

Challenge 4 ("Meme Maker 3000")

Description

“ You've made it very far, I'm proud of you even if no[]one else is. You've earned yourself a break with some nice HTML and JavaScript before we get into challenges that may require you to be very good at computers.

Writeup

Wow, thanks for those kind words, Flare-On. Anyways... we are given a zip with a single file, `mememaker3000.html`. The webpage allows you to select a meme format and randomly generate captions from a (presumably) predefined set of strings. Upon inspecting the contents of the file, it is immediately obvious that the JavaScript application logic is obfuscated.

The first idea I had was to use some of the many tools for deobfuscating javascript code that are available online. This helped make the code a bit more readable:

The core element of the obfuscation logic is a function called `a0a`, which simply returns a reference to a huge array of what are initially random-looking strings. Another function, declared with the name `a0b`, but also aliased to `a0p`, accepts an index `i` and returns the element at index `i - 475` of the aforementioned array.

This function is used throughout the rest of the code as a means of obfuscating strings, hiding the names of methods, etc. For instance, in the following statement, the function is used to hide the HTML attribute name `"alt"` of the object referenced by `a0g` and the method name `"pop"` on the array returned by `split`:

```
const t = a0p, a = a0g[t(2589)].split("/")[t(2024)]();
```

(note that in Javascript, `a.b()` is equivalent to `a["b"]()`).

Finally, the application data is defined:

- The array `a0c` contains the set of meme captions,
- `a0d` is an array containing metadata for positioning the captions in each image / meme format,
- `a0e` is a dictionary (or JS object) containing image filenames as keys and corresponding binary data as values,
- `a0g`, `a0h`, ..., `a0j` and `a0l`, ..., `a0n` are references to DOM objects selected using the `document.getElementById` API.

My first idea was to simply copy the definition of `a0a` into a blank JS console (or, since that almost froze my whole laptop, create another HTML file with just the definition of `a0a` and open that in the browser) and use it to evaluate the different calls to `a0p` manually. It turned out, however, that the

string array is manipulated by the script at runtime by an anonymous function at the very start of the script, the behaviour of which can be expressed as follows.

```
while (true) { const c = a0a();          try {          const d = parseInt(a0b(55277)) / 1 *
(parseInt(a0b(14365)) / 2)              + -parseInt(a0b(68223)) / 3 * (-
parseInt(a0b(90066)) / 4)              + parseInt(a0b(76024)) / 5
+ -parseInt(a0b(73788)) / 6              + parseInt(a0b(58137)) / 7 *
(parseInt(a0b(59039)) / 8)              + -parseInt(a0b(97668)) / 9
+ parseInt(a0b(26726)) / 10 * (-parseInt(a0b(11835)) / 11);          if (d ===
356255)                                break;          else                                c.push(c.shift());          }
catch (e) {                            c.push(c.shift());          }}
```

Eventually, I figured the simplest way would be to just read the deobfuscated values of every variable directly from the JS console on the open webpage after the script was loaded and all top-level statements executed (although of course as a *security expert*, I have a certain disdain for this "just run it and see what it does" strategy).

The crucial bit to solving the puzzle, however, turned out to be at the very end of the file. There, another function, `a0k`, is defined, and then, an `EventListener` for the "keyup" event is added onto `a0l`, the DOM object containing the first caption, with a handler that calls `a0k` with no arguments.

Analyzing and deobfuscating `a0k` gives the following outline:

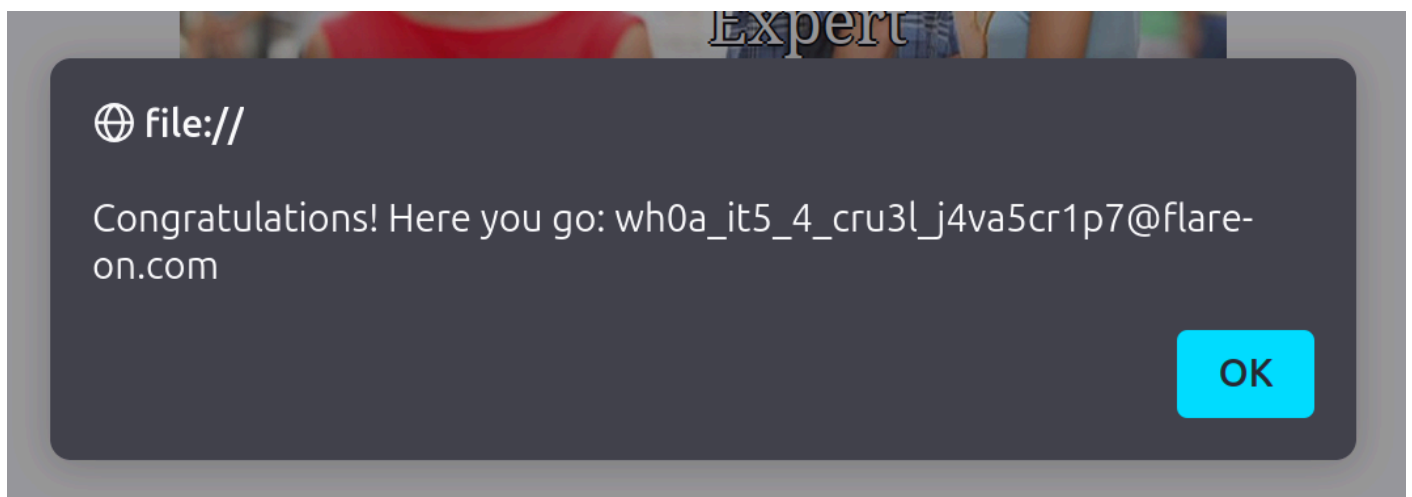
```
function a0k() {          const a = a0g["alt"].split("/").pop(); if (a === "boy_friend0.jpg")
return; const b = a0l.textContent,          c = a0m.textContent,          d =
a0n.textContent;          if (          a0c.indexOf(b) === 14          && a0c.indexOf(c) =
25 // a0c.length - 1          && a0c.indexOf(d) === 22 ) {          // ... }}
```

In other words, when the selected meme format is the `boy_friend0.jpg` image and the three captions have specific values, some code is executed. It is only reasonable to suspect that the contained code decodes the flag and prints it.

Running these four expressions in the JS console,

```
a0l.textContent = a0c[14]a0m.textContent = a0c[25]a0n.textContent = a0c[22]a0k()
```

(or omitting the call to `a0k` and selecting the first caption and pressing any key) triggers the following alert.



(And to be fair, the meme itself is pretty funny, too.)



Appendix: Getting trolled by the authors (again)

The writeup above explains, at least in my opinion, the correct way to approach and solve this challenge. That, however, was absolutely not my experience — I was stuck on this challenge for almost a day. Let me explain why.

"... the simplest way would be to just read the deobfuscated values of every variable directly from the JS console ..."

When looking at the deobfuscated arrays and objects, I noticed something peculiar — the `a0e` object, which mapped image names to image files, contained 9 files, while the app only offered 8 meme formats to choose from. I thought, if there was a hidden image, it might lead me to the flag.

Judging by the filenames, the object key `fish.jpg` was the only one that didn't correspond to any of the meme formats offered by the application. Therefore, I opened the JS console and typed `a0e["fish.jpg"]`. The result was somewhat unexpected — instead of seeing the string `data:image/jpeg;base64,` and a long base64 string, like with the other images, the mime type of `fish.jpg` was declared as `binary/red`. Intrigued, I copied the base64 encoded data, decoded it to a file, and ran the UNIX `file` command on it.

```
fish.jpg: PE32 executable (console) Intel 80386 (stripped to external PDB), for MS Windows, 9 sections
```

So we have a Windows binary. Interesting. Maybe this challenge wasn't so much about easy HTML and JavaScript after all.

I copied the binary to my Windows VM and took a look in IDA. Straight away, I noticed a string in the `.rdata` section that said "Oh, hello! You found something here.". Clearly, this must have been the next step to solving the Meme Maker challenge. This time, the binary was a standard C executable, albeit a stripped one, so I manually found and analyzed the C `main` function. When the executable was run without any modification, it simply printed two lines of text.

Unfortunately, this file is not relevant AT ALL! Have fun with FLARE-ON this year!

Of course, based on the string I saw, I didn't believe that this was all there was to this binary. After all, Flare-on is a CTF aimed mainly towards malware analysts, and if a malware sample told me that "this file is not relevant AT ALL", I certainly wouldn't have taken its word for it. In the main function, I found that 7 different strings were passed to the same function, which I assigned the working name `conditional_puts`, along with a single byte as another argument. Depending on whether the value of this byte was {either `0x50` or `0x25`} or something else, the string was either printed or not printed. (I'm actually simplifying this, the byte value wasn't just there for decision making, it probably served as a key to decode/decrypt the strings, as the one mentioned earlier was the only one stored in plaintext and called with `0x00`, but I (luckily) didn't explore this further. Similarly, I suspect the explicit check for the `0x25` value indicated that only a part of the string passed along with that argument should be printed.)

So, out of 7 strings in total, only 2 were printed. The logical next step was to see what would happen if the other 5 were printed. For this, I simply patched a single byte in the binary at offset `0x8a4`, replacing a `jnz` opcode (`0x0f 0x85`) with a `jz` opcode (`0x0f 0x84`). To my disappointment, the result was the following.

Oh, hello! You found something here. Really, don't waste your time here. Just kidding: here is the flag... Just kidding again... there's nothing exciting to be found here. You don't believe me? Fair enough. You should have trusted me though. Have fun with FLARE-ON this year.

Being completely honest, at this point, I was even more convinced that the flag was supposed to be obtained from this binary than before, and also kind of mad at the authors for trolling me so much. But since I was running out of ideas about where to look for it, I decided to ask a friend (who had completed this challenge already) if I should really be looking in the binary, and he nudged me back onto the right track.

Even though this year's Flare-On is my first time competing, I didn't want to get any help from other people and figure everything myself. At the time of writing this, I'm still trying to do that. However, in this particular instance, I found that this was not so much an issue of reverse engineering skill, but rather knowing how the Flare-On CTF works and how likely it is that the authors are just trolling me. Of course, in a real-world malware analysis scenario, psychological warfare and trolling the analyst is perfectly ordinary, but then again, in a real-world malware analysis scenario, I would not trust a binary if it told me that there's *really* nothing interesting in there and that I shouldn't waste my time analysing it.

🕒 Revision #1

★ Created 23 December 2024 21:14:00 by Annatar

✎ Updated 23 December 2024 21:23:28 by Annatar