate name and version such as `hashbrown-0.15.2\src\raw\mod.rs` point to external libraries.

If the malware project has a somewhat organized codebase, the file paths in panic strings can directly map to logical modules. For instance, the decrypted string `apps\bin\src\utils\json.rs:48:39` is referenced in `sub_140011b4c`.

By examining the call tree for incoming calls to the function, many of them trace back to `sub_14002699d`. This function (`sub_14002699d`) is called within a known C2 communication routine (`jy::C2::RetrieveAndDecryptConfig`), right after decrypting additional configuration data known to be JSON formatted.

Based on the `json.rs` path and its calling context, an educated guess would be that `sub_14002699d` is responsible for parsing JSON data. We can verify it by stepping over the function call. Sure enough, by inspecting the stack struct that is passed as reference to the function call, it now points to a heap address populated with parsed configuration fields.

For standard library and open-source third-party crates, the file path, line number, and (if available) the rustc commit hash or crate version allow you to look up the exact source code online.

Stack Slot Reuse

One of the optimization features involves reusing stack slots for variables/stack structs that don't have overlapping timelines. Variables that aren't "live" at the same time can share the same stack memory location, reducing the overall stack frame size. Essentially, a variable is live from the moment it is assigned a value until the last point where that value could be accessed. This makes the decompiled output confusing as the same memory offset may hold different types or values at different points.

To handle this, we can define unions encompassing all possible types sharing the same memory offset within the function.

Rust Error Handling and Enums

Rust enums are tagged unions that define types with multiple variants, each optionally holding data, ideal for modeling states like success or failure. Variants are identified by a discriminant (tag).

Error-handling code can be seen throughout the binary, making up a significant portion of the decompiled code. Rust's primary mechanism for error handling is the `Result<T, E>` generic enum. It has two variants: `Ok(T)`, indicating success and containing a value of type `T`, and `Err(E)`, indicating failure and containing an error value of type `E`.

In the example snippet below, a discriminant value of `0x8000000000000000` is used to differentiate outcomes of resolving the `CreateFileW` API. If `CreateFileW` is successfully resolved, the `reuse` variable type contains the API function pointer, and the `else` branch executes. Otherwise, the `if` branch executes, assigning an error information string from `reuse` to `arg1`.

For more information on how other common Rust types might look in memory, check out this cheatsheet and this amazing talk by Cindy Xiao!

Malware and MITRE ATT&CK

Elastic uses the MITRE ATT&CK framework to document common tactics, techniques, and procedures that threats use against enterprise networks.

Tactics

Techniques

Techniques represent how an adversary achieves a tactical goal by performing an action.

Detections

YARA

Elastic Security has created the following YARA rules related to this research:

- Windows.Infostealer.EddieStealer

Behavioral prevention rules

Observations

The following observables were discussed in this research.

Observable	Type	Name
47409e09afa05fcc9c9eff2c08baca3084d923c8d82159005dbae2029e1959d0	SHA-256	MvUlUwagHeZd.exe
162a8521f6156070b9a97b488ee902ac0c395714aba970a688d54305cb3e163f	SHA-256	:metadata (copy)
f8b4e2ca107c4a91e180a17a845e1d7daac388bd1bb4708c222cda0eff793e7a	SHA-256	AegZs85U6COc.exe
53f803179304e4fa957146507c9f936b38da21c2a3af4f9ea002a7f35f5bc23d	SHA-256	:metadata (copy)
20eeae4222ff11e306fded294bebea7d3e5c5c2d8c5724792abf56997f30aaf9	SHA-256	PETt3Wz4DXEL.exe
1bdc2455f32d740502e001fce51dbf2494c00f4dcadd772ea551ed231c35b9a2	SHA-256	Tk7n1al5m9Qc.exe
d905ceb30816788de5ad6fa4fe108a202182dd579075c6c95b0fb26ed5520daa	SHA-256	YykbZ173Ysnd.exe
b8b379ba5aff7e4ef2838517930bf20d83a1cfec5f7b284f9ee783518cb989a7	SHA-256	2025-04-03_20745dc4d048f67e0b62aca33be80283_akira_cobalt strike_satacom
f6536045ab63849c57859bbff9e6615180055c268b89c613dfed2db1f1a370f2	SHA-256	2025-03-23_6cc654225172ef70a189788746cbb445_akira_cobalt strike
d318a70d7f4158e3fe5f38f23a241787359c55d352cb4b26a4bd007fd44d5b80	SHA-256	2025-03-22_c8c3e658881593d798da07a1b80f250c_akira_cobalt strike
73b9259fecc2a4d0eeb0afef4f542642c26af46aa8f0ce2552241ee5507ec37f	SHA-256	2025-03-22_4776ff459c881a5b876da396f7324c64_akira_cobalt strike
2bef71355b37c4d9cd976e0c6450bfed5f62d8ab2cf096a4f3b77f6c0cb77a3b	SHA-256	TWO[1].file
218ec38e8d749ae7a6d53e0d4d58e3acf459687c7a34f5697908aec6a2d7274d	SHA-256	
5330cf6a8f4f297b9726f37f47cffac38070560cbac37a8e561e00c19e995f42	SHA-256	verifcheck.exe
acae8a4d92d24b7e7cb20c0c13fd07c8ab6ed8c5f9969504a905287df1af179b	SHA-256	3zeG4jGjFkOy.exe
0f5717b98e2b44964c4a5dfec4126fc35f5504f7f8dec386c0e0b0229e3482e7	SHA-256	verification.exe
e8942805238f1ead8304cfdcf3d6076fa0cdf57533a5fae36380074a90d642e4	SHA-256	g_verify.js
7930d6469461af84d3c47c8e40b3d6d33f169283df42d2f58206f43d42d4c9f4	SHA-256	verif.js
45.144.53[.]145	ipv4-addr	

Observable	Type	Name
84.200.154[.]47	ipv4-addr	
shiglimugli[.]xyz	domain-name	
xxxivi[.]com	domain-name	
llll[.]fit	domain-name	
plasetplastik[.]com	domain-name	
militrex[.]wiki	domain-name	

References

The following were referenced throughout the above research: