Flare-On 11 Challenge 4: FLARE Meme Maker 3000

By Moritz Raabe (@m_r_tz)

Overview

This FLARE On challenge is an HTML page containing an obfuscated JavaScript script. To solve it, reverse engineers need to deobfuscate and understand the embedded script.
This writeup focuses on the challenge key components and does not describe all functionalities in detail. The main analysis tools we use are browser Developer tools and CyberChef – both run within FLARE VM.

Basic Analysis

The provided file contains HTML, CSS, and obfuscated JavaScript code. Opening the page shows the content shown in Figure 1 allowing users to create memes.

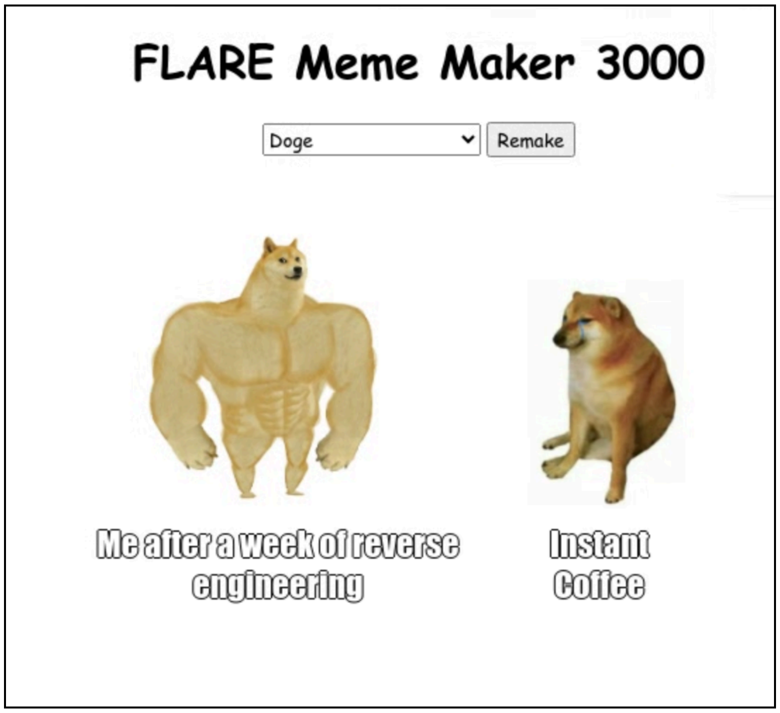


Figure 1: running the challenge file in a browser

JavaScript Deobfuscation

The file's HTML and CSS code doesn't provide useful information. So, the first order of business is to understand the obfuscated JavaScript code. After beautifying the code (e.g. using the CyberChef recipe JavaScript Beautify), we can make a bit more sense of the script. The experienced eye will also spot that the code appears to be the output of JavaScript obfuscator – an open source and commonly used obfuscator.

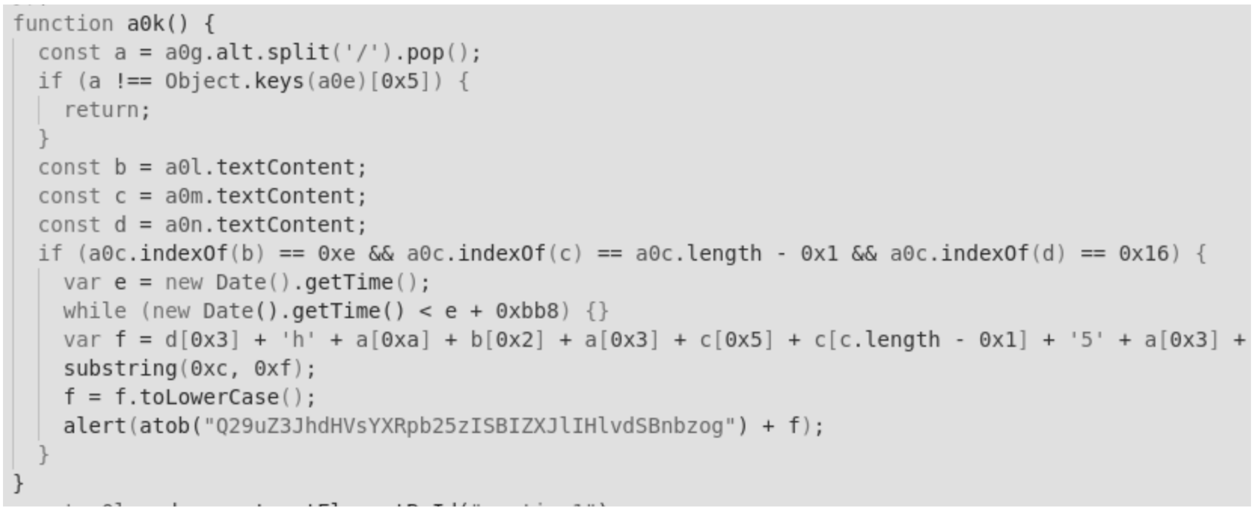
Luckily for us, there also exists an open source deobfuscator project: Obfuscator.io Deobfuscator (already part of FLARE-VM). We extract the obfuscated code into a separate file and run the deobfuscator tool on it:

```
Unset
obfuscator-io-deobfuscator obfuscated.js -o deobfuscated.js
```

The tool does a great job of recovering JavaScript code making it much easier to understand. Granted, this small challenge can also be analyzed without deobfuscating the entire script. Note that the JavaScript contains an irrelevant Base64 encoded binary meant to briefly distract the overambitious analyst 🙂.

Analyzing the Deobfuscated Code

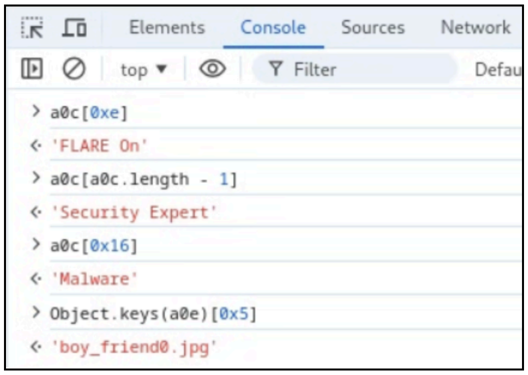
The deobfuscated code starts with several objects defining script data – including the string captions used to create memes. The end of the script is where the functions are defined. One of them contains a Base64 encoded string (`Q29uZ3JhdHVsYXRpb25zISBIZXJlIHlvdSBnbzog`) that decodes to "`Congratulations! Here you go: `". Just before that, a string is concatenated if certain conditions are met (see Figure 2).



```
function a0k() {
  const a = a0g.alt.split('/').pop();
  if (a !== Object.keys(a0e)[0x5]) {
    return;
  }
  const b = a0l.textContent;
  const c = a0m.textContent;
  const d = a0n.textContent;
  if (a0c.indexOf(b) == 0xe && a0c.indexOf(c) == a0c.length - 0x1 && a0c.indexOf(d) == 0x16) {
    var e = new Date().getTime();
    while (new Date().getTime() < e + 0xbb8) {}
    var f = d[0x3] + 'h' + a[0xa] + b[0x2] + a[0x3] + c[0x5] + c[c.length - 0x1] + '5' + a[0x3] +
    substring(0xc, 0xf);
    f = f.toLowerCase();
    alert(atob("Q29uZ3JhdHVsYXRpb25zISBIZXJlIHlvdSBnbzog") + f);
  }
}
```

Figure 2: Deobfuscated JavaScript code to perform condition check and string concatenation

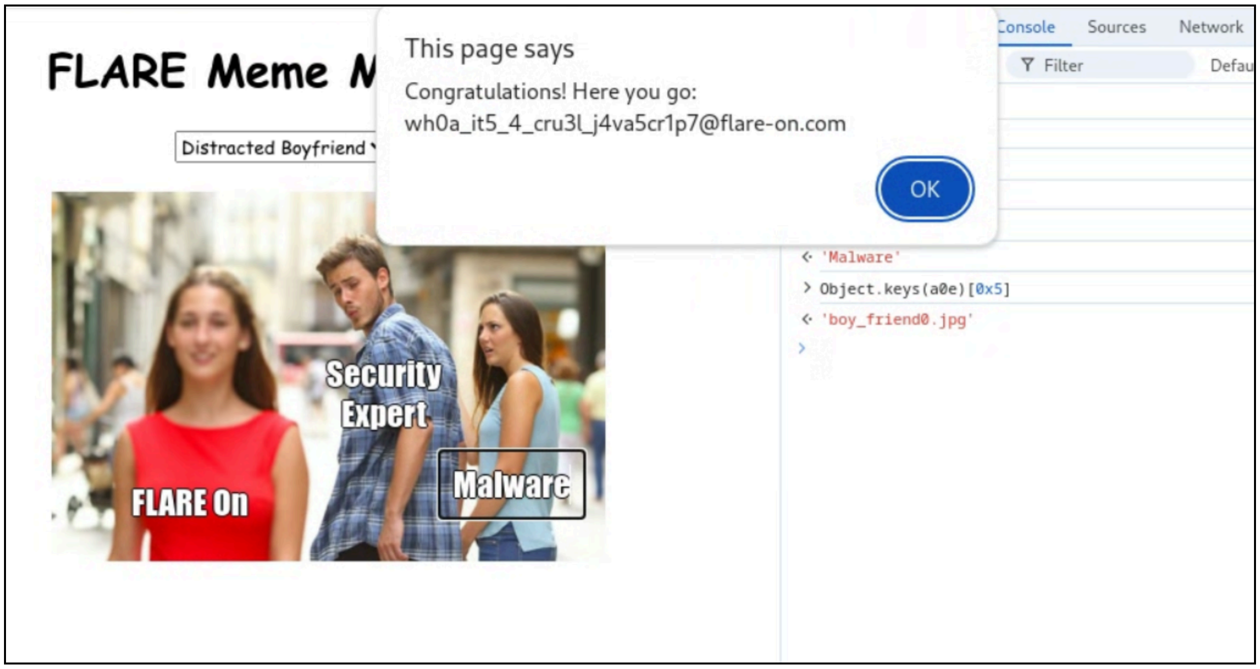
We use the browser Developer tools to recover the relevant data as shown in Figure 3.



```
> a0c[0xe]
<· 'FLARE On'
> a0c[a0c.length - 1]
<· 'Security Expert'
> a0c[0x16]
<· 'Malware'
> Object.keys(a0e)[0x5]
<· 'boy_friend0.jpg'
```

Figure 3: Using the Developer tools to recover relevant data

Piecing this information together with the HTML definitions, we can recreate the expected meme data that gets verified here. As shown in Figure 4, after selecting the right meme template and entering the recovered captions, we see an alert box providing the challenge flag: `wh0a_it5_4_cru3l_j4va5cr1p7@flare-on.com`.



This page says
Congratulations! Here you go:
wh0a_it5_4_cru3l_j4va5cr1p7@flare-on.com

```
<· 'Malware'
> Object.keys(a0e)[0x5]
<· 'boy_friend0.jpg'
>
```

Figure 4: Alert box with the challenge flag after recreating the expected meme

Final Flag

```
Unset
wh0a_it5_4_cru3l_j4va5cr1p7@flare-on.com
```