The ClickFix Deception: How a Fake CAPTCHA Deploys an Evasive Infostealer

By Louis Schürmann

Published: 2025-08-21 · Archived: 2026-04-06 00:22:28 UTC

Cybersecurity analyst Louis Schürmann from Swiss Post Cybersecurity investigated an incident that began with a seemingly harmless email and developed into a multi-stage attack.

Swiss Post Cybersecurity detected and analyzed an infostealer campaign. The attackers use Clickfix for initial access and DonutLoader for shellcode delivery. In our blog article, we show you step by step how users are being deceived.

The modern threat landscape is defined by its complexity. Rather than relying on simple email attachments, attackers now use complex infection chains that incorporate legitimate system tools to remain invisible. Cybersecurity analyst Louis Schürmann from Swiss Post Cybersecurity investigated an incident that began with a seemingly harmless email and developed into a multi-stage attack. First, social engineering was employed, followed by a PowerShell dropper, and finally an in-memory infostealer payload was delivered.
This blog post provides a step-by-step deconstruction of this campaign. We will explore how the attackers deceive users, use dynamic code compilation to hide their tools, and leverage in-memory execution to avoid leaving a trace on disk. By dissecting these TTPs, we can better understand the current evasion strategies used by modern malware and identify key opportunities for detection and defence.

The Lure: Abusing User Trust with "ClickFix"

The initial point of entry is user-driven execution of a PowerShell command, delivered through a social engineering tactic known as ClickFix. The user is directed to a malicious URL which presents a fake CAPTCHA. This CAPTCHA instructs the user to execute a series of keyboard shortcuts (Win + R, Ctrl + V, Enter), which runs a PowerShell command previously copied to the clipboard by the website.

Years of legitimate CAPTCHA challenges and security prompts have conditioned the user to believe that they are performing a necessary verification step. In reality, however, they are executing the initial payload of the attacker. This method effectively makes the user an unwitting accomplice in their own compromise. This 'human-in-the-loop' approach is a powerful evasion technique designed to bypass automated security measures. The attack is structured so that neither the email delivering the link nor the URL itself contains any malware as traditionally defined. This effectively renders many email security tools and network filters ineffective, as there is no malicious file to scan or signature to match at the perimeter.

The command itself is short and inconspicuous. First, a hex string is decoded to extract a malicious URL. Next, a PowerShell script is retrieved from this URL via the `Invoke-RestMethod` command and executed directly within the current process. This marks the beginning of the first stage of the infection.

Stage 1: PowerShell Dropper with In-Memory Execution

After initial access, the PowerShell command loads a script that acts as an in-memory dropper. The script hides its own console window using dynamically compiled C# code and Windows API calls. If this initial method fails, the script uses a fallback technique based on assigning a unique window title and hiding it by process lookup.

Slide through the 4 Steps:

Stage 2 – Donut Loader: pivoting to memory

The PowerShell Stager injects a compact .NET loader (internally tagged believemesh) directly into its own powershell.exe process. The primary objective of this loader ist to decrypt a Donut shell-code package, map it in place, and start it without dropping a file or spawning another process. From the first instruction onwards, all unpacking, mapping, and hand-off to the native infostealer happen inside the original PowerShell runtime.

2.1 From wrapper to Donut in four seamless steps

Within the recovered .NET project, the loader's entry point is located in sytuczzzsgolfnefxrmr.cs. The Main() method is concise, with each instruction carefully chosen to ensure seamless execution.

First, the code checks the machine's UI language and exits immediately if it detects a CIS locale, such as Russian (ru), Belarusian (by), or Kazakh (kz). The test is simple `(CultureInfo.CurrentUICulture.TwoLetterISOLanguageName)` but it prevents accidental infections within the threat actor's home region.

If the locale is acceptable, the loader turns its attention to a 1.7 MB ASCII-hex string embedded just beneath the class definition.

Eight bytes near the start of the string serve double duty as an RC4 key. The loader decodes the text, applies RC4 with the extracted key, and then passes the result through an LZMA stream. Within less than a second, a 0x69300-byte buffer containing two concatenated binaries is held: a Donut 0.9.3 stub followed by the final native payload binary.

Now that the package is resident in memory, the loader reserves a block of address space using a direct `P/Invoke` call to *VirtualAlloc*. The package is copied, marked as executable and launched in a new thread, all without touching high-level Win32 APIs. Instead, the loader uses raw syscalls supplied by a small framework that will be revisited in the next subsection. Once the thread has begun executing at the start of the Donut blob, the original loader thread enters an infinite sleep loop so that powershell.exe never terminates.

2.2 Making syscalls the long way round

Relying on ordinary imports, such as NtWriteVirtualMemory, would make the loader vulnerable to user-mode hooks. Therefore, the authors developed their own indirection layer.

These user-mode hooks are techniques employed by EDRs (Endpoint Detection and Response) to intercept API calls in user space before they reach the kernel. These hooks enable suspicious activity, such as memory manipulation or thread creation, to be monitored or blocked by injecting custom code into running processes. If malware uses standard Windows APIs, these hooks can detect it.

To avoid this, the loader uses a custom syscall implementation involving five source files:

- **kzxuzlacbtgdylqcgzjz.cs** queries RtlGetVersion and converts build numbers into tokens such as *WINDOWS_10_22H2.*
- **eiwfcumbsveijxmimpdp.cs** looks up, for example, "NTWRITEVIRTUALMEMORY" in a nested dictionary keyed by the relevant build token, returning the correct syscall number for the current system.
- **zhvzoykgdxolwkvykjfs.cs** stores the classic eleven-byte x64 stub 4C 8B D1 B8 ?? 00 00 00 0F 05 C3.

- **bcfsizsukpmqlglwqnzc.cs** replaces the “??” placeholder with the number obtained from the resolver and returns the ready-to-run stub to the caller.
- **ptieowjalbutmbxcthhv.cs** then flips the stub’s heap page to PAGE_EXECUTE_READWRITE, casts it to the appropriate delegate type, and calls it exactly as if it were managed code.
  The file names were taken from the extracted .NET project.

Since all high-risk operations, such as copying bytes, changing protection and starting a thread, run through these handcrafted syscalls, standard API calls to monitor are never seen by common user-land EDR hooks.

2.3 What really happens inside the Donut stub

Once control passes into the Donut code (verified byte-for-byte as Donut 0.9.3), the stub:

1. Builds its own syscall thunk table, providing subsequent stages with the same hook-evasion privileges enjoyed by the wrapper.
2. Decrypts and inflates the embedded binary using the same RC4 → LZMA routine.
3. It manually reconstructs the binary in memory without invoking the Windows loader.
4. Jumps to the entry point of the binary at `RVA 0xB0F8` on a new thread.
5. Calls Sleep(INFINITE). From this point onwards, the stub performs no additional work and leaves barely a footprint beyond two executable memory regions.

As neither LoadLibrary nor CreateProcess is involved, the binary will never appear in a module list or ImageLoad ETW events, and the intrusion will now live exclusively in volatile memory.

Stage 3: Infostealer Payload

Once Donut hands over execution, the mapped binary reveals itself as a bespoke infostealer that does not align with any catalogued malware family. Static identifiers, hard-coded into multiple payloads, imply that the same toolkit has been used in more than one campaign; however, none of the usual signatures match. Functionally, however, its objectives are unmistakable.

The code focuses almost exclusively on browsers. In the case of Firefox, it accesses the active profile directly, pulling *cookies.sqlite, cert9.db, key4.db,* and *places.sqlite*. These four files alone provide session cookies, saved logins, certificate stores, and full browsing history. Immediately after grabbing these files, the stealer takes a one-shot screenshot of the foreground desktop and stores the PNG image in memory as *Screenshot.png*. This image is sent to the command-and-control (C2) channel without being stored locally.

Chromium-based browsers receive different treatment. Here, the malware opens a WebSocket to the browser’s debugging port and uses native Chrome DevTools commands, such as `Network.getAllCookies`.

The approach yields live, decrypted cookie objects, meaning there is no need to locate the on-disk cookies database or wrestle with DPAPI. The paths hard-coded in the binary reference BinanceBrowser\Default highlight a particular appetite for cryptocurrency sessions.

All collected material is funnelled via a minimalist TCP service listening at 31.177.108.17:12345. The wire protocol begins with plain-text control words (PING, PONG and ACK!), then switches to an XOR-obfuscated data block. Each transmission embeds two constant tags (em1 and a GUID such as `66f89dce-5240-4124-b3ff-54c731c55507`), which allow the operator to track victims across multiple uploads. Filenames (user.txt, server_info.txt, browser paths) are sent in plain text, followed by the encrypted payload of each file.

No persistence mechanism has been implemented, nor is any residue left on the disk. After the final ACK!, the process sleeps briefly, releases its working buffers and exits, leaving only the RX memory region created by Donut and a terminated thread in volatile memory.

Conclusion

This campaign serves as a stark reminder of how far modern malware operations have evolved. From a fake CAPTCHAs that trick users into executing malicious code, to a multi-stage loader that hide entirely in memory and bypass virtually all traditional defences, this attack showcases the cutting edge of stealth and sophistication.

While attribution remains uncertain, the infrastructure, coding patterns, and reuse of modules across campaigns suggest that this is the work of an experienced and well-resourced threat actor.

At Swiss Post Cybersecurity, we continuing to monitor this threat and similar campaigns closely. The message is clear: defenders must recognise that traditional IOCs and file-based scanning alone are no longer sufficient. Behavioural detection, memory analysis, and user education are mandatory components of any effective security strategy.

As attackers raise the bar, so must we.

You can't defend. You can't prevent. The only thing you can do is detect and respond.
– Bruce Schneier

IOC’s

IOC	Type	Name	Context
admilzsolutions.co[.]ke	domain		ClickFix Subdomain
rtjj[.]store	domain		Payload delivery
rtjj[.]store/f/h	url		Stage 1 URL
rtjj.store/amountatom/believemesh	url		Stage 2 URL
31.177.108[.]17	ipv4-addr		Stealer C2 server IP
12345	port		Stealer C2 server port
9d9d5bebe19a536491720424d69cbbc808e96f5dca9d18e0133ec45bdd39092e	SHA-256	believemesh	DONUTLOADER
1e52ed52921eedcd858cc0d0ed9164d70840c742ddbad7d3fd65666d24c40cda	SHA-256	h(.ps1)	Stager
e0038e6450f74f388e8952e1f7baa15de4631ed99568b0bddf99ab336d7d6343	SHA-256		DONUTINJECTOR
5a65621791cdcbce3cd1ee200454bec87f99a7e46d2aa0ffcb8e15870e378ecd	SHA-256		StealerExecutable

References

https://github.com/TheWover/donut
https://github.com/volexity/donut-decryptor
https://3xperience.substack.com/p/bite-sized-insights-diving-into-donut?utm_campaign=post&utm_medium=web
https://malware.tech/posts/unpacking-shellcode-loaders/#thewover-s-donut

---