FLARE-On 13 Challenge 1: FLAreCAPTCHA

By Nick Harbour (@nickharbour)

Overview

The `flarecaptcha.html` challenge is a client-side obfuscation puzzle presented as a web page. The goal is to bypass a custom reCAPTCHA verification widget to uncover the hidden flag: **`triskaidekaphobia@flare-on.com`**.

> tris·kai·dek·a·pho·bi·a
>
> /ˌtriskīˌdekəˈfōbēə/
>
> noun
>
> the intense, irrational fear of the number 13.

Initial Inspection

Opening `flarecaptcha.html` in a web browser presents the player with an extremely authentic-looking Google reCAPTCHA v2 verification widget. The visual styles, SVG animations, and layout precisely mirror standard web verification dialogs.

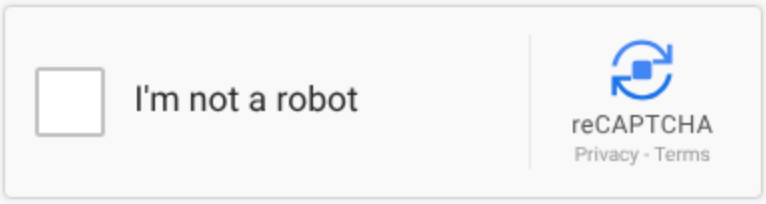

Figure 1: Initial Page Load

Attempting to interact with the widget normally by left-clicking the "I'm not a robot" checkbox triggers a brief spinning verification wheel before resulting in a failed state with an animated red "X" and a humorous detection alert: “Robot, or unskilled reverse engineer detected.”

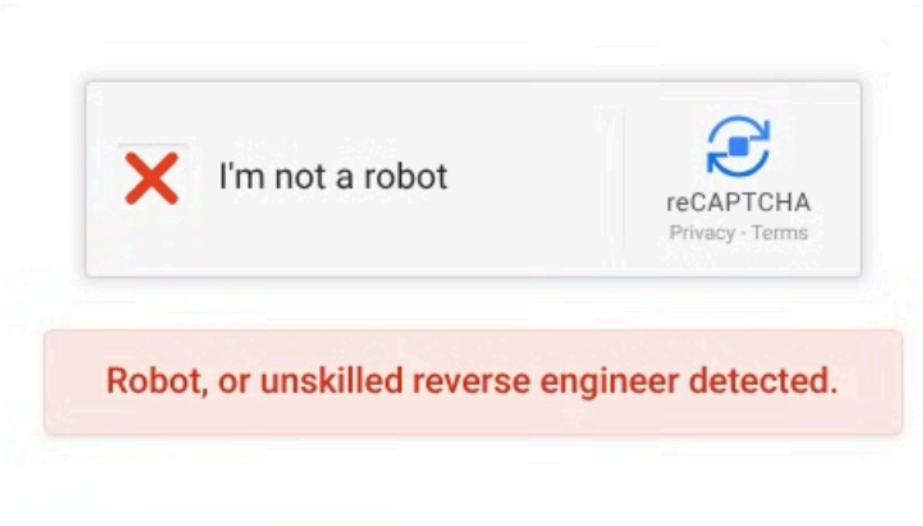

Figure 2: Unsuccessful Verification

Source Code & Obfuscation Analysis

Examining the underlying JavaScript reveals intentional obfuscation utilizing misleading variable and identifier names designed to confuse static analysis (e.g., framing cryptographic variables as UI layout metrics like `widgetUpperBoundsPx` and `displayPaddingPx`).

Event Handling & Button Verification

The core verification routine binds a `mousedown` event listener to the interactive widget area (`flarecaptcha.html:L374-L442`):

```javascript
interactiveArea.addEventListener('mousedown', (e) => {
    if (isVerified || isFailed || isVerifying) return;

    // Ensure click occured in captcha box with 5,0 and 1,1 layout grids
    if (e.button === (Math.pow(5, 0) + Math.pow(1, 1))) {
        // ... success verification routine ...
```

Evaluating the conditional expression: `Math.pow(5, 0) + Math.pow(1, 1) = 1 + 1 = 2`, because 5^0 = 1 and 1^1 = 1.

In standard DOM events, `e.button === 2` corresponds directly to a **secondary button click (Right-Click)**.

Conversely, clicking with the standard primary button (`e.button === 0`) routes into a trap branch that automatically invokes `triggerFailure()` after 1 second:

```javascript
else if (e.button === 0) {
    isVerifying = true;
    // Spin briefly (1s) then fail test with red X
    setTimeout(() => {
        if (!isVerified && !isFailed) triggerFailure();
    }, 1000);
}
```

The "Hold" Constraint

Once a right-click initiates the verification routine, a timer `rightClickHoldTimer` is scheduled for **2000 milliseconds (2 seconds)**. Meanwhile, global `mouseup` and `mouseleave` handlers actively monitor the user's cursor:

```javascript
const cancelHold = () => {
    if (rightClickHoldTimer && isVerifying && !isVerified && !isFailed) {
        triggerFailure();
    }
};

window.addEventListener('mouseup', (e) => {
    if (e.button === 2) cancelHold();
});
```

If the player releases the right-click button or moves the cursor out of the interactive boundary before the 2-second duration lapses, verification aborts and triggers the failure banner.

How to Solve the reCAPTCHA

Holding right-click for the required duration (2 seconds) triggers the victory condition, presenting an elegant green checkmark and rendering the decrypted flag inside an interactive copy widget:

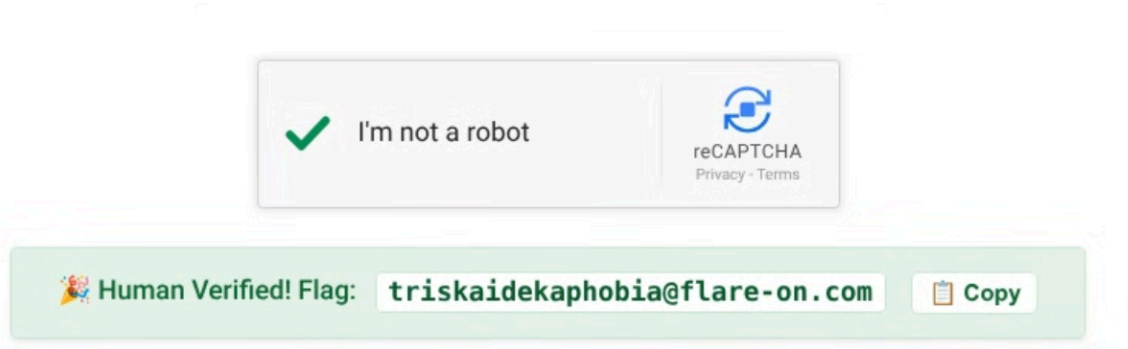

Figure 3: Successful Verification

At this point, the final flag is revealed to the player and they can consider this one solved. Continue reading for details about how the challenge decrypts the flag for its presentation as shown in this figure.

Flag Construction & Cryptographic Reverse Engineering

When the 2-second hold completes successfully, the script derives decryption keys from runtime metrics to reconstruct the embedded flag string.

The Array Proxy Layer

The underlying ciphertext is stored as an array of byte values:

```javascript
const recaptchaLogoBitmap = [123, 5, 102, 4, 100, 22, 102, 19, 106, 28, 110, 7, ...];
```

Accessing this array is mediated via a JavaScript `Proxy` object (`recaptchaLogoBitmapProx`) that applies an automatic XOR transformation against the document's encoding shift (`flarecaptcha.html:L364-L372`):

```javascript
const envShift = document.characterSet.length; // "UTF-8".length == 5
return target[prop] ^ envShift;
```

Every raw byte fetched from the bitmap is immediately XORed by 5.

Deriving Layout Metrics (1337 Magic Number)

The script calculates an upper bound pixel metric utilizing a timestamp delta and function serialization lengths:

```javascript
const displayPaddingPx = Math.round((errorLogTime - errorLogStartTime) / 1000); // ~2 seconds
const widgetLength = resetWidget.toString().length; // 222 bytes
const widgetUpperBoundsPx = (displayPaddingPx * (widgetLength + 1115)) & 0xFFFF;
```

With `displayPaddingPx` resolving to `2` and `resetWidget` serialization lengths resolving to `222`, the derivation yields the classic hacker magic number: 222 + 1115 = 1337 widgetUpperBoundsPx = 2 * 1337 = 2674 (0x0A72)

Splitting `0x0A72` into high and low extraction bytes:

- `wupperLeft` (High Byte) = `0x0A` (10 in base10)
- `wupperRight` (Low Byte) = `0x72` (114 in base 10)

Final Flag Decryption

The final flag string (`displaySurfaceMap`) maps across the proxied array, applying an alternating XOR key based on parity:

```javascript
const displaySurfaceMap = recaptchaLogoBitmapProx.map((b, i) =>
    String.fromCharCode(b ^ (i % 2 === 0 ? wupperLeft : wupperRight))
).join('');
```

Executing this transformation across the ciphertext successfully outputs the target string: `triskaidekaphobia@flare-on.com`.