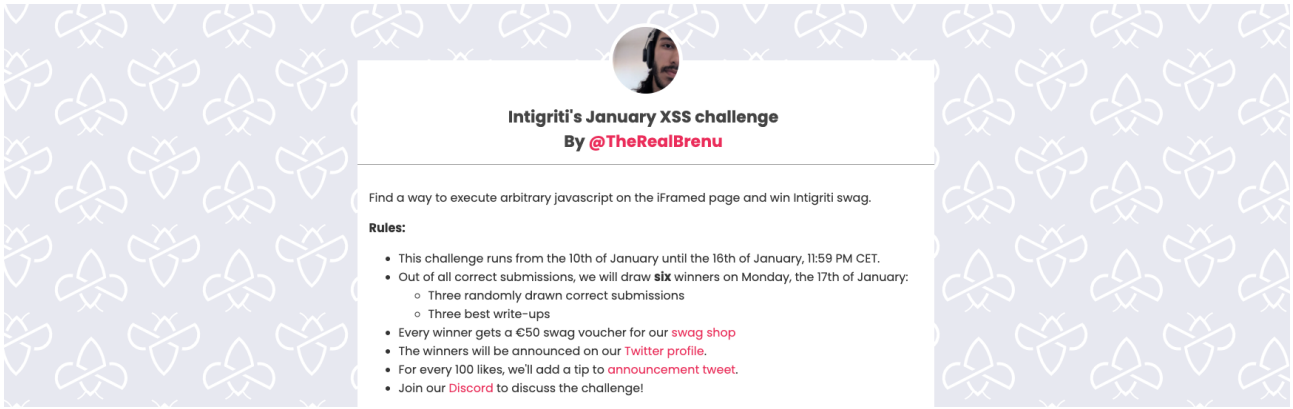
Intigriti January 2022 Challenge: XSS Challenge 0122 by TheRealBrenu

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



Rules of the challenge

- Should work on the latest version of Firefox **AND** Chrome.
- Should execute alert (document.domain).
- Should leverage a cross site scripting vulnerability on this domain.
- Shouldn't be self-XSS or related to MiTM attacks.

Challenge

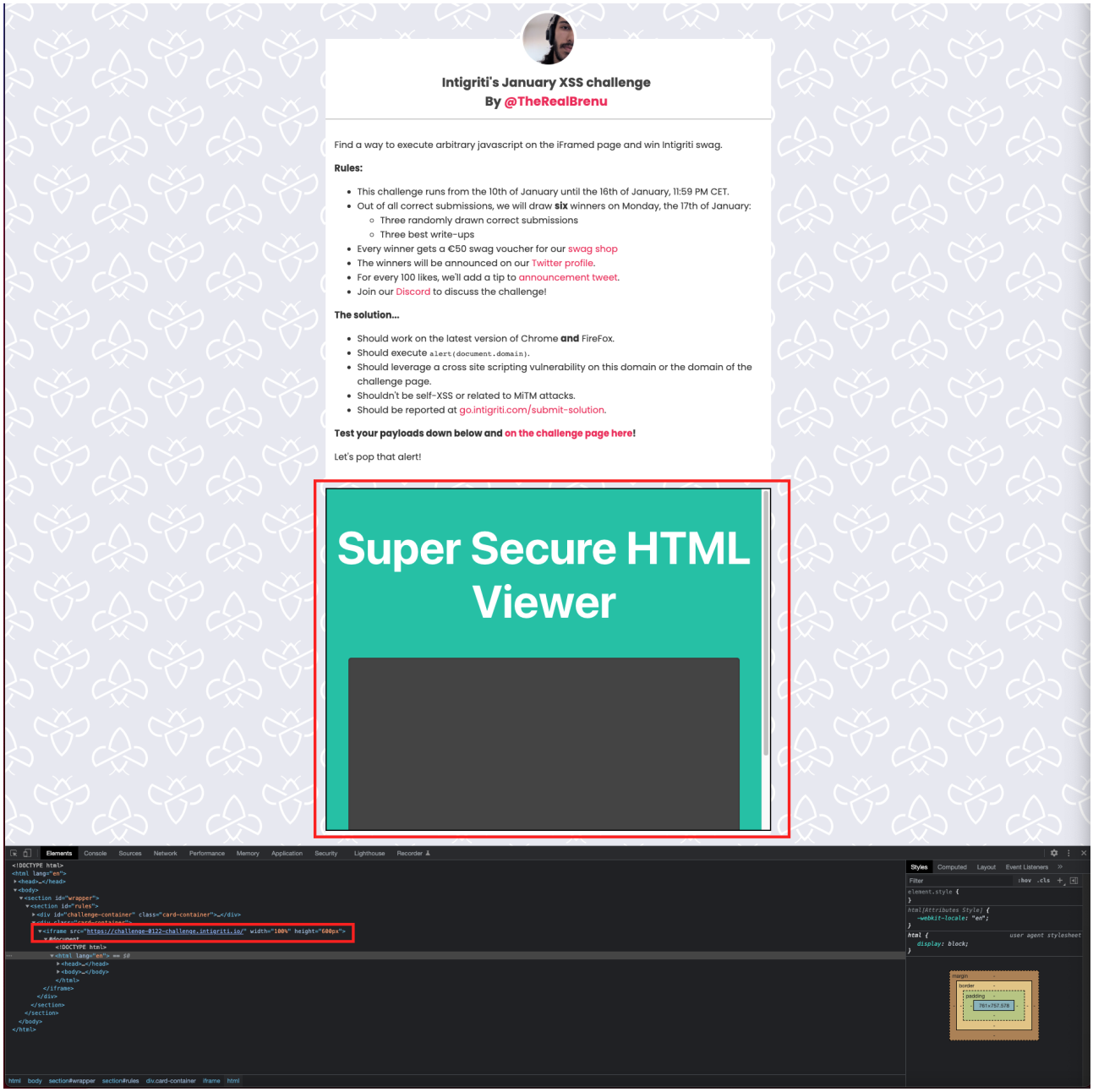
To simplify a victim needs to visit our crafted web url for the challenge page and arbitrary javascript should be executed to launch a Cross Site Scripting (XSS) attack against our victim.

The XSS (Cross Site Scripting) attack

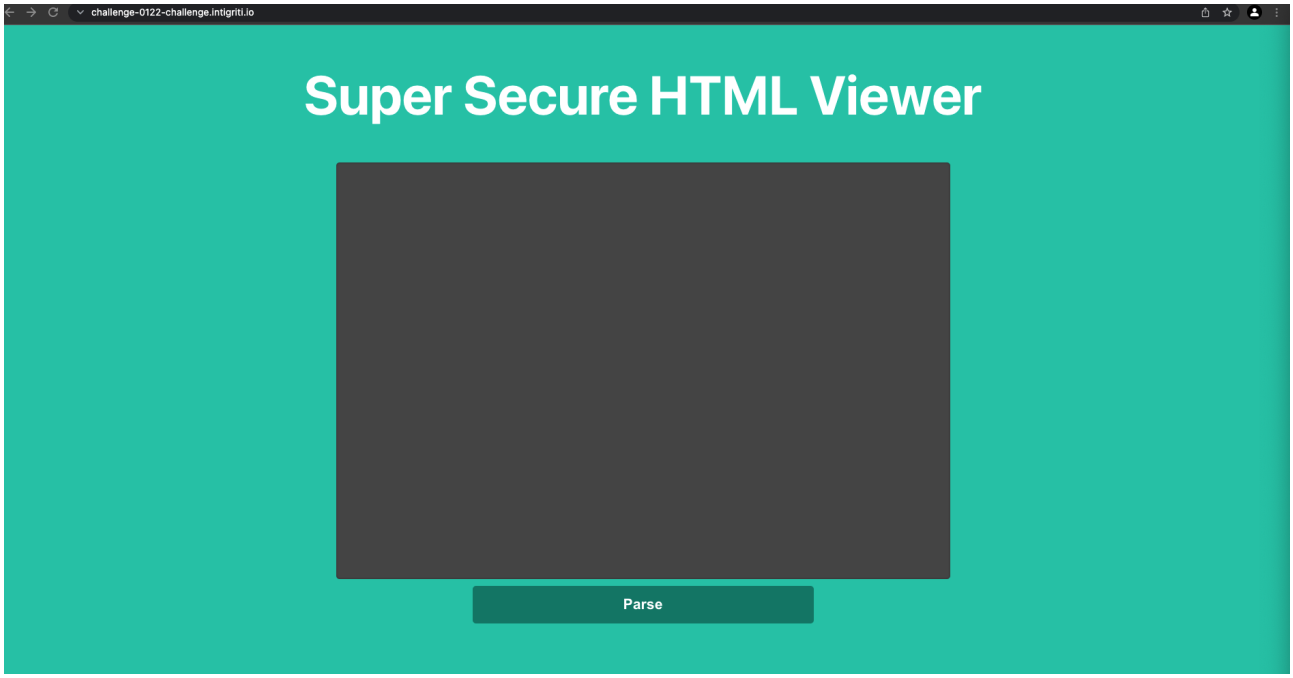
Step 1: Recon

As always we try to understand what the web application is doing. A good start for example is using the web application, reading the challenge page source code and looking for possible input.

The challenge page itself shows an embedded iframe with a “Super Secure HTML Viewer”.

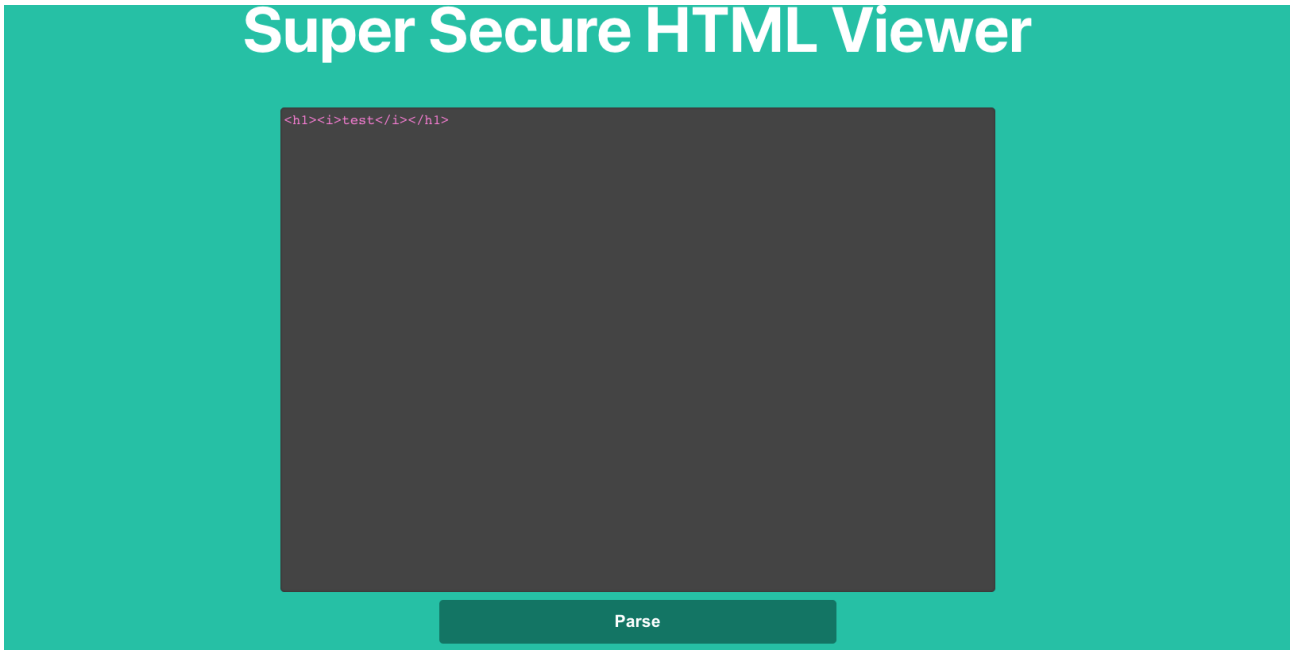


To make our life a bit easier we can go directly to the page loaded by that iframe: *https://challenge-0122-challenge.intigriti.io/* which then only shows the Super Secure HTML Viewer" itself.

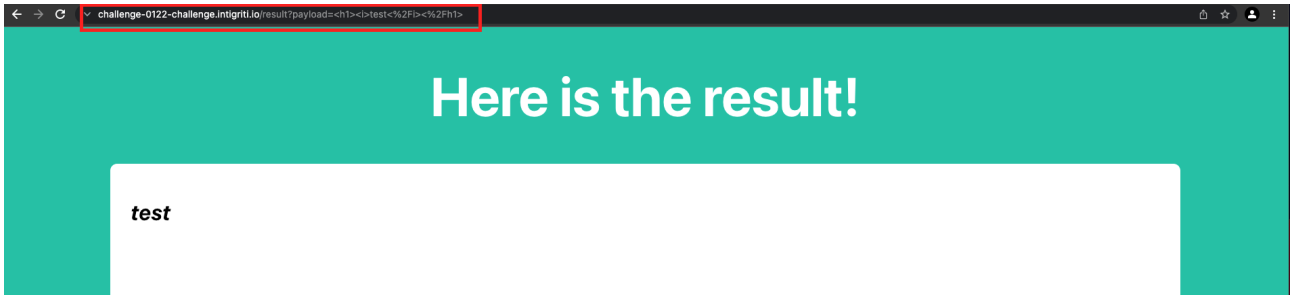


We are up against a HTML viewer so a first thing we can do is see if our HTML viewer is actually parsing our input. Let's give it a try with some very easy HTML code *<h1>test</h1>* for example.

Enter the input *<h1>test</h1>* and click the "Parse" button.



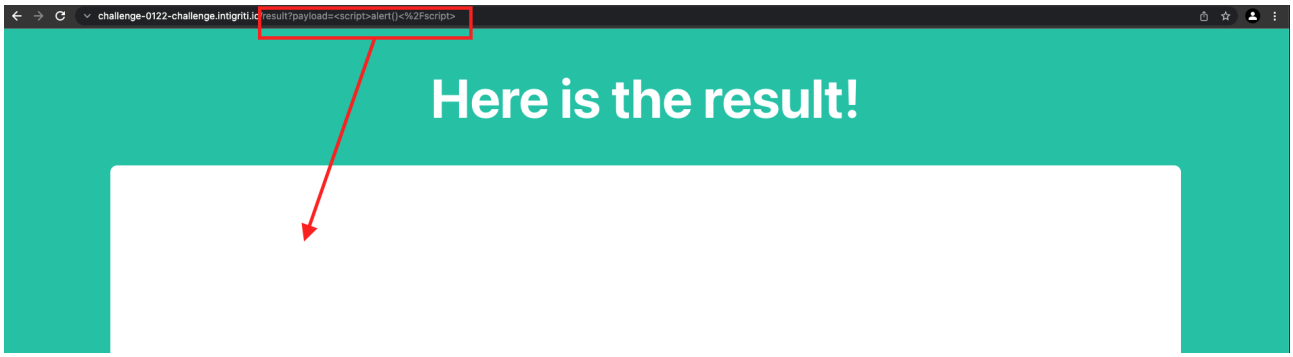
That worked fine. Our input text “test” is parsed in italic and a bit bold from the heading tag. Looking at the browser address bar this already reveals something. Once parsed we can see a “payload” parameter being used.

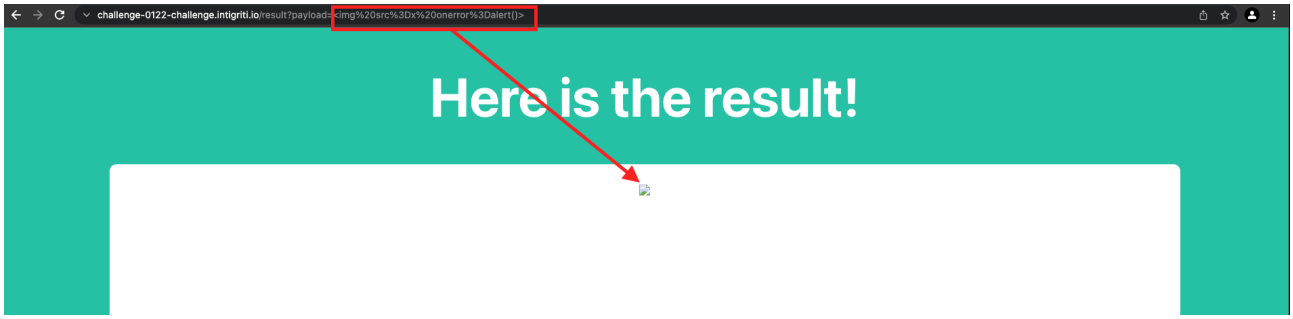


At this point we notice HTML being parsed then my idea is simple try to input javascript or an XSS vector based on HTML context. The payloads are URL encoded in the browser address bar.

	XSS Payload	URL encoded payload
Javascript alert box	<script>alert()</script>	<script>alert()<%2Fscript>
HTML context XSS payload	<img src=x onerror=alert()>	<img%20src%3Dx%20onerror%3Dalert()>

The javascript payload results in nothing shown and no alert box. The XSS vector that should fire in HTML context seems to get parsed as we can see the image symbol reflected but the alert box is also not firing so the XSS is not executed:

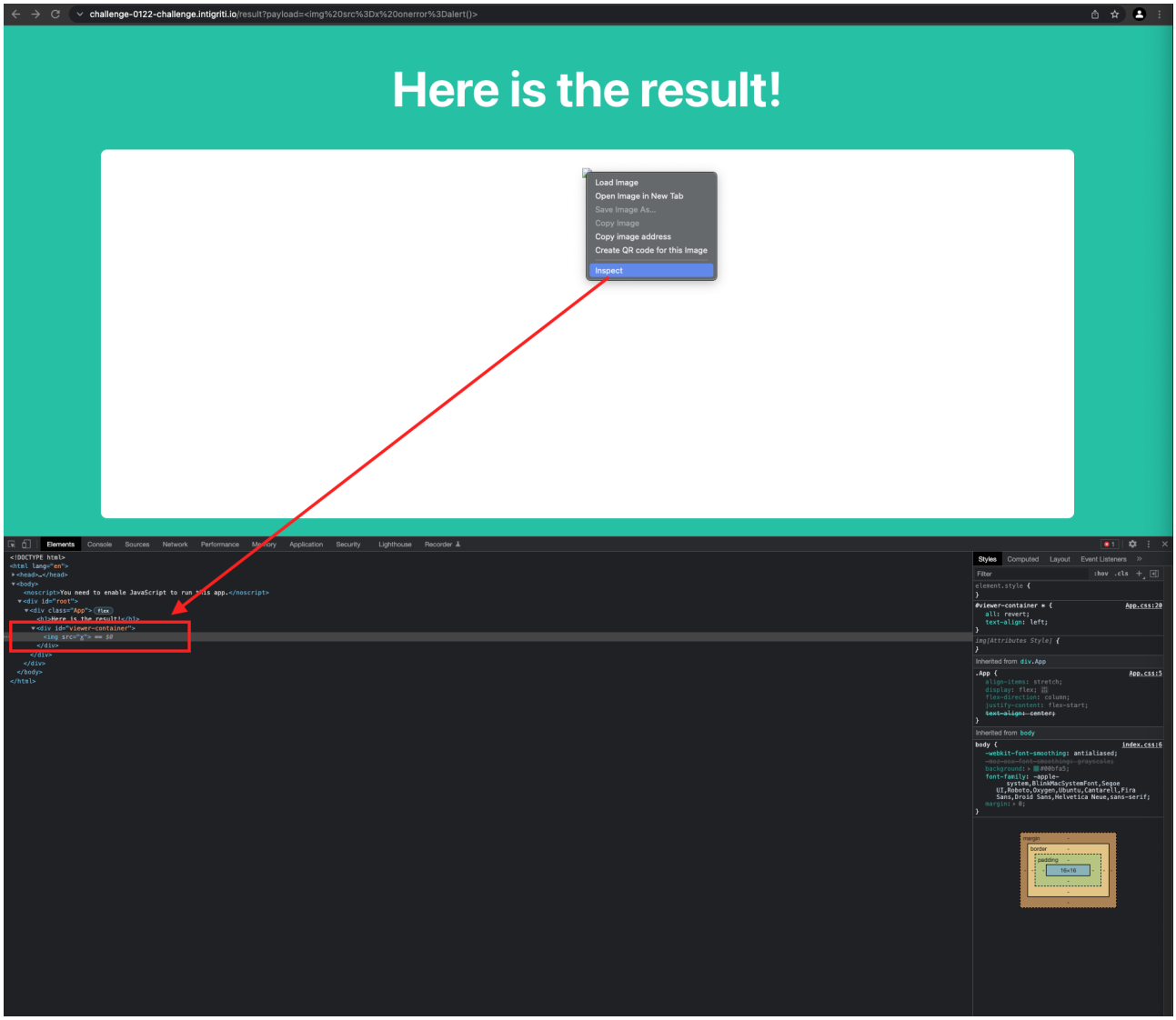




So something is blocking us from parsing javascript and once we try to parse HTML that uses an event attribute like “onerror” this seems not to execute. We need to take this a step further and have a look at the source code if we want to get an XSS payload to fire.

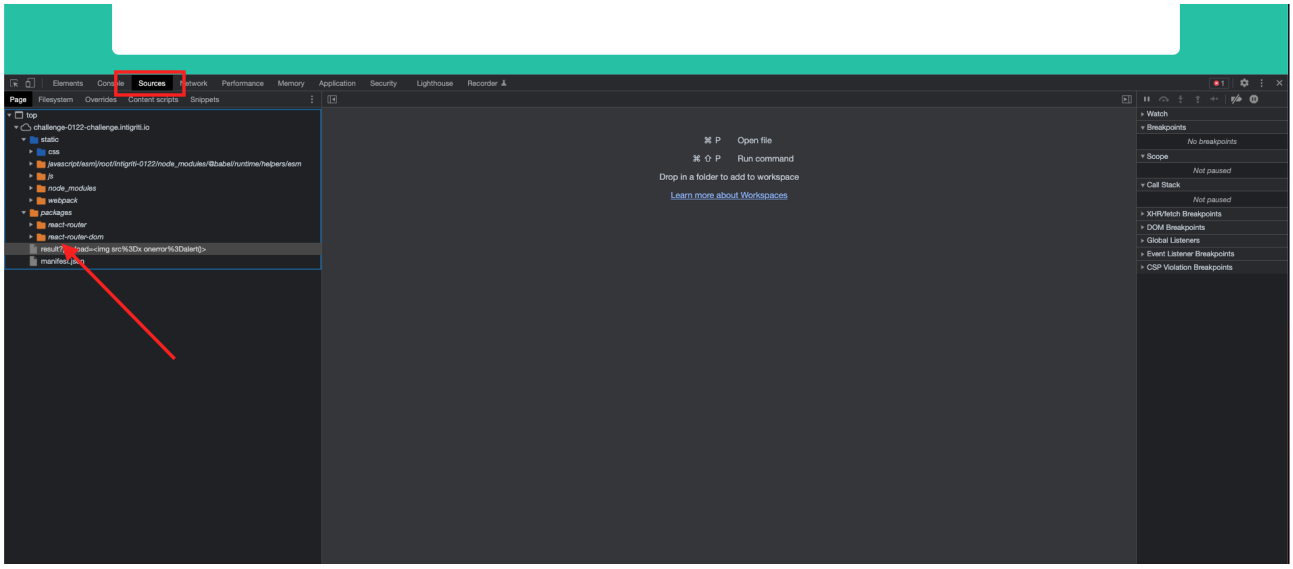
Our HTML context XSS payload *<img src=x onerror=alert()>* seems to get parsed for a part so we can inspect this via the developer tools and check how it is exactly reflected in the source code.

Right click on the reflected image shown and choose “Inspect”



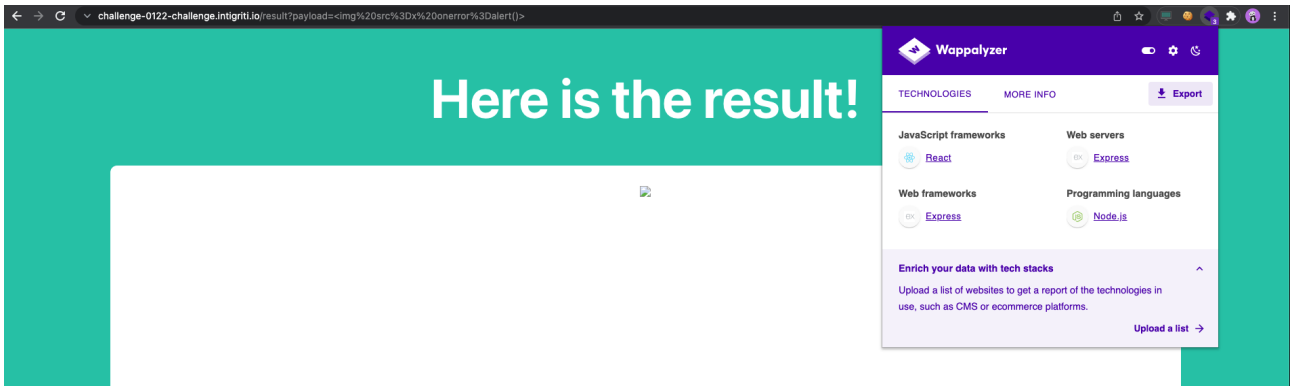
The event attribute “onerror” is clearly missing in the source code. Something in this web application is filtering the input for safety reasons. We need to figure out what it is and of course try to bypass this “safety” mechanism for our XSS attack to fire.

As we are now in the developer tools we can have a look at the other sources of this web page via the “Sources” tab.



Ok this could look overwhelming with many folders and subfolder but a quick glance at these folders reveals we are facing a web application built with the React library (https://reactjs.org/).

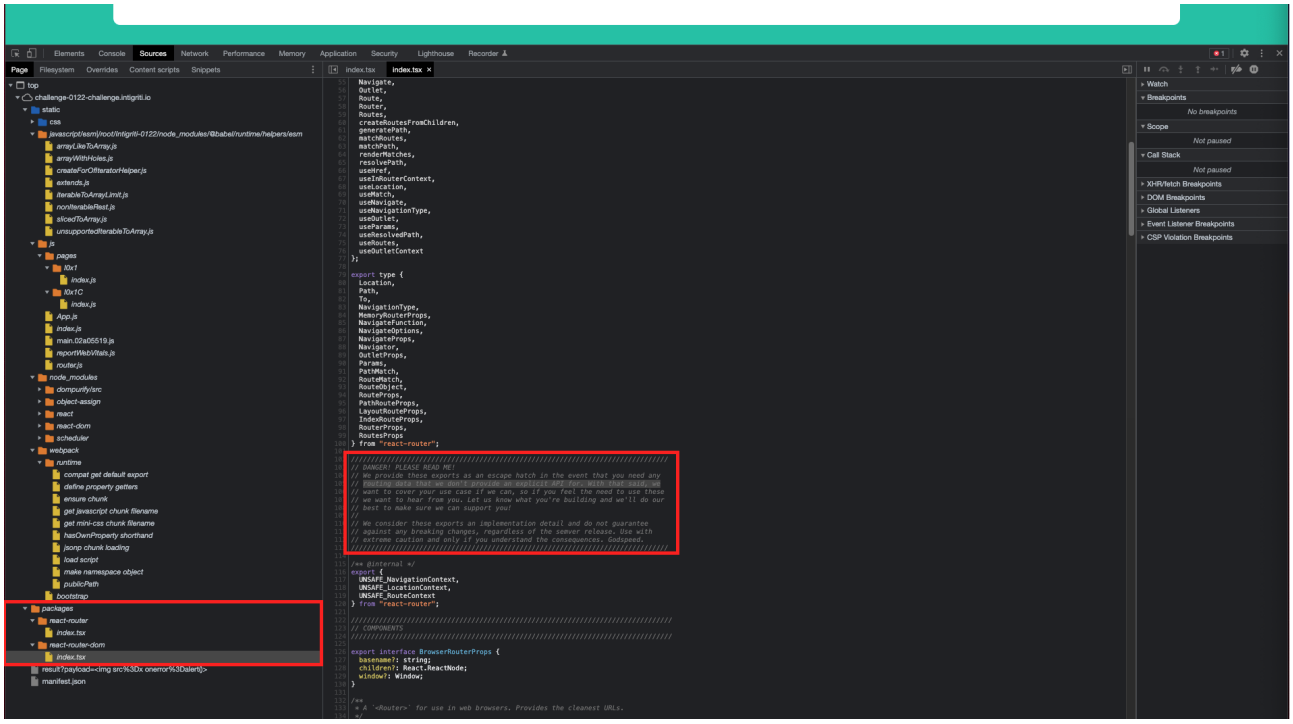
Another way to get this information is via browser plugins like “Wappalyzer” for example in Chrome:



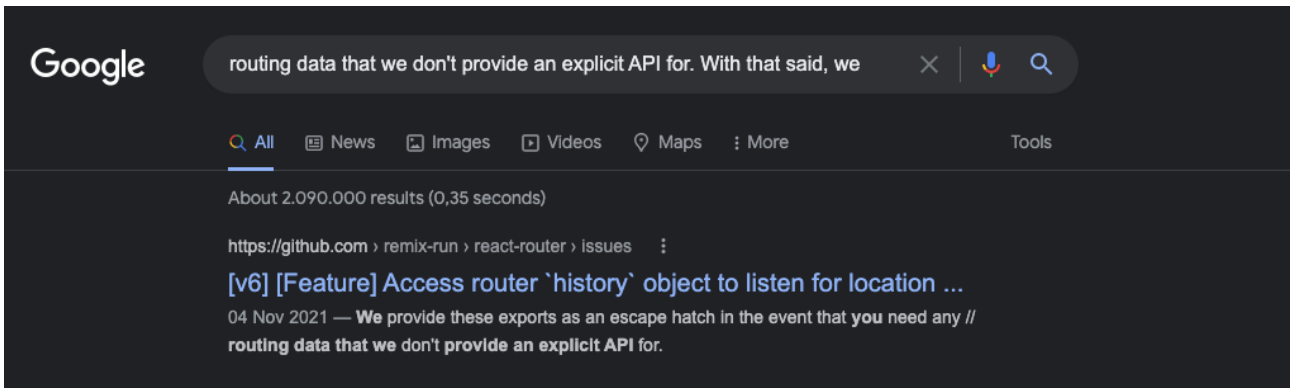
The first hurdle to take at this point is to find in the React application folder structure the custom made webpages for the challenge.

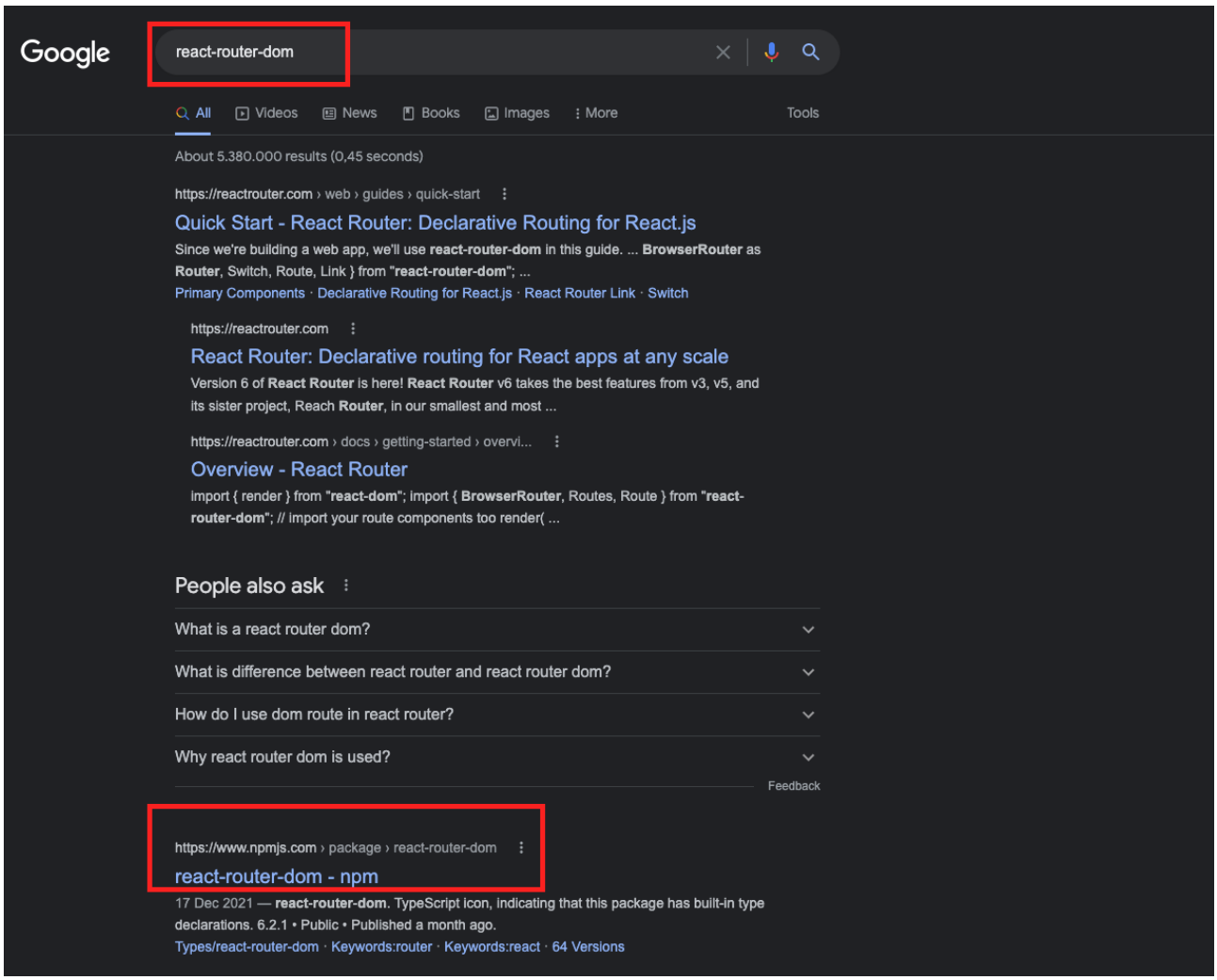
Probably not the fastest way but if I doubt if a certain file or folder is custom made I simply Google it. Take certain text from the source code or folder name and check if you get other results and compare if they are similar. Then you can know if this is a generally used folder or file for react applications or a custom one.

Example for the packages “react-router-dom”



We find the same code on Goolge so this is not custom made:



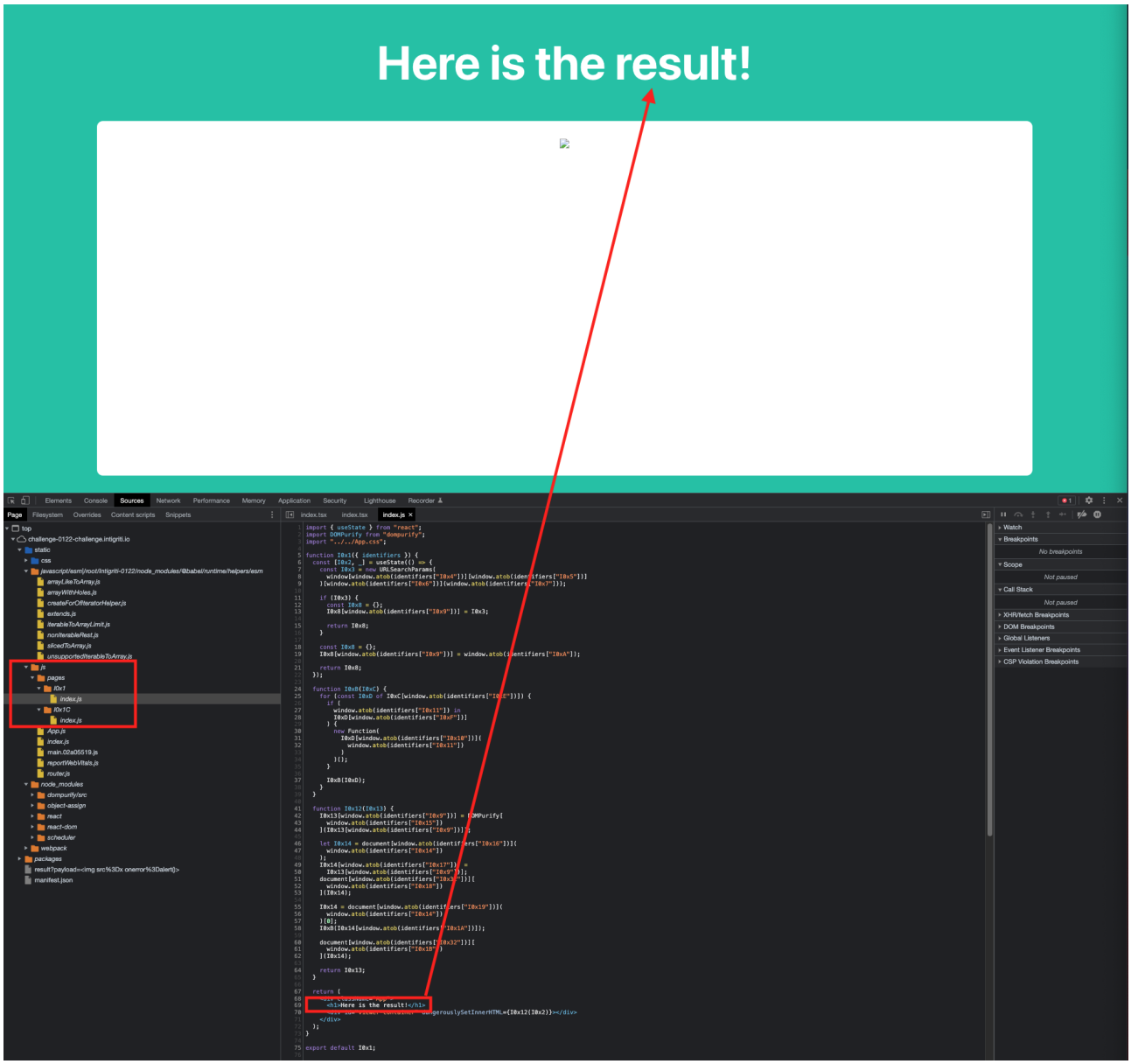


If you are not familiar with the react framework this is a possible way to check which files are custom made and which ones not.

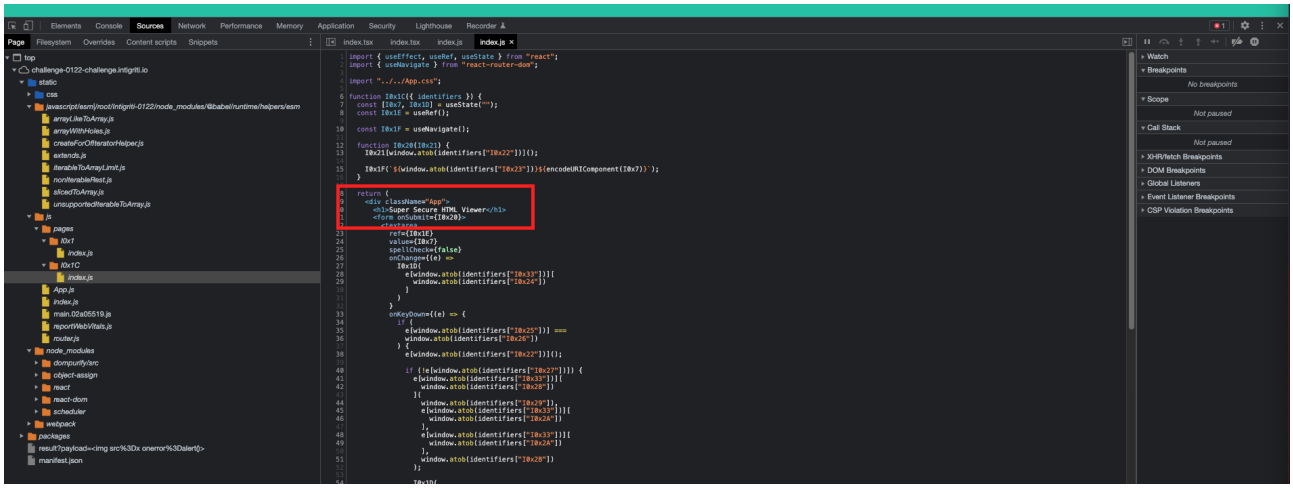
If we check the folder structure there are 2 files that should catch our eye:

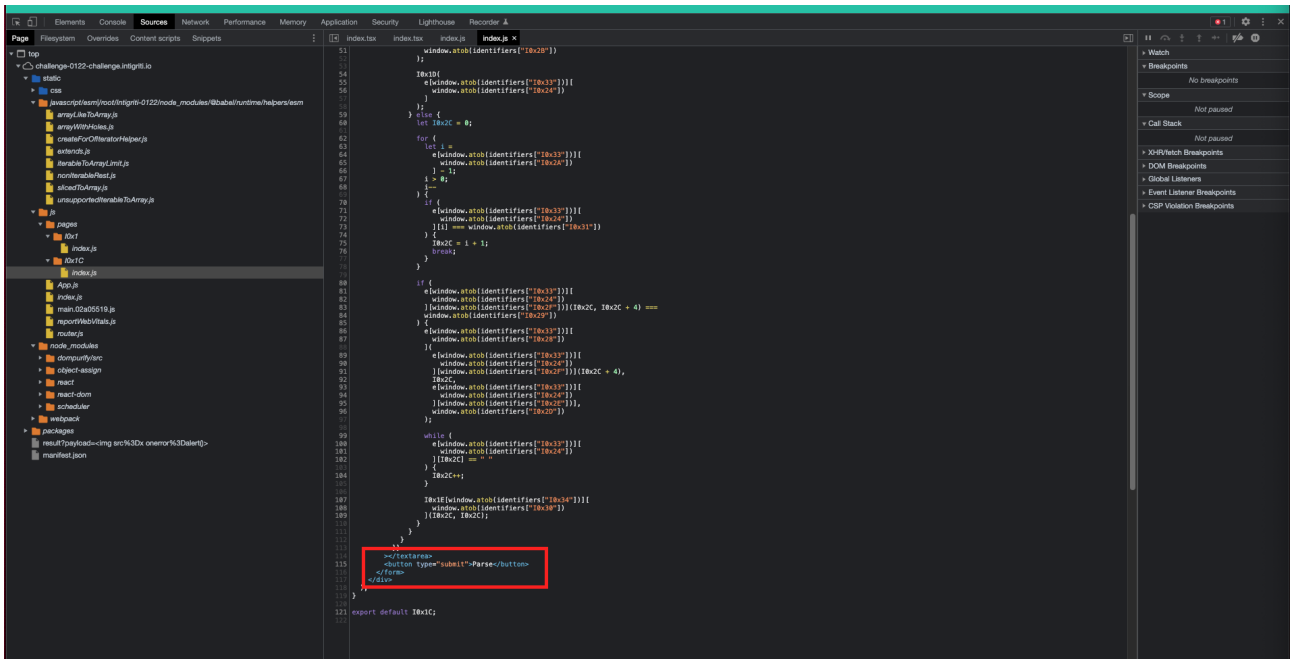
js → pages → I0x1 → index.js
js → pages → I0x1C → index.js

The source code for example reveals the text “Here is the result!” so we can be sure that this is directly linked to the challenge page we are using:



The other index.js file source code reveals the page title “Super Secure HTML viewer” and the “Parse” button:

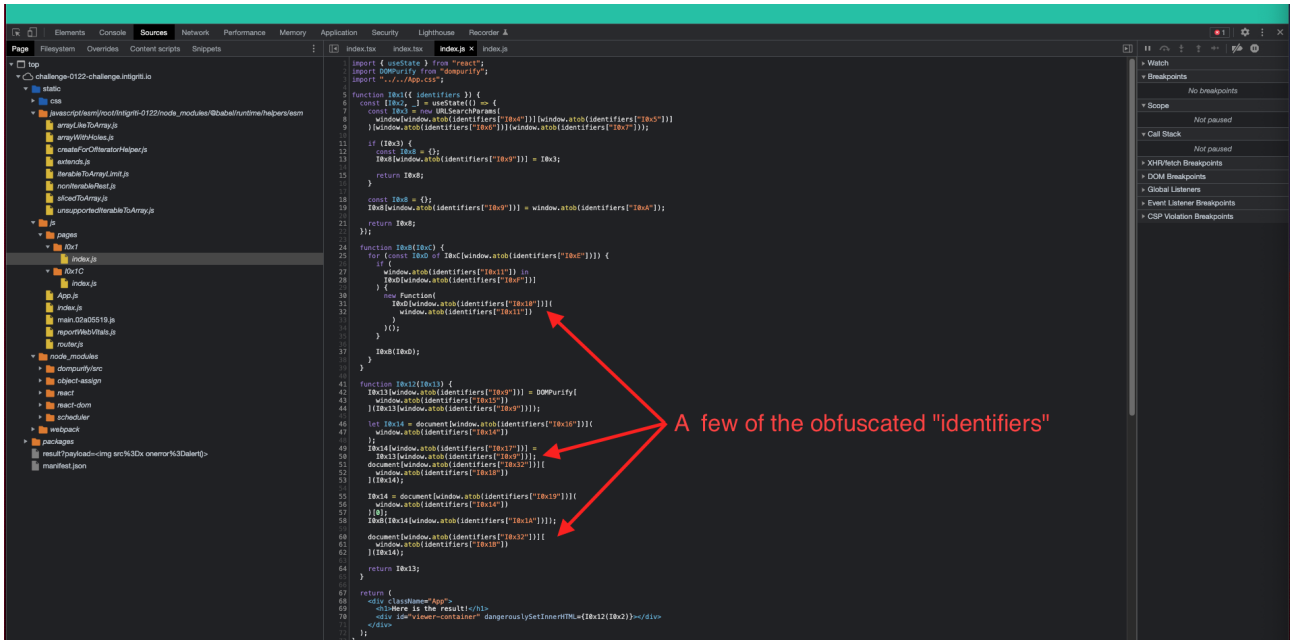


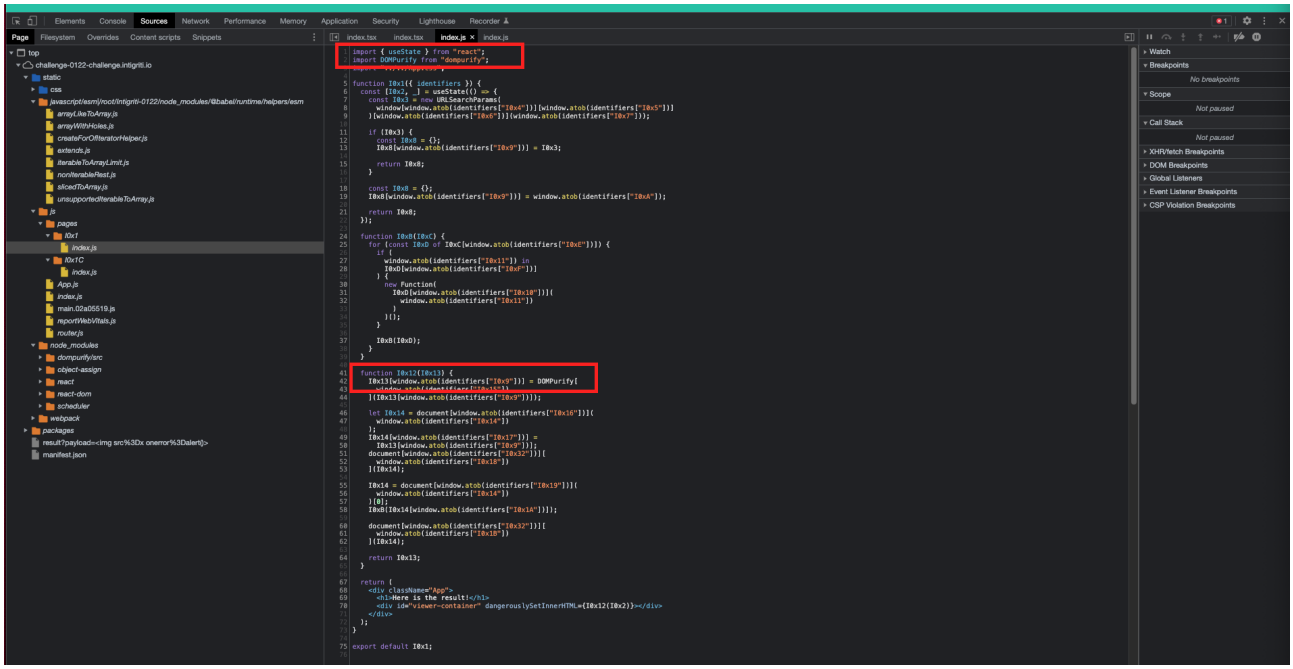


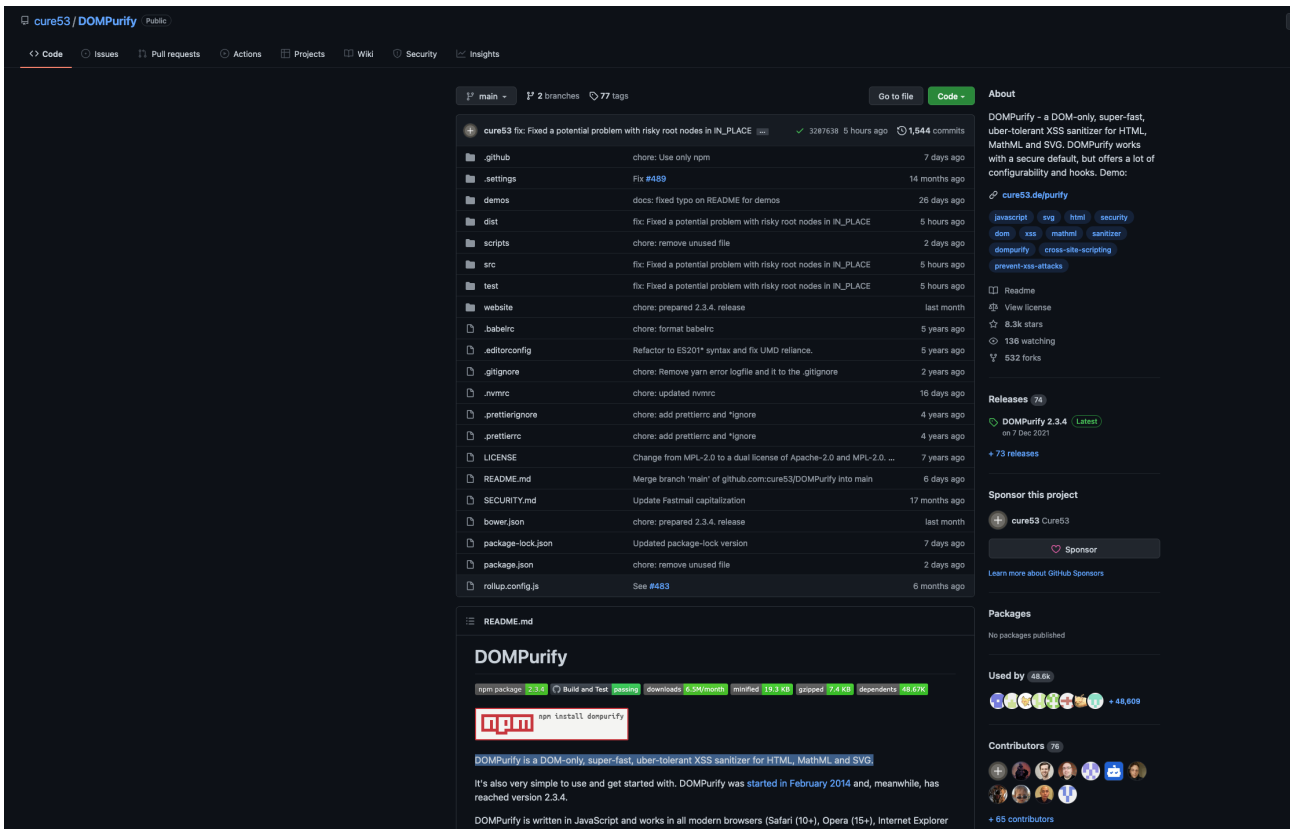
Quick inspection of our 2 custom made js files reveals some important things:

- Parts of the code are obfuscated as “identifiers” which seem base64 encoded with “atob”: https://developer.mozilla.org/en-US/docs/Web/API/atob

- We are up against DomPurify: *DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG. https://github.com/cure53/DOMPurify*





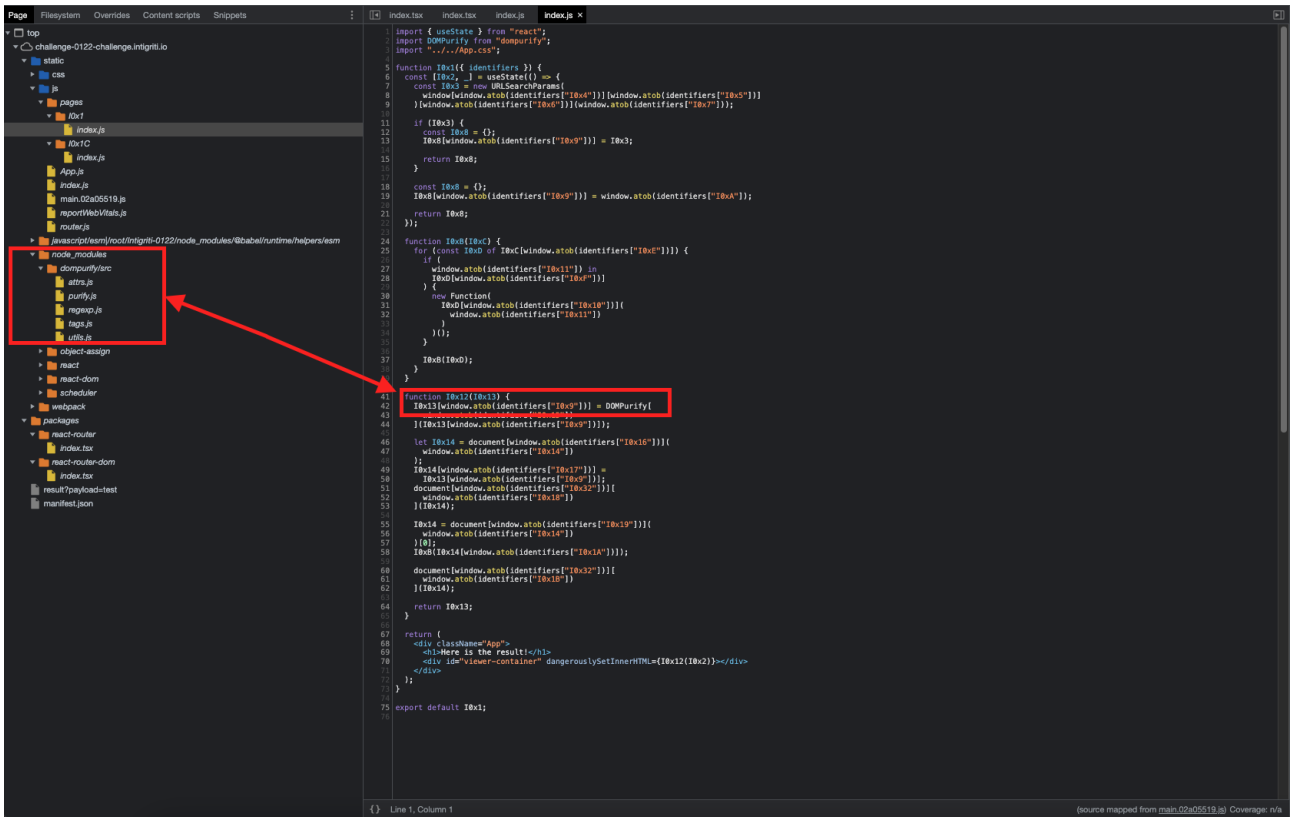


Takeaways from our recon:

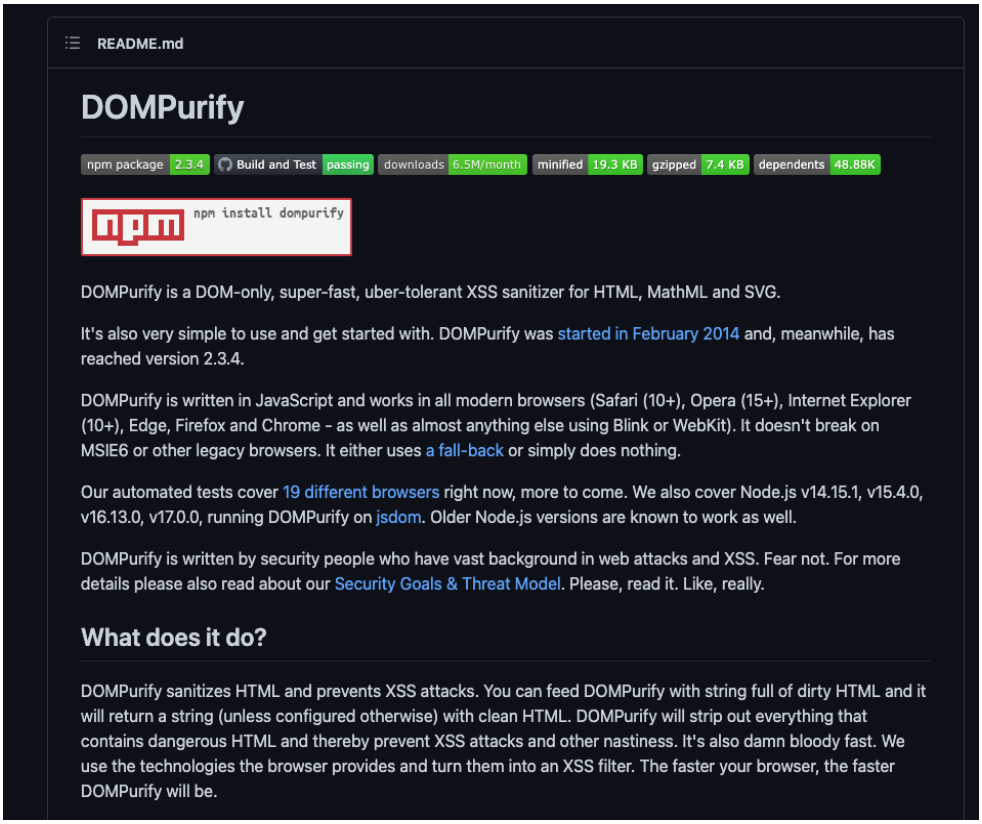
- Parameter: https://challenge-0122-challenge.intigriti.io/**result?payload=**
- React javascript library is used with 2 custom javascript files in folders “I0x1” and “I0x1C”
- Both javascript files have obfuscated “identifiers” that are base64 encoded.
- DOMPurify: sanitizer that blocks or prevents our XSS attacks.

Step 2: DOMPurify

As we have seen during our recon DOMPurify is implemented as a module into the React web application. This causes our input HTML being sanitized and thus our XSS payload not firing.



https://github.com/cure53/DOMPurify



README.md

DOMPurify

npm package 2.3.4 | Build and Test passing | downloads 6.5M/month | minified 19.3 KB | gzipped 7.4 KB | dependents 48.88K

npm install dompurify

DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG.

It's also very simple to use and get started with. DOMPurify was started in February 2014 and, meanwhile, has reached version 2.3.4.

DOMPurify is written in JavaScript and works in all modern browsers (Safari (10+), Opera (15+), Internet Explorer (10+), Edge, Firefox and Chrome - as well as almost anything else using Blink or WebKit). It doesn't break on MSIE6 or other legacy browsers. It either uses a fall-back or simply does nothing.

Our automated tests cover 19 different browsers right now, more to come. We also cover Node.js v14.15.1, v15.4.0, v16.13.0, v17.0.0, running DOMPurify on jsdom. Older Node.js versions are known to work as well.

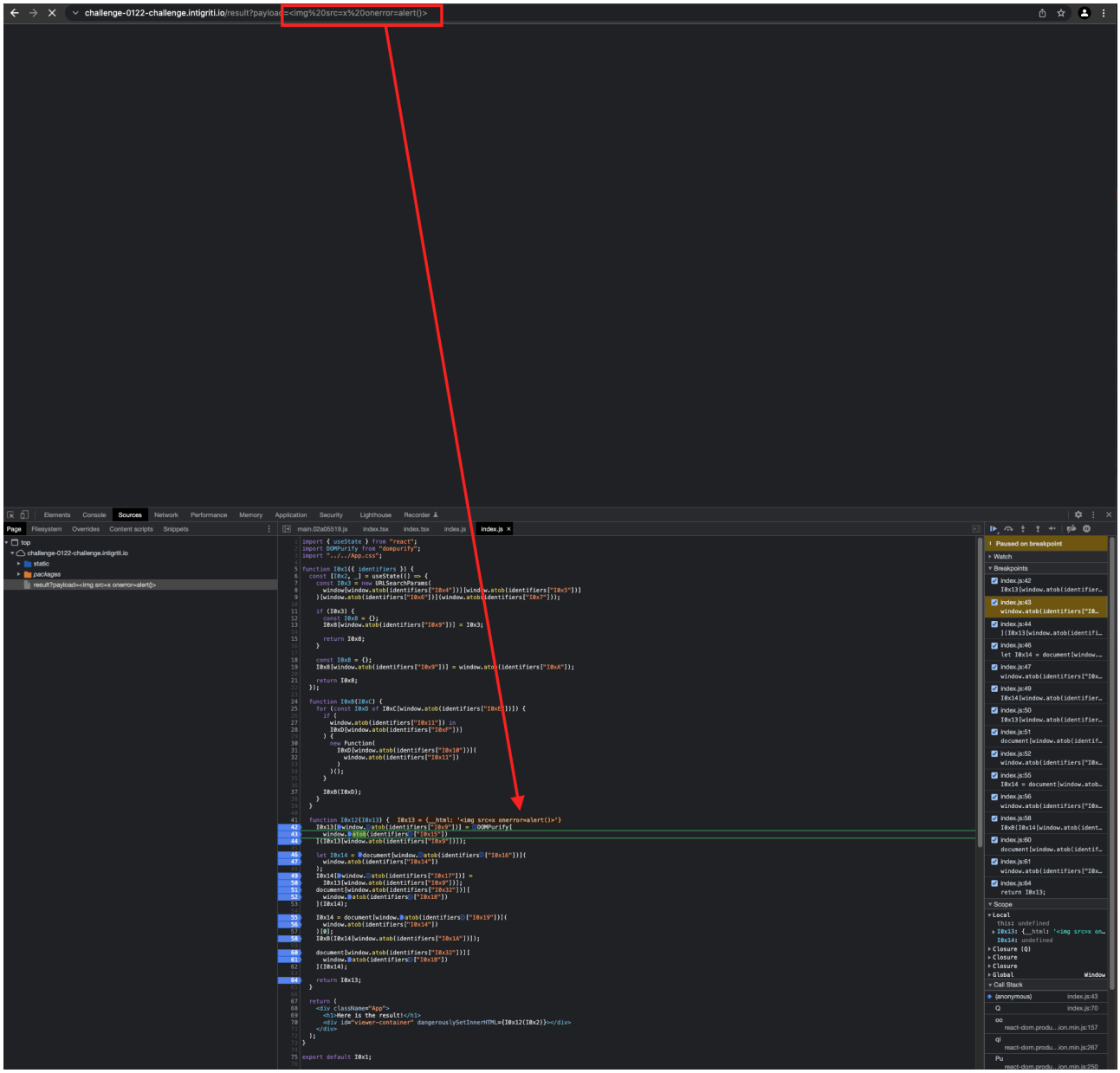
DOMPurify is written by security people who have vast background in web attacks and XSS. Fear not. For more details please also read about our Security Goals & Threat Model. Please, read it. Like, really.

What does it do?

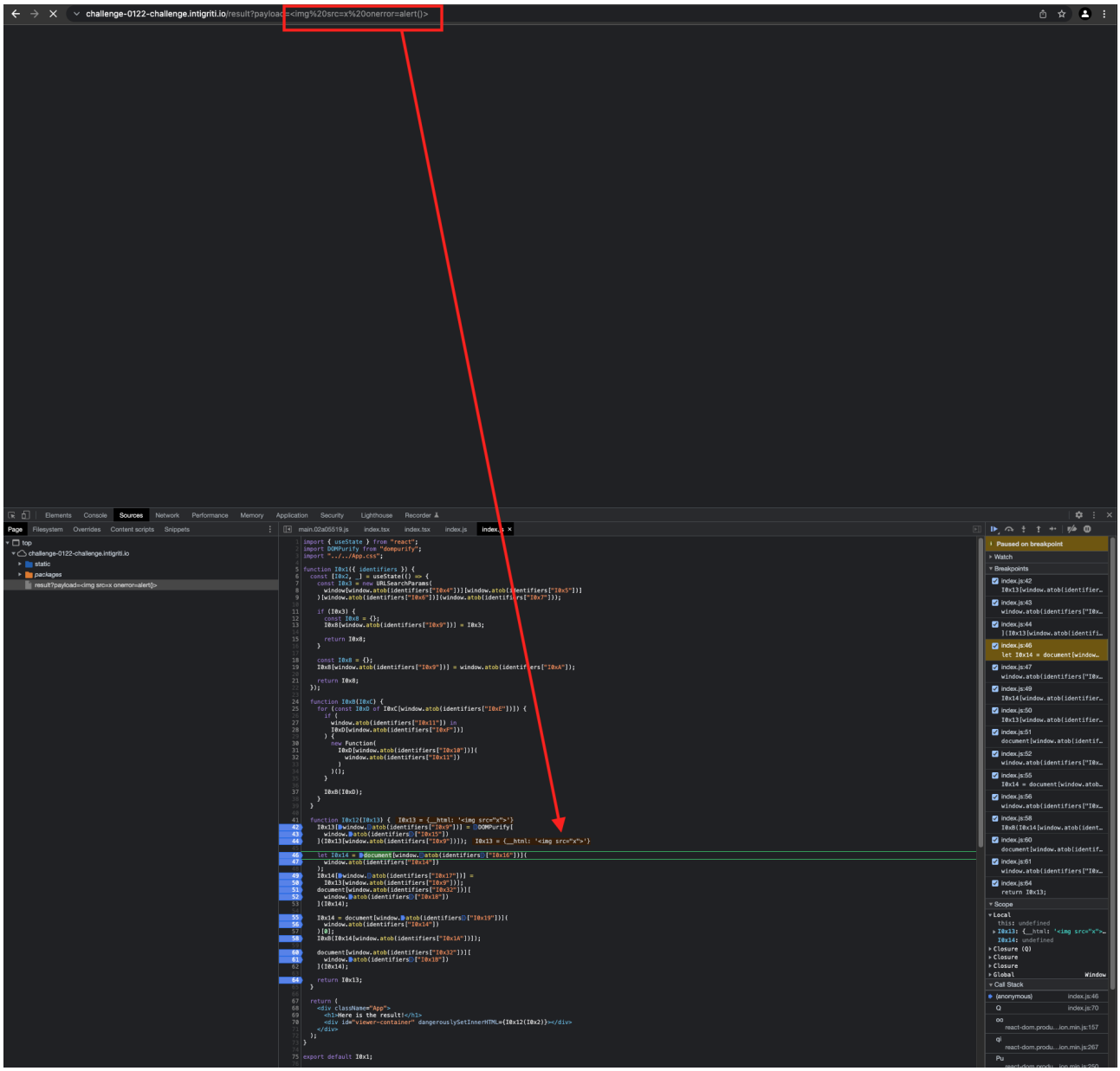
DOMPurify sanitizes HTML and prevents XSS attacks. You can feed DOMPurify with string full of dirty HTML and it will return a string (unless configured otherwise) with clean HTML. DOMPurify will strip out everything that contains dangerous HTML and thereby prevent XSS attacks and other nastiness. It's also damn bloody fast. We use the technologies the browser provides and turn them into an XSS filter. The faster your browser, the faster DOMPurify will be.

Here an example what happens with our XSS payload if I set breakpoints in the developer console:
<img src=x onerror=alert()>

Just before we pass the line starting the DOMpurify check on our input:



After DOMPurify did its checks on our input:



Our payload went from *<img src=x onerror=alert()>* to *<img src=x>* after DOMPurify sanitization.

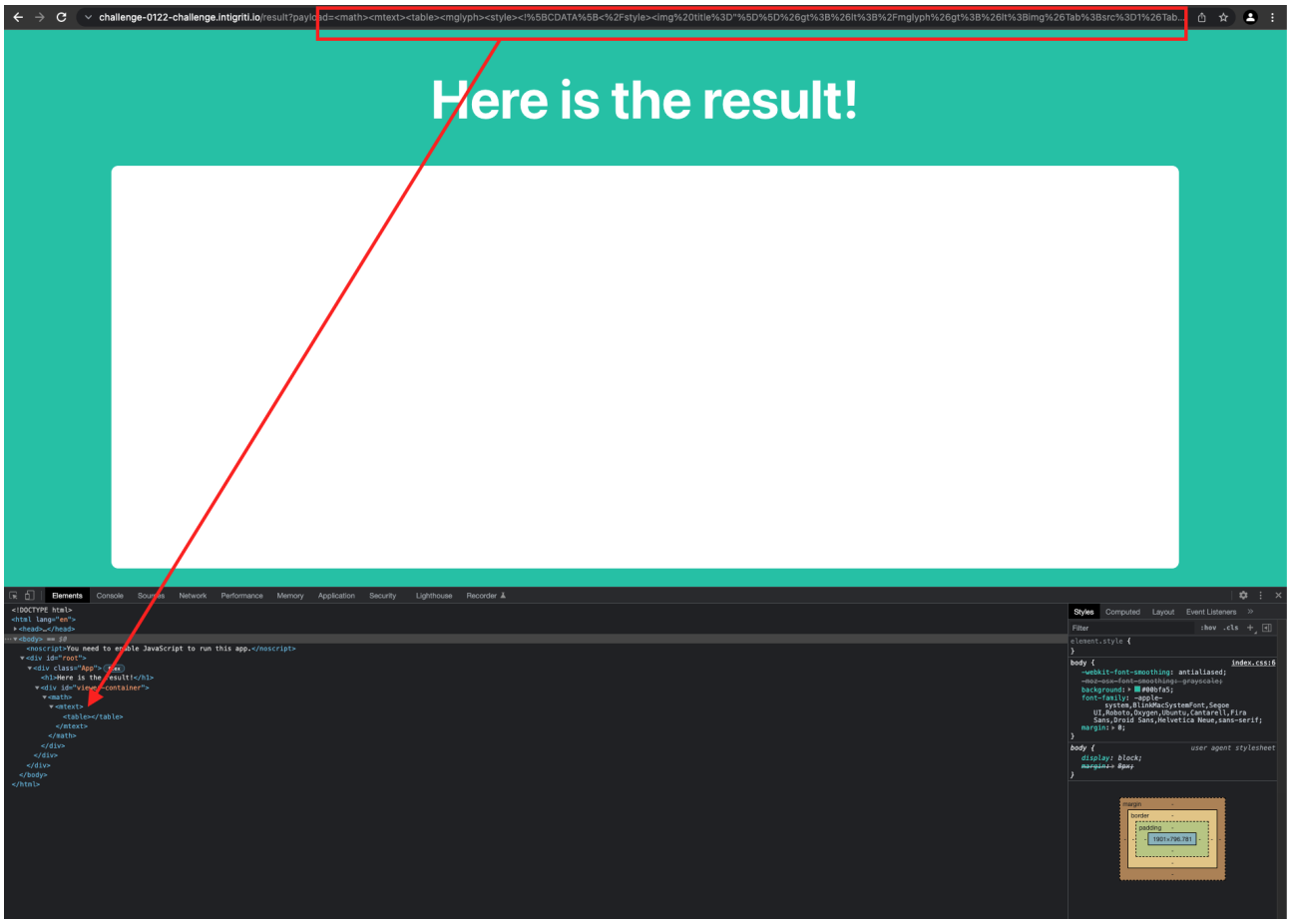
The “onerror” event handler is removed and we need it to execute arbitrary javascript or in other words to fire the XSS attack.

Of course DOMPurify had some bypasses in the past mainly via mutation XSS attacks. If the DOMPurify implemented by the web application developer is not up to date we have a chance to get our XSS.

A good article about DOMPurify bypasses: https://portswigger.net/research/bypassing-dompurify-again-with-mutation-xss

Lets try following mutation XSS payload from the article:
*<math><mtext><table><mglyph><style><![CDATA[</style><img title="]]></mglyph><img	src=1	onerror=alert(1)>">*

The XSS does not fire. Inspecting the page source code shows some reflection of a part of the input tags but not everything. The developer of this web application seems to have implemented an up to date version of DOMPurify.



With DOMPurify up to date it becomes hard to just fire XSS payloads as they get sanitized. In my opinion at this point only 2 options:

- We find a zero day against DOMPurify and bypass the sanitization. (chances are low ;-))
- The developer made a mistake in the source code and there is another way to bypass or skip the DOMPurify sanitization.

Step 3: Javascript obfuscation

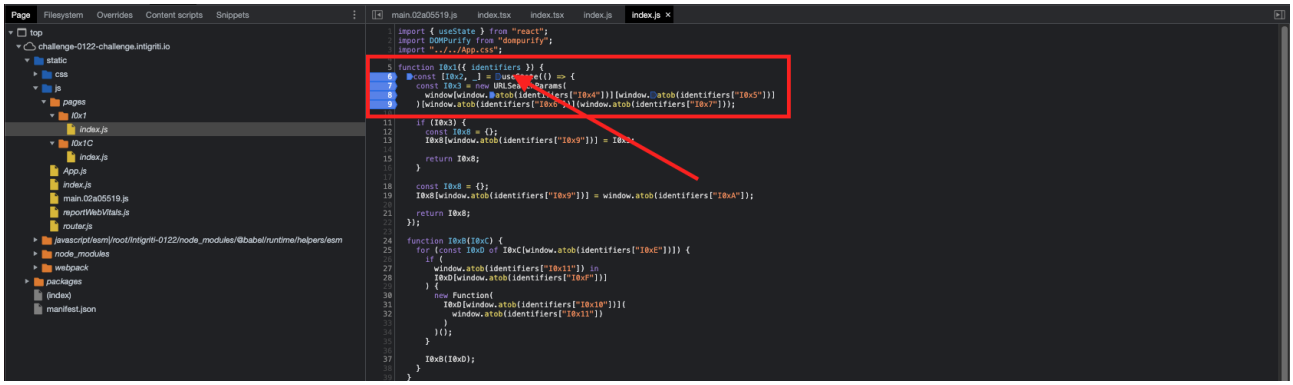
Ok with DOMPurify standing in our way we hope to find a mistake from the web application developer to bypass or skip the sanitization check.
Next hurdle that we noticed during our recon is that a big part of both custom made javascript files are obfuscated and not really readable.

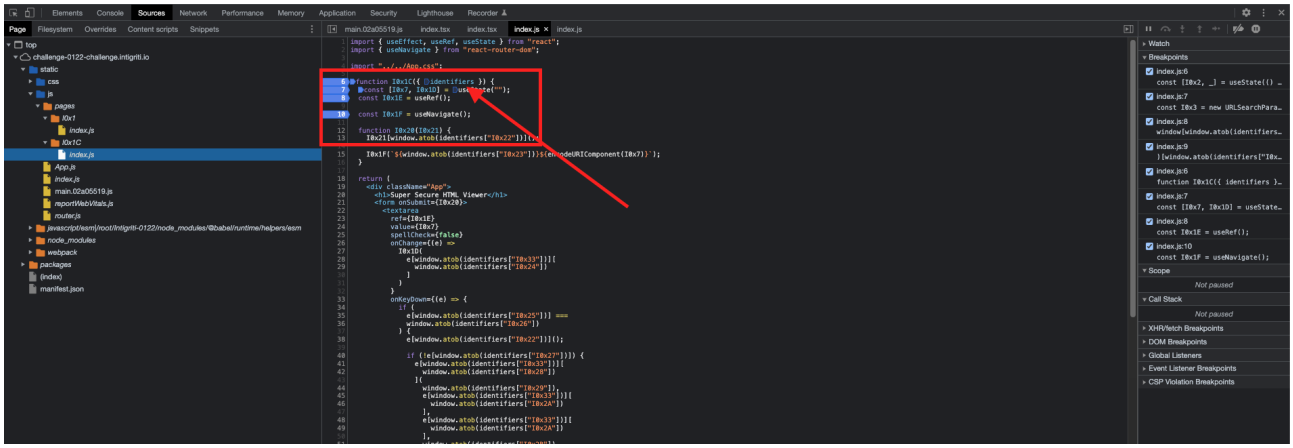
Possible approach at this point is to look for certain patterns and check if they can be de-obfuscated.

Both js files are full with this kind of patterns: *window.atob(identifiers["I0x15"])*

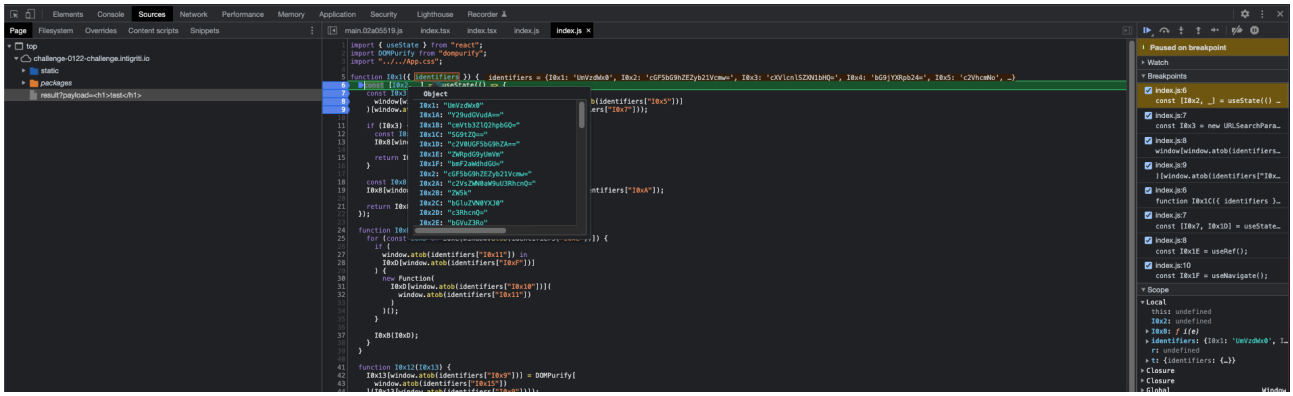
```
function I0x12(I0x13) {
  I0x13[window.atob(identifiers["I0x9"])] = DOMPurify[
    window.atob(identifiers["I0x15"])
  ](I0x13[window.atob(identifiers["I0x9"])]);

  let I0x14 = document[window.atob(identifiers["I0x16"])](
    window.atob(identifiers["I0x14"])
  );
  I0x14[window.atob(identifiers["I0x17"])] =
    I0x13[window.atob(identifiers["I0x9"])];
  document[window.atob(identifiers["I0x32"])][
    window.atob(identifiers["I0x18"])
  ](I0x14);

  I0x14 = document[window.atob(identifiers["I0x19"])](
    window.atob(identifiers["I0x14"])
  )[0];
  I0x8(I0x14[window.atob(identifiers["I0x1A"])]);

  document[window.atob(identifiers["I0x32"])][
    window.atob(identifiers["I0x1B"])
  ](I0x14);

  return I0x13;
}
```

We need to get those “identifiers”. Both js files contain a function that seems to use “identifiers”. That is interesting because we can set breakpoints in our source code and check the content of “identifiers” (use F8 to go through the breakpoints step by step):





With F8 button we can go through each breakpoint step by step and this reveals the content of “identifiers”. (This can be copied and pasted somewhere else.)

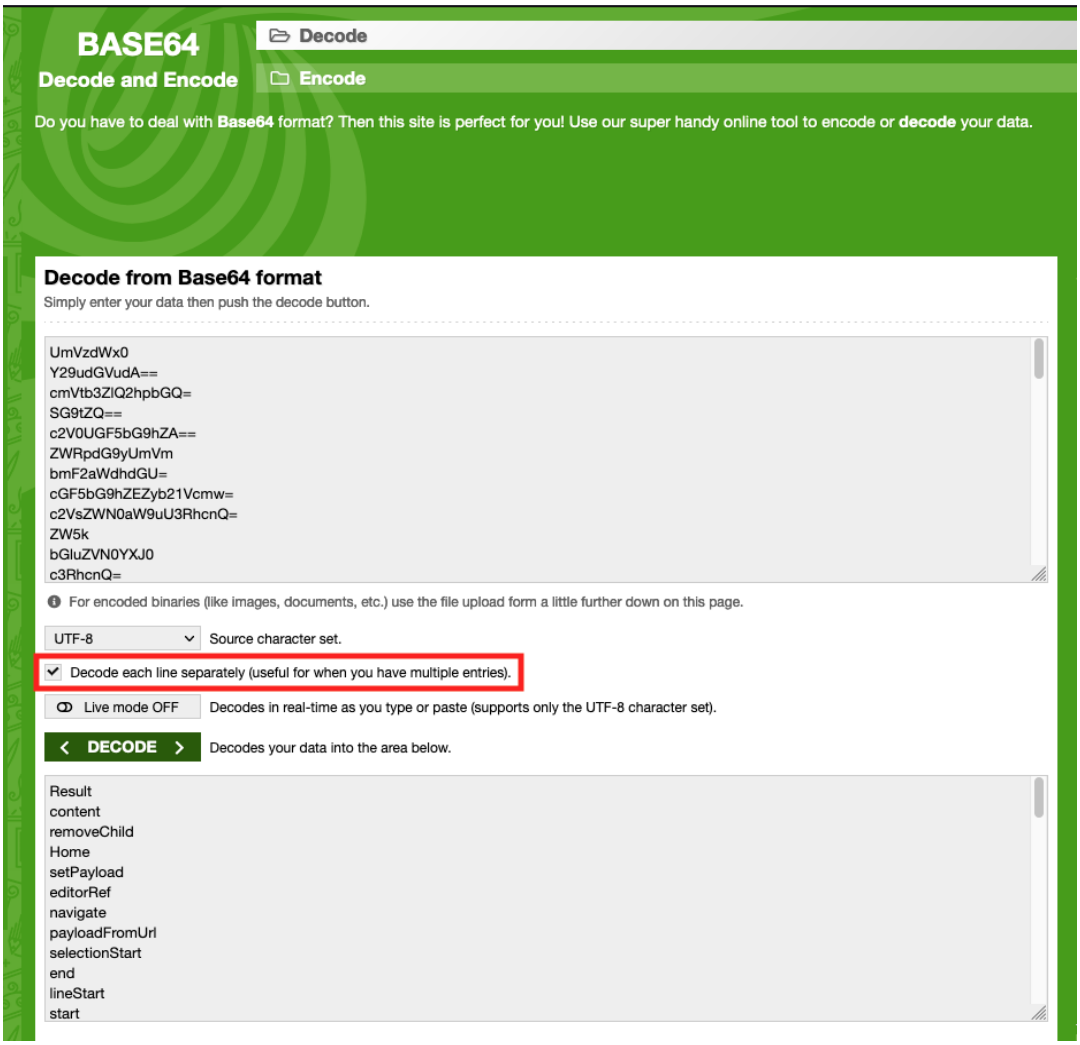


Here the “identifiers” pasted in a text file with their corresponding base64 value:

```
I0x1: "UmVzdWx0"
I0x1A: "Y29udGVudA=="
I0x1B: "cmVtb3ZlQ2hpbGQ="
I0x1C: "SG9tZQ=="
I0x1D: "c2V0UGF5bG9hZA=="
I0x1E: "ZWRpdG9yUmVm"
I0x1F: "bmF2aWdhdGU="
I0x2: "cGF5bG9hZEZyb21Vcmw="
I0x2A: "c2VsZWN0aW9uU3RhcnQ="
I0x2B: "ZW5k"
I0x2C: "bGluZVN0YXJ0"
I0x2D: "c3RhcnQ="
I0x2E: "bGVuZ3Ro"
I0x2F: "c2xpY2U="
I0x3: "cXVlcnlSZXN1bHQ="
I0x4: "hG9iYXRpb24="
I0x5: "c2VhcmNo"
I0x6: "Z2V0"
I0x7: "cGF5bG9hZA=="
I0x8: "cmVzdWx0"
I0x9: "X19odG1s"
I0x10: "Z2V0QXR0cmlidXRl"
I0x11: "ZGF0YS1kZWJ1Zw=="
I0x12: "c2FuaXRpemVIVE1M"
I0x13: "aHRtbE9iag=="
I0x14: "dGVtcGxhdGU="
I0x15: "c2FuaXRpemU="
I0x16: "Y3JlYXRlRWxlbWVudA=="
I0x17: "aW5uZXJIVE1M"
I0x18: "YXBwZW5kQ2hpbGQ="
I0x19: "Z2V0RWxlbWVudHNCeVRhZ05hbWU="
I0x20: "aGFuZGxlU3VibWl0"
I0x21: "ZXZlbnQ="
I0x22: "cHJldmVudERlZmF1bHQ="
I0x23: "L3Jlc3VsdD9wYXlsb2FkPQ=="
I0x24: "dmFsdWU="
I0x25: "a2V5"
I0x26: "VGFi"
I0x27: "c2hpZnRLZXk="
I0x28: "c2V0UmFuZ2VUZXh0"
I0x29: "ICAgIA=="
I0x30: "c2V0U2VsZWN0aW9uUmFuZ2U="
I0x31: "Cg=="
I0x32: "Ym9keQ=="
I0x33: "dGFyZ2V0"
I0x34: "Y3VycmVudA=="
I0xA: "PGgxIHN0eWxlPSdjb2xvcjogIzAwYmZhNSc+Tm90aGluZyBoZXJlITwvaDE+"
I0xB: "aGFuZGxlQXR0cmlidXRlcw=="
I0xC: "ZWxlbWVudA=="
I0xD: "Y2hpbGQ="
I0xE: "Y2hpbGRyZW4="
I0xF: "YXR0cmlidXRlcw=="
```

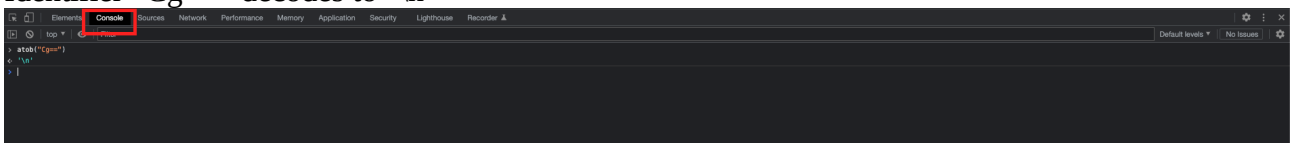
The source code already revealed they are base64 encoded via the “atob” function that became clear during our recon. We can now easily decode each value.

https://www.base64decode.org/ (mark the option to decode a list). You will notice some of the base64 encoded lines convert to a blank line. Those I decoded manually via the browser developer tools.



Or via the browser developer tools Console manual decode from base64 with “atob”:

Identifier “Cg==” decodes to “\n”



This gave me following list for the “identifiers” – base64 value – decoded value:

	A	B	C
1	l0x1:	UmVzdWx0	Result
2	l0x1A:	Y29udGVudA==	content
3	l0x1B:	cmVtb3ZlQ2hpbGQ=	removeChild
4	l0x1C:	SG9tZQ==	Home
5	l0x1D:	c2V0UGF5bG9hZA==	setPayload
6	l0x1E:	ZWRpdG9yUmVm	editorRef
7	l0x1F:	bmF2aWdhdGU=	navigate
8	l0x2:	cGF5bG9hZEZyb21Vcmw=	payloadFromUrl
9	l0x2A:	c2VsZWN0aW9uU3RhcnQ=	selectionStart
10	l0x2B:	ZW5k	end
11	l0x2C:	bGluZVN0YXJ0	lineStart
12	l0x2D:	c3RhcnQ=	start
13	l0x2E:	bGVuZ3Ro	length
14	l0x2F:	c2xpY2U=	slice
15	l0x3:	cXVlcnlSZXN1bHQ=	queryResult
16	l0x4:	bG9jYXRpb24=	location
17	l0x5:	c2VhcmNo	search
18	l0x6:	Z2V0	get
19	l0x7:	cGF5bG9hZA==	payload
20	l0x8:	cmVzdWx0	result
21	l0x9:	X19odG1s	__html
22	l0x10:	Z2V0QXR0cmlidXRl	getAttribute
23	l0x11:	ZGF0YS1kZWJ1Zw==	data-debug
24	l0x12:	c2FuaXRpemVlVE1M	sanitizeHTML
25	l0x13:	aHRtbE9iag==	htmlObj
26	l0x14:	dGVtcGxhdGU=	template
27	l0x15:	c2FuaXRpemU=	sanitize
28	l0x16:	Y3JlYXRlRWxlbWVudA==	createElement
29	l0x17:	aW5uZXJlVE1M	innerHTML
30	l0x18:	YXBwZW5kQ2hpbGQ=	appendChild
31	l0x19:	Z2V0RWxlbWVudHNCeVRhZ05hbWU=	getElementsByTagName
32	l0x20:	aGFuZGxlU3VibWl0	handleSubmit
33	l0x21:	ZXZlbnQ=	event
34	l0x22:	cHJldmVudERlZmF1bHQ=	preventDefault
35	l0x23:	L3Jlc3VsdD9wYXlsb2FkPQ==	/result?payload=
36	l0x24:	dmFsdWU=	value
37	l0x25:	a2V5	key
38	l0x26:	VGFi	Tab
39	l0x27:	c2hpZnRLZXk=	shiftKey
40	l0x28:	c2V0UmFuZ2VUZXh0	setRangeText
41	l0x29:	ICAgIA==	' '
42	l0x30:	c2V0U2VsZWN0aW9uUmFuZ2U=	setSelectionRange
43	l0x31:	Cg==	\n
44	l0x32:	Ym9keQ==	body
45	l0x33:	dGFyZ2V0	target
46	l0x34:	Y3VycmVudA==	current
47	l0xA:	PGgxIHN0eWxlPSdjb2xvcjogIzAwYmZhNSc+Tm90aGluZyBoZXJlITwvaDE+	<h1 style='color: #00bfa5'>Nothing here!</h1>
48	l0xB:	aGFuZGxlQXR0cmlidXRlcw==	handleAttributes
49	l0xC:	ZWxlbWVudA==	element
50	l0xD:	Y2hpbGQ=	child
51	l0xE:	Y2hpbGRyZW4=	children
52	l0xF:	YXR0cmlidXRlcw==	attributes
53			

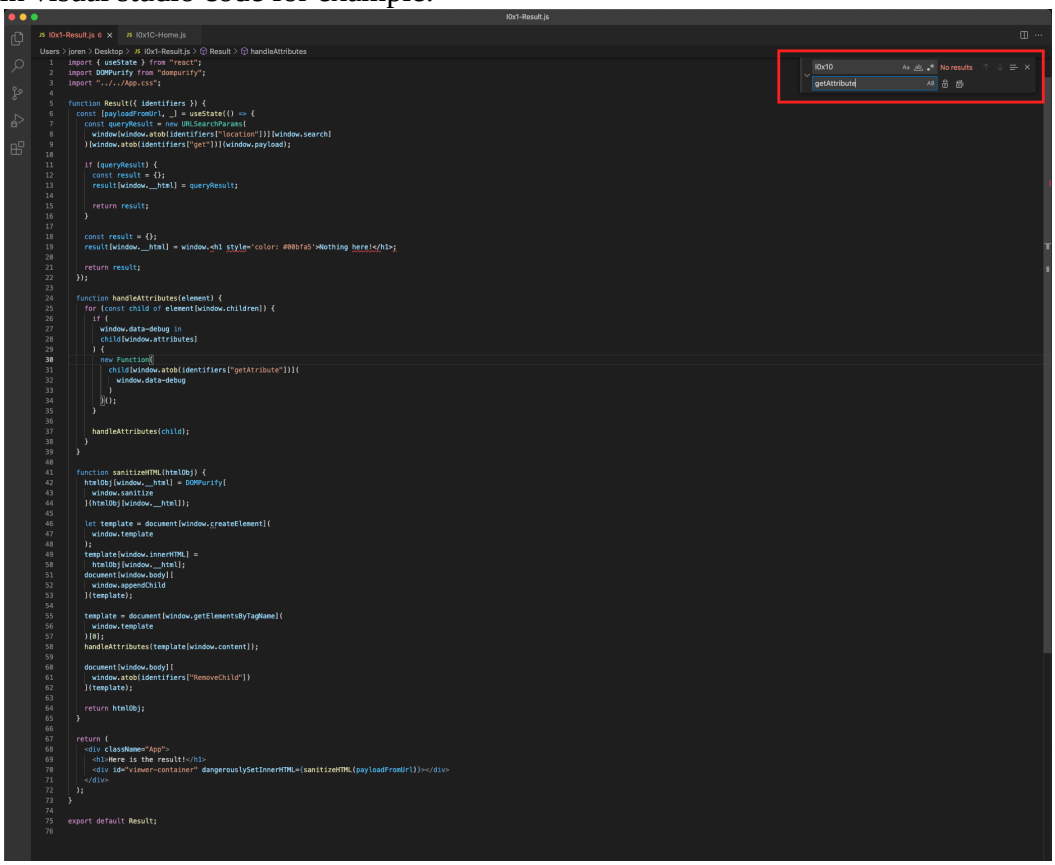
If we now replace each identifier in the 2 custom js files with the decoded values it becomes much more readable.

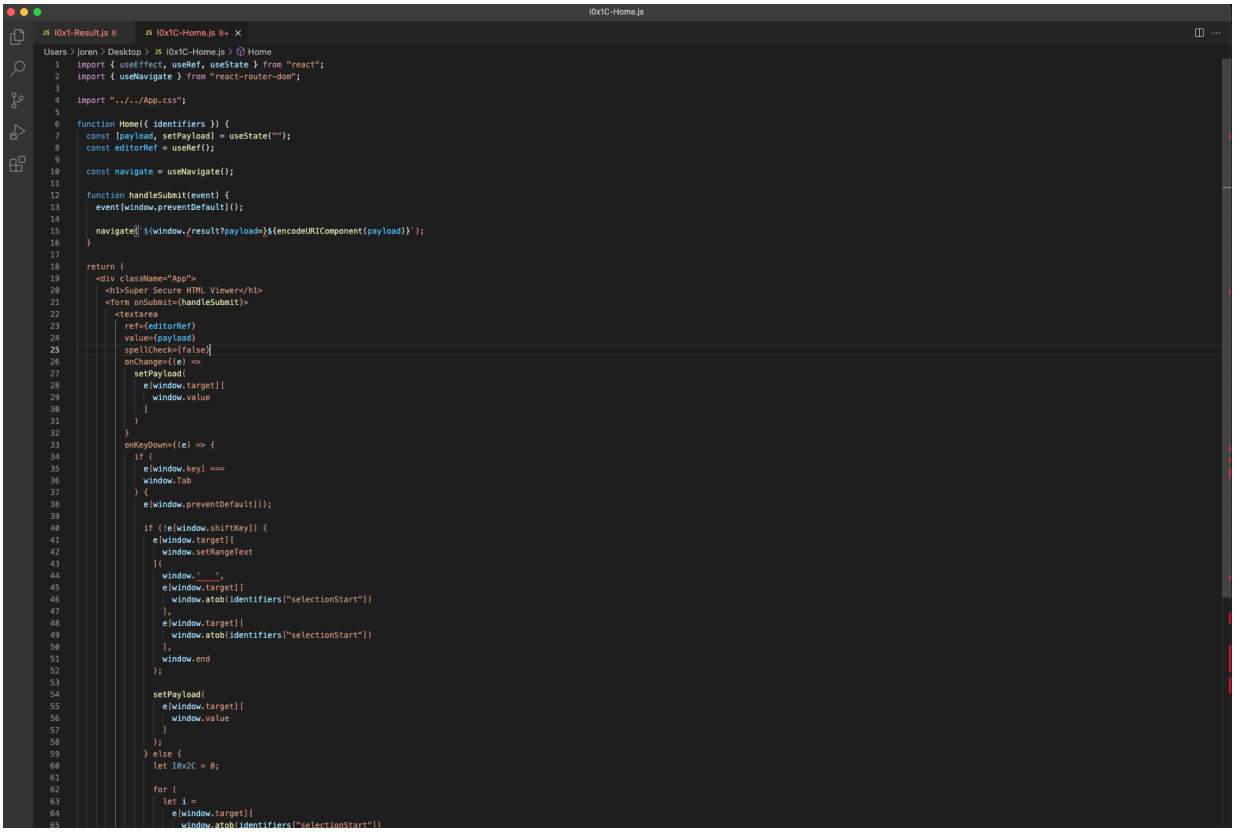
If we check the decoded list we now know the real names of both folders containing the js files:

I0X1 = result
I0X1C = home

Now there are multiple options. You could automate the replacing of each identifier by its decoded value via a Linux bash script or a python script for example. Anything can be used here.

Or a bit more manual work first copy the source code from the developer tools and use “find and replace” in visual studio code for example:



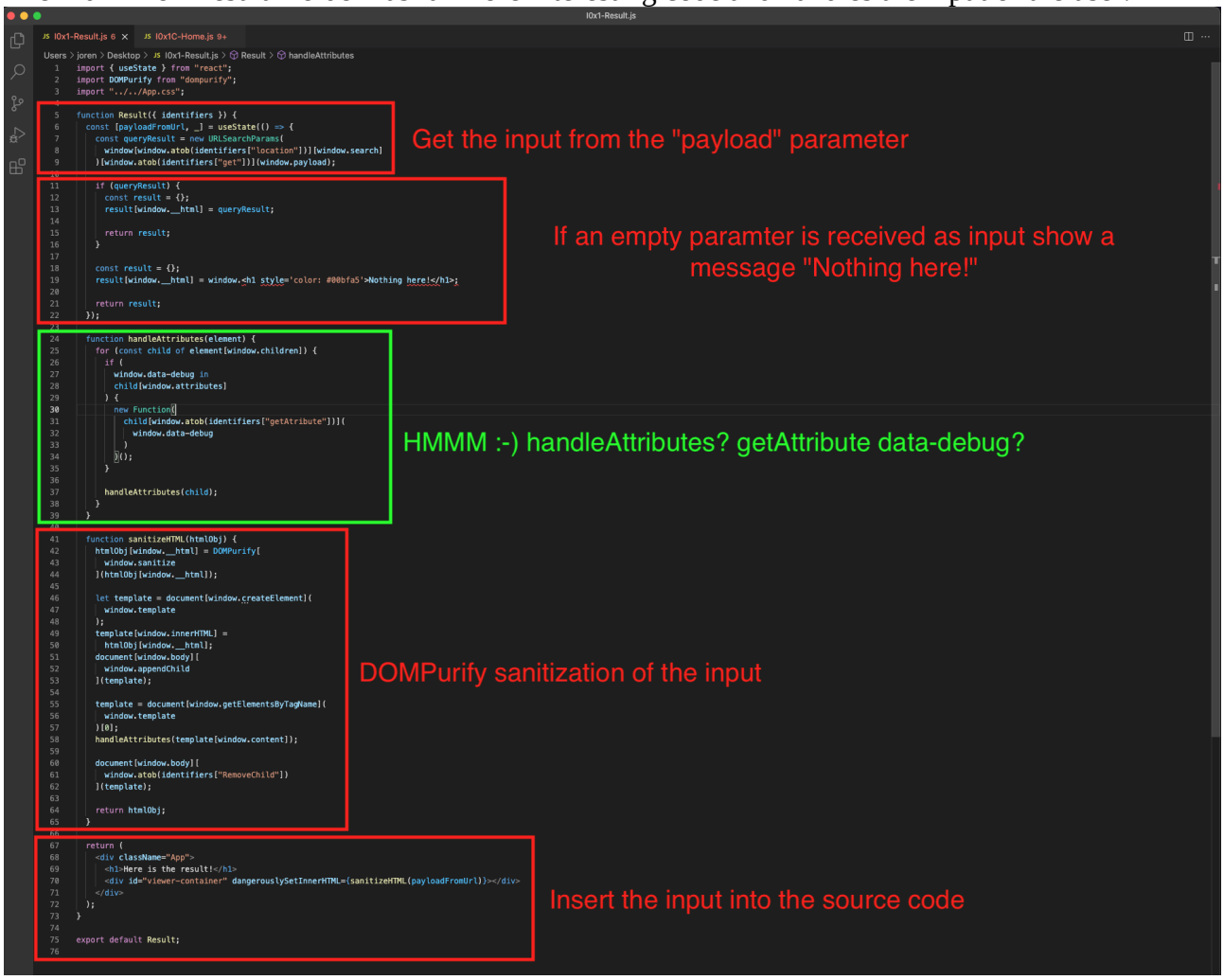


Step 4: Finding the weak spot

With both custom js files a bit more readable we can check them better. We already figured that this web application probably somewhere has a weak spot that bypasses or skips the DOMPurify check.

The “I0X1C” or “home” folder contains less interesting code in my opinion. It takes care of setting up the input textarea, parse button… but does not really handle the users input:

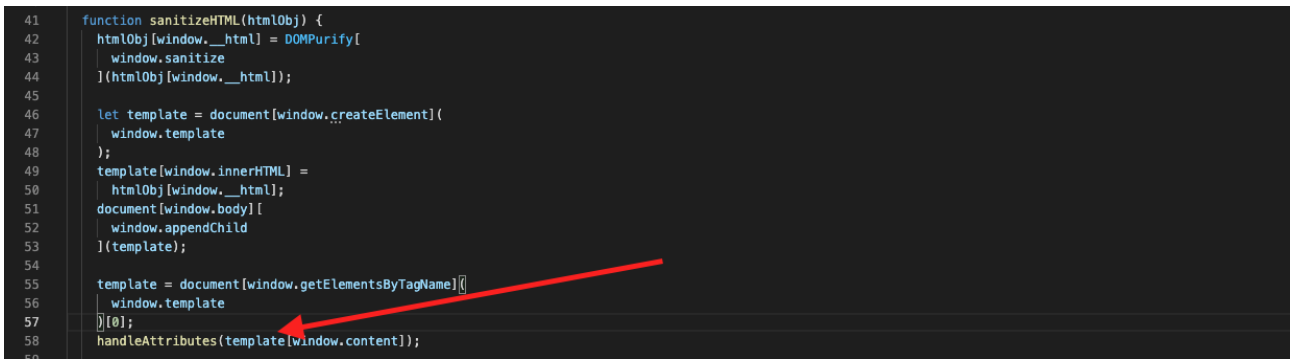
The “I0X1” or “result” folder has far more interesting code and handles the input of the user:



Something is a bit strange here. Why is the developer handling attributes and trying to get a “data-debug” attribute? This immediately makes alarm bells go off :-). Is this some kind of debugging part of the code the developer forgot to remove?

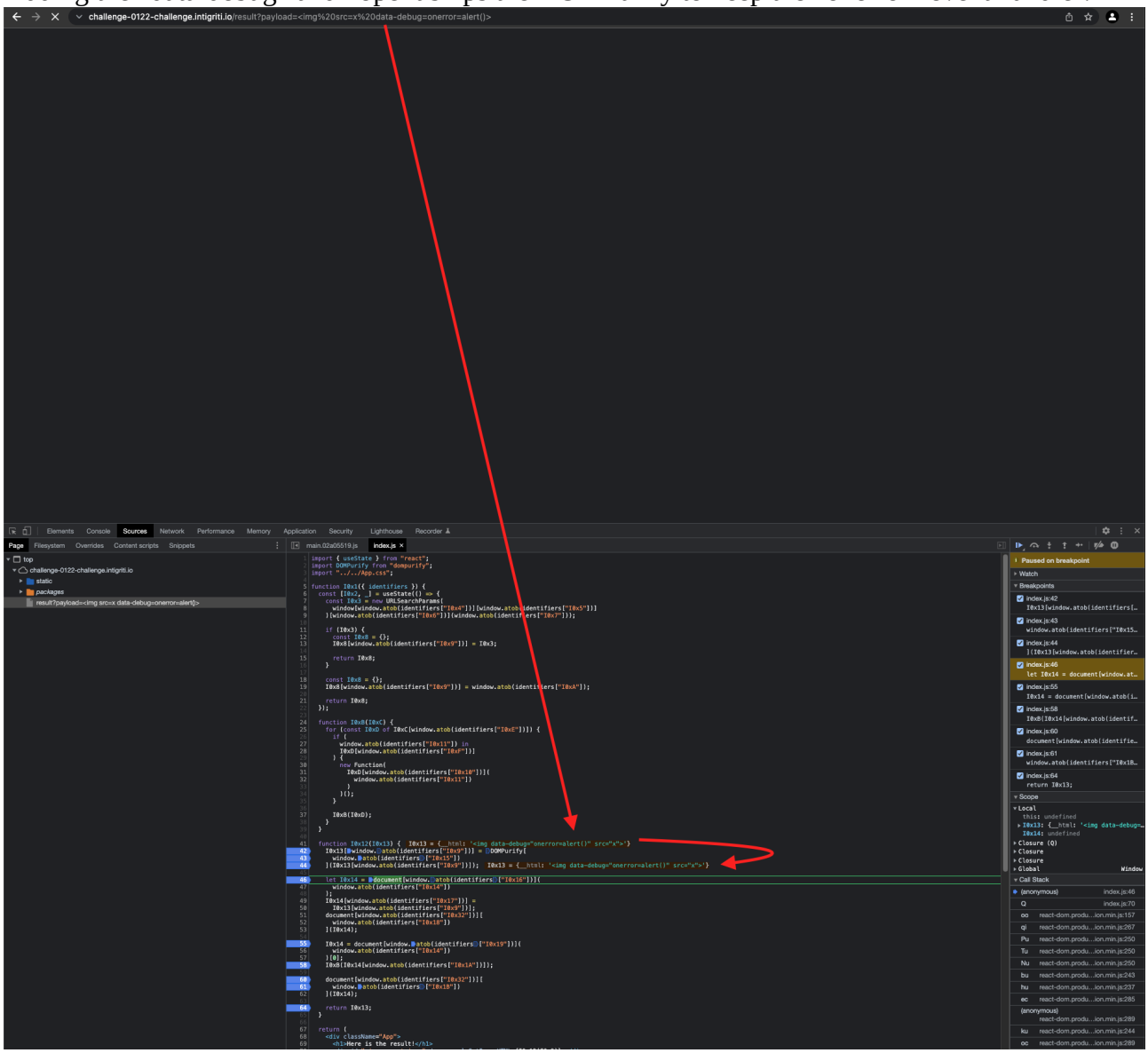
If you look at the DOMPurify part of the code you also see the “handleAtributes” function being used there in a HTML template tag so there is a big chance the “data-debug” attribute bypasses or skips the DOMPurify check.

HTML template tags hold some content hidden on the page when it loads until javascript calls to display the content: https://www.w3schools.com/tags/tag_template.asp

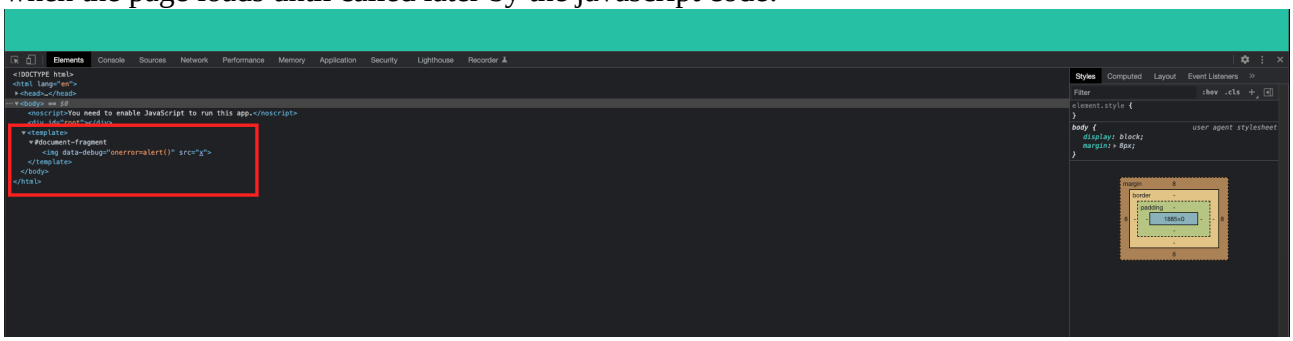


The first idea that came to my mind was following payload as input: *<img src=x* ***data-debug=****onerror=alert()>*

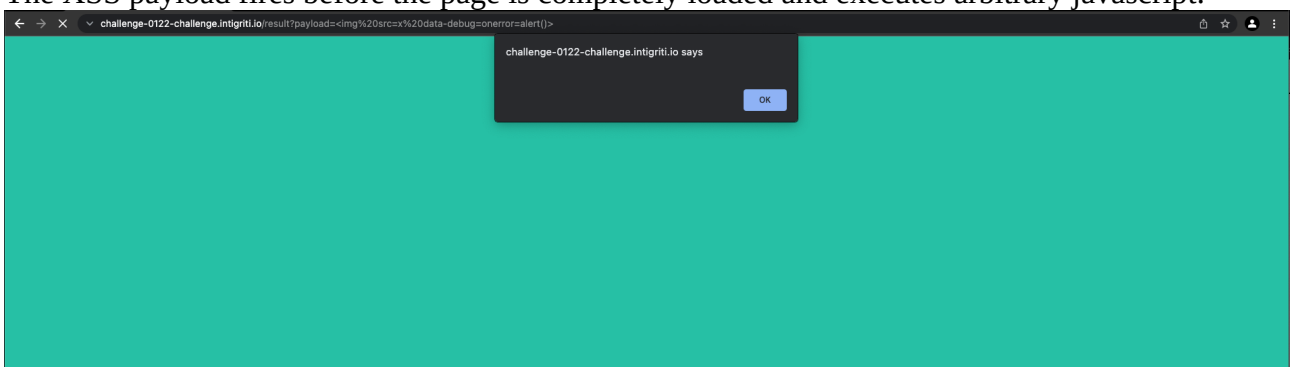
Adding the “data-debug” and hope it skips the DOMPurify to keep the “onerror” event handler:



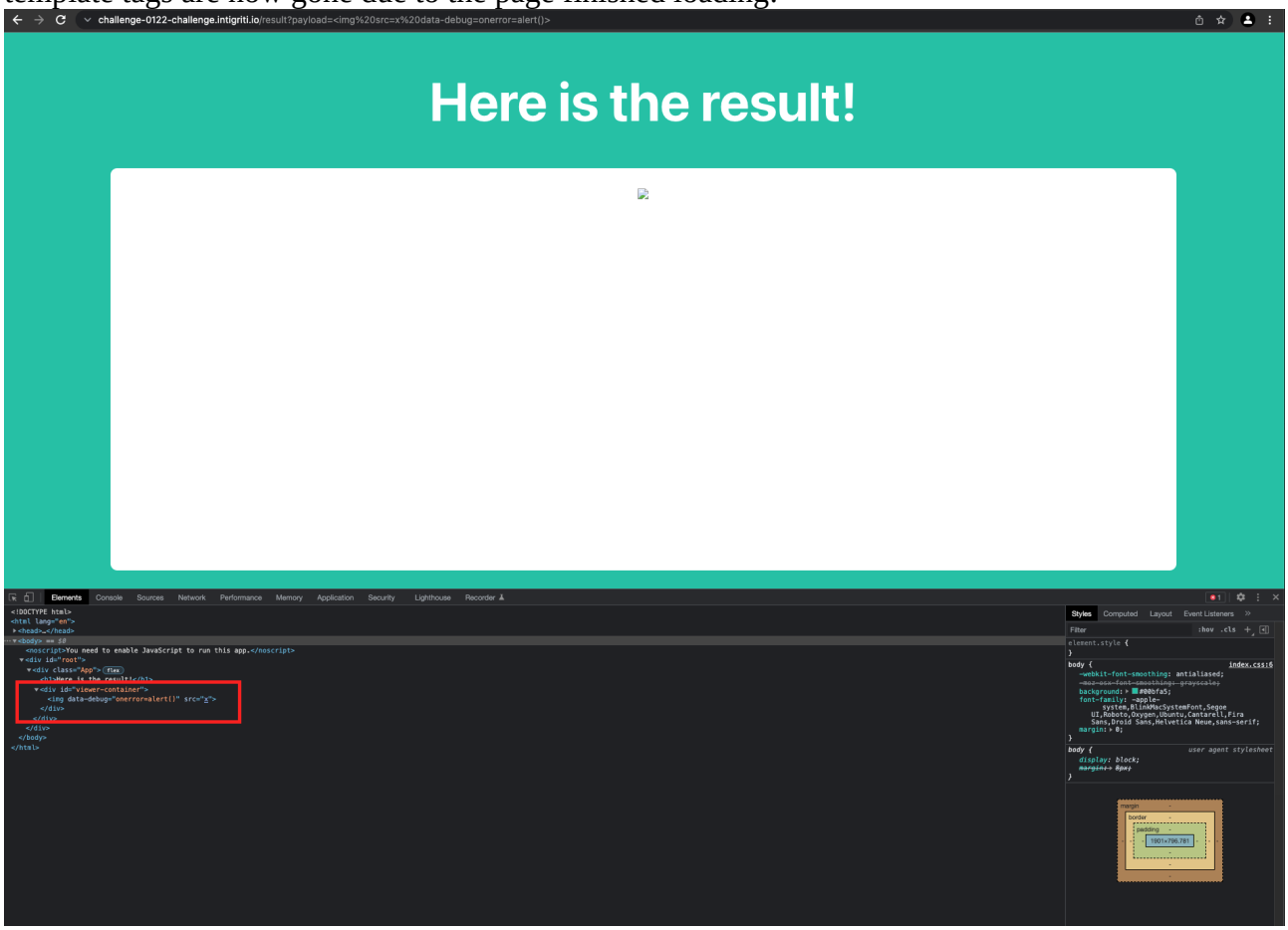
The javascript code generates the template tags and in between our payload that will be hidden when the page loads until called later by the javascript code:



The XSS payload fires before the page is completely loaded and executes arbitrary javascript:



The complete input got into the source code and executed just before the page finished loading. The template tags are now gone due to the page finished loading:



Following payload works both on Chrome and FireFox and alerts “document.domain”:

https://challenge-0122-challenge.intigriti.io/result?payload=*<img src=x data-debug=onerror=alert(document.domain)>*

https://challenge-0122-challenge.intigriti.io/result?payload=%3Cimg%20src=x%20data-debug=onerror=alert(document.domain)%3E