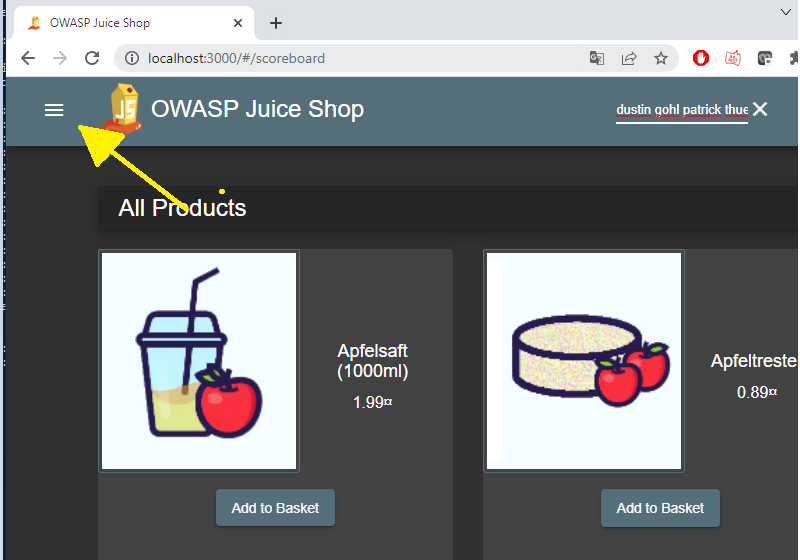
Lab 1: Web application Security

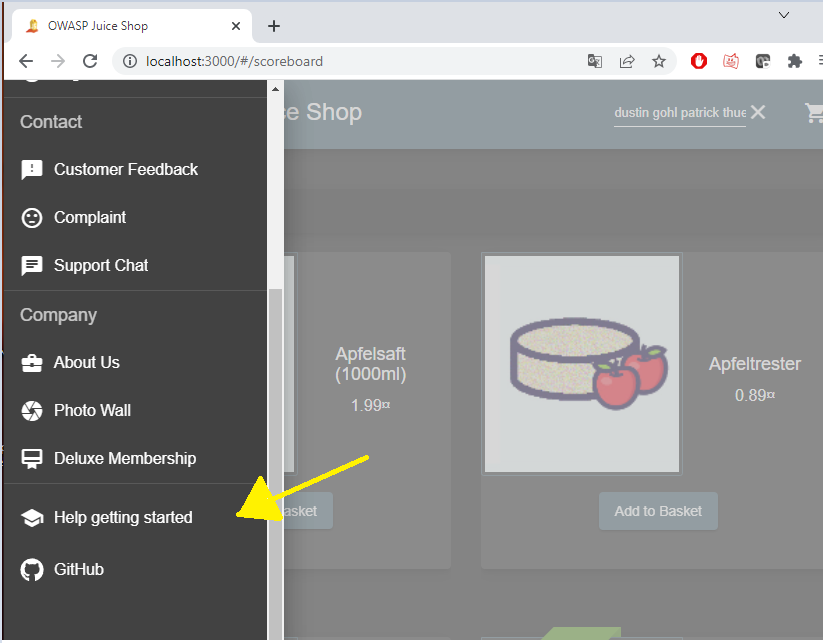
Created by: Dustin Gohl and Patrick Thuemer

Part 1. Juice Shop

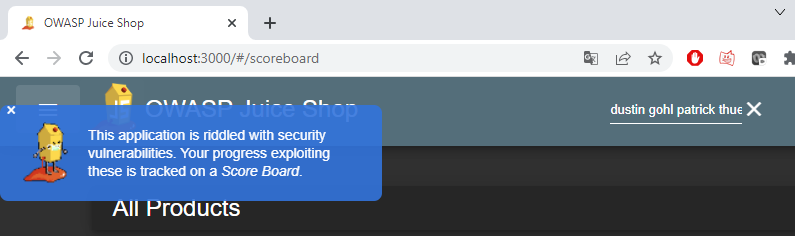
After setting up and starting the application, we analysed the Juice Shop and tested the different functions of it. We tried to get a good understanding of the shop and explored the menu items. Afterwards we began with the first task of laboratory.

Search and access the hidden Scoreboard:

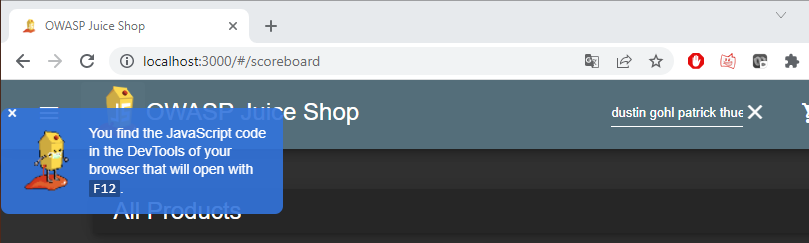


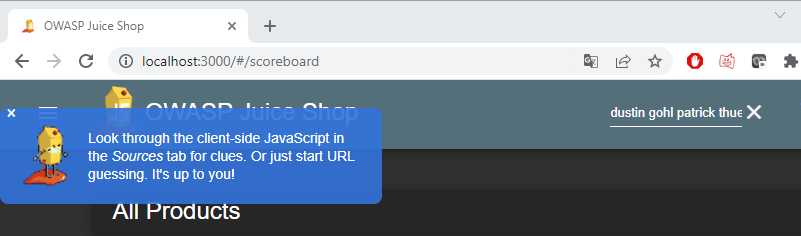


We noticed the menu in the top left corner. In the menu, we found the topic “help getting started” that we used to find the hidden scoreboard.

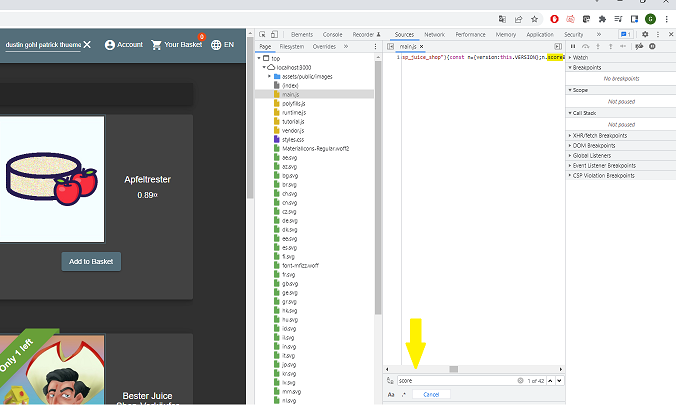


After clicking on the menu point, a tutorial opened, that explains in detailed steps how to find the hidden scoreboard. We followed this tutorial step by step.

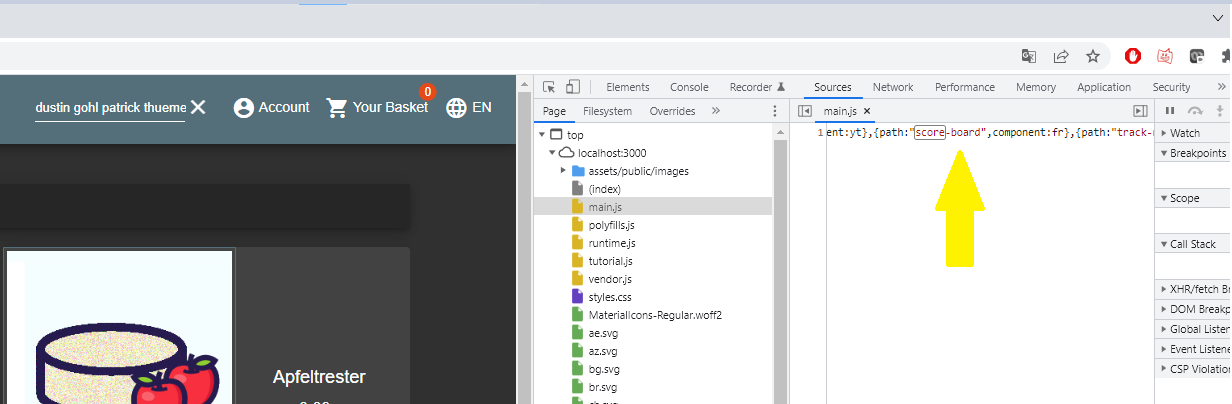


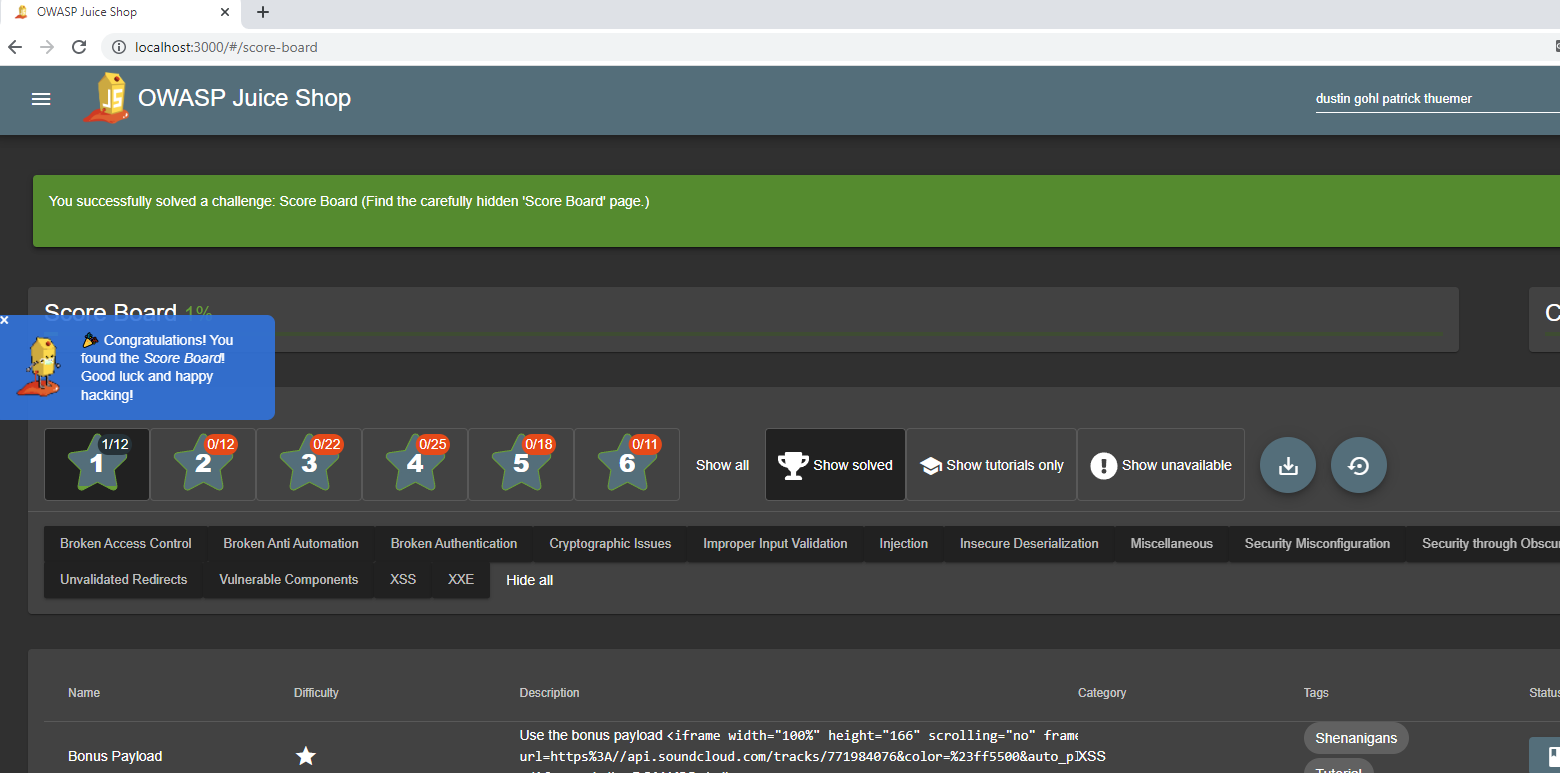


The tutorial explained that the JavaScript code of the application can be accessed in the development tool that can be opened by pressing F12. After opening the dev tool. We should search for clues in the JavaScript of the web-application in the source tab.



Here we could look through the different files and we searched for clues. We guessed a clue probably must be in the main.js file, so we analysed the code inside it. By pressing STRG+F we searched through the files with the catchword “score” and went over the different results.

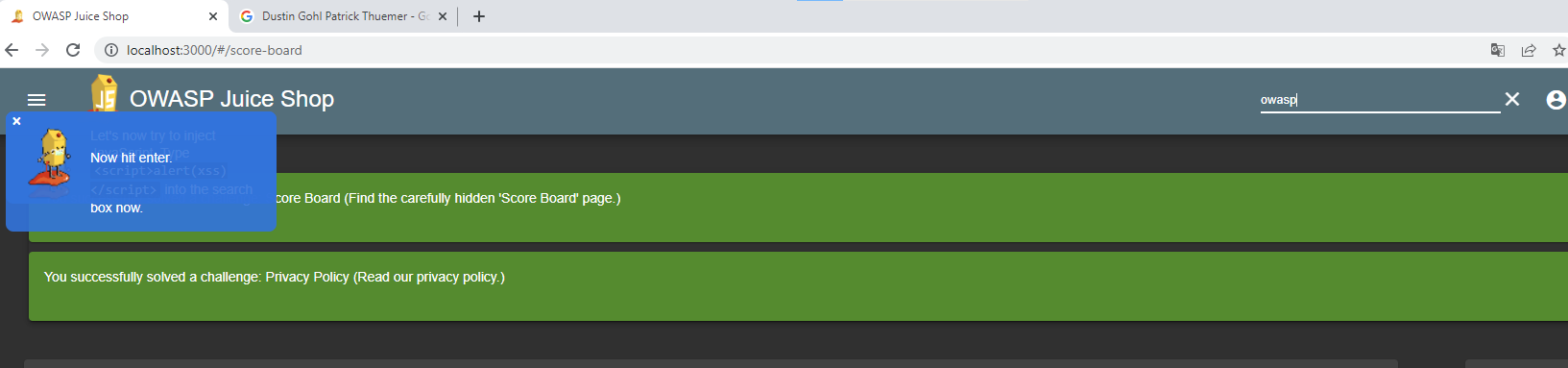
The twelfth search result included “(path:"score-board") that gave us a clue how the URL of the hidden scoreboard must look like.



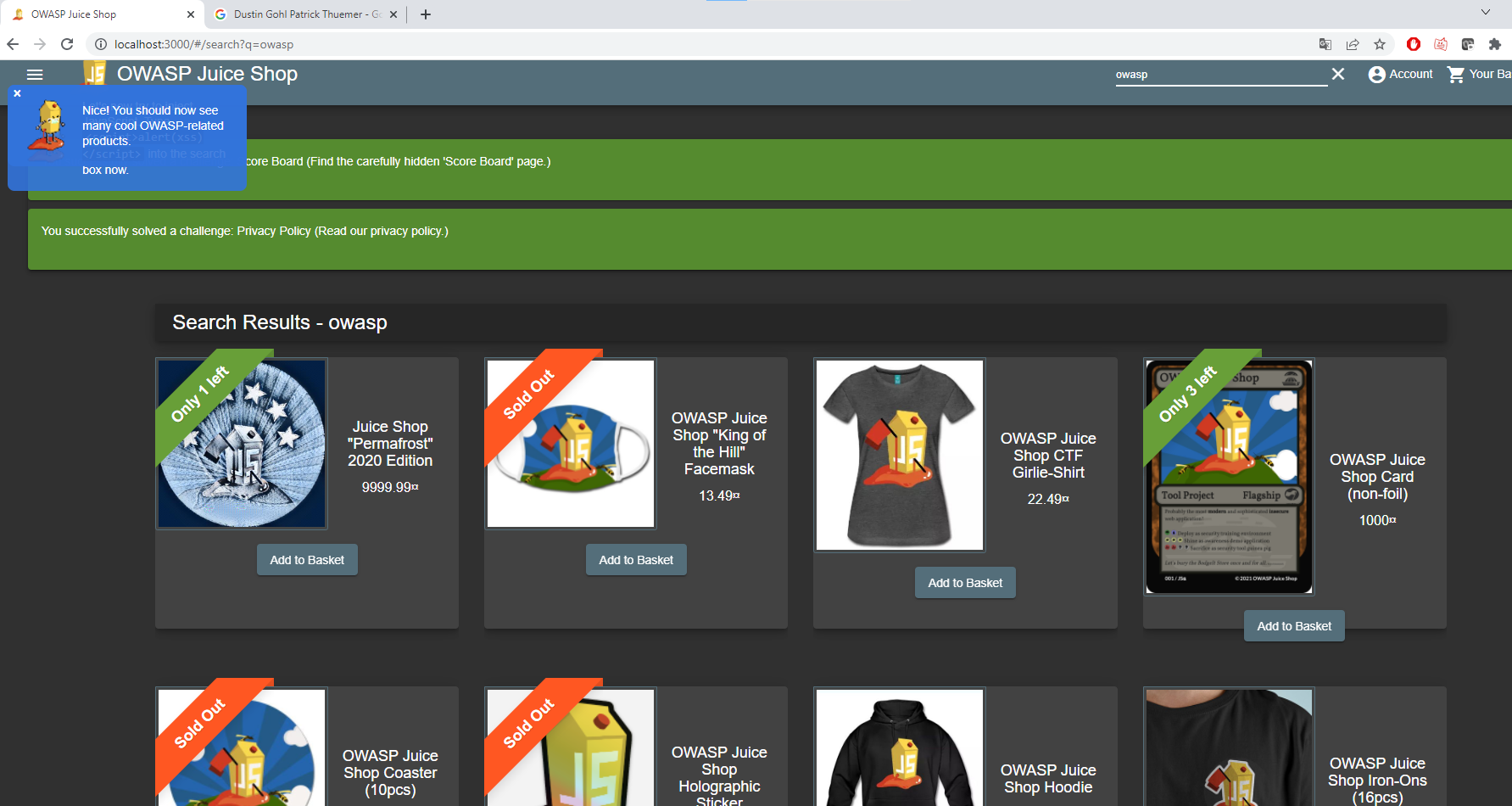
After testing the URL followed by the path (localhost:3000/score-board), we found the scoreboard and the scoreboard item got added to the menu.

DOM XSS

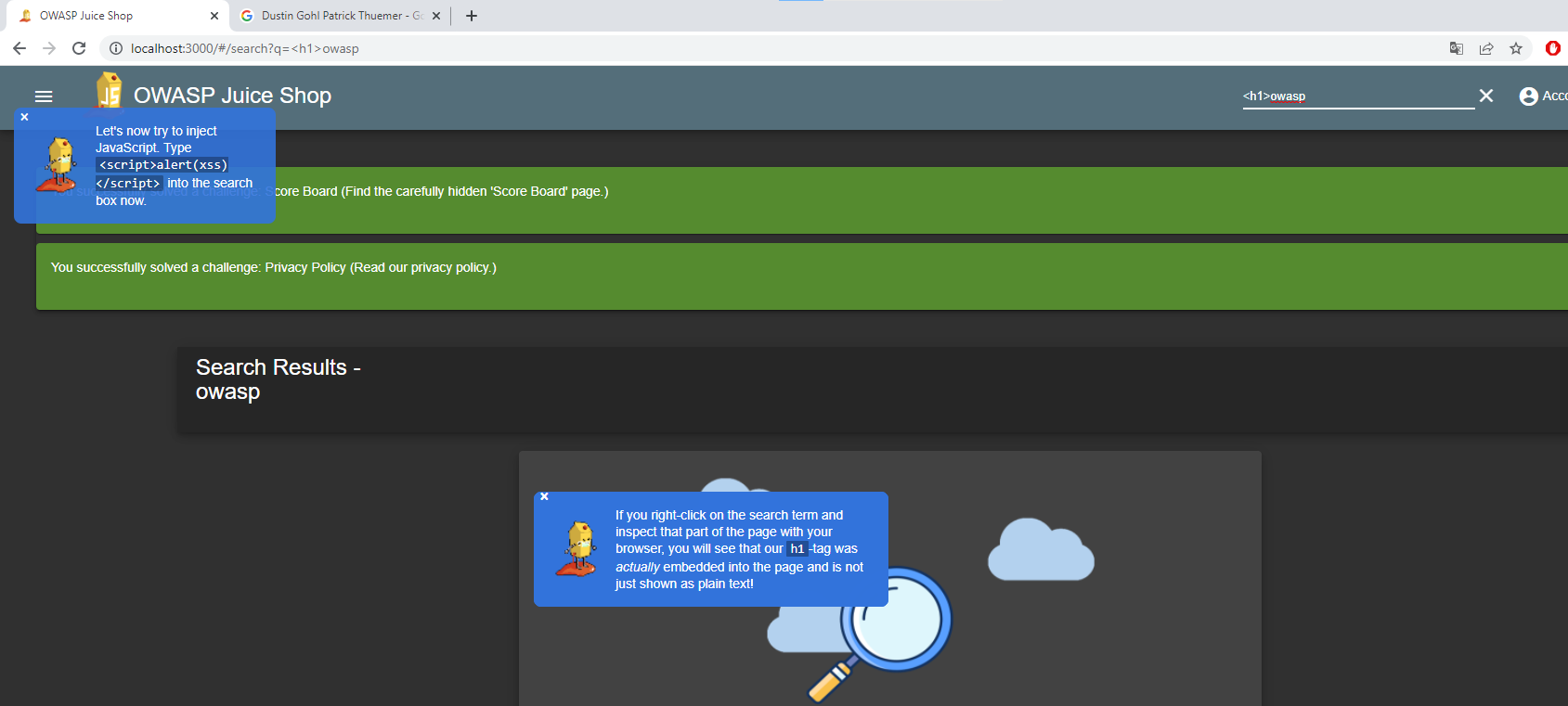
The first vulnerability we exploited was the 1-star problem DOM XSS.

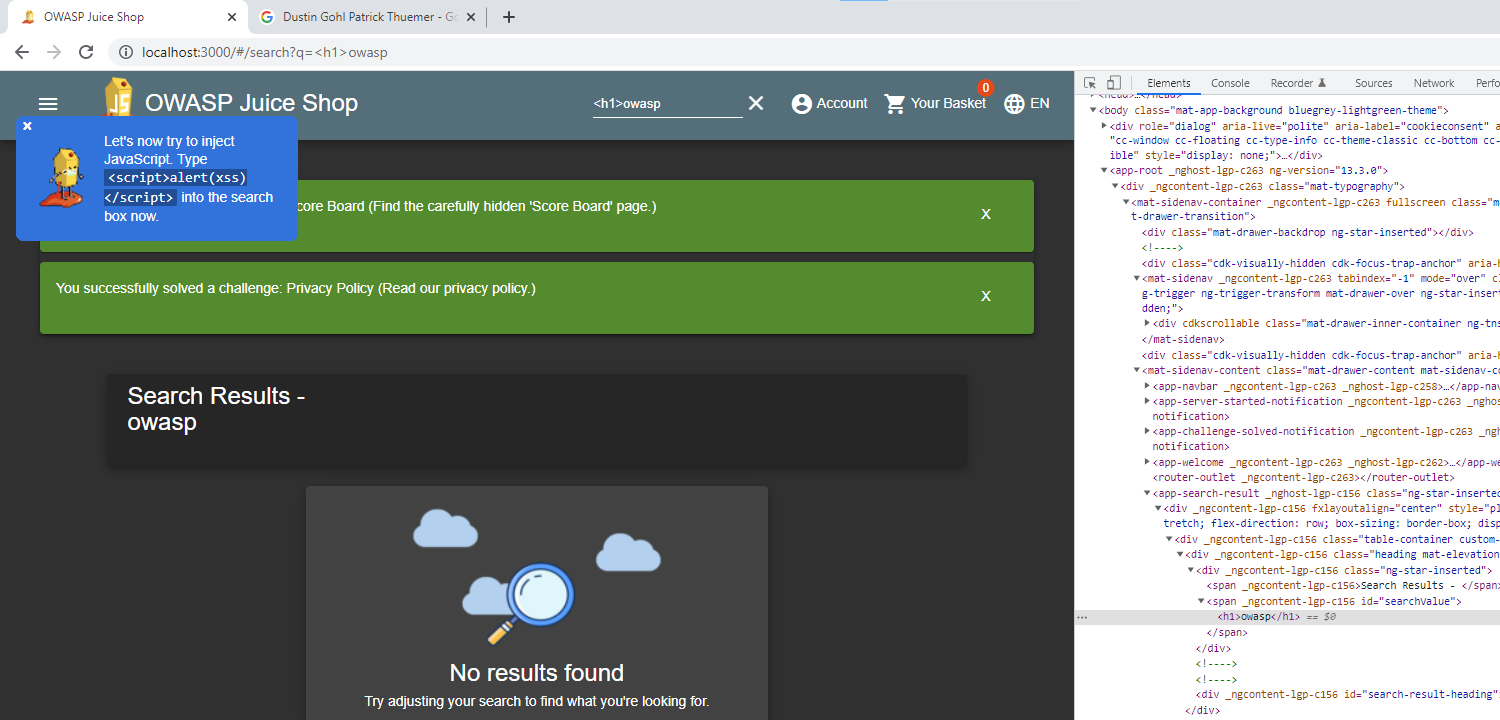


The first step was to look at the search bar in the top right corner and input “owasp” into it.

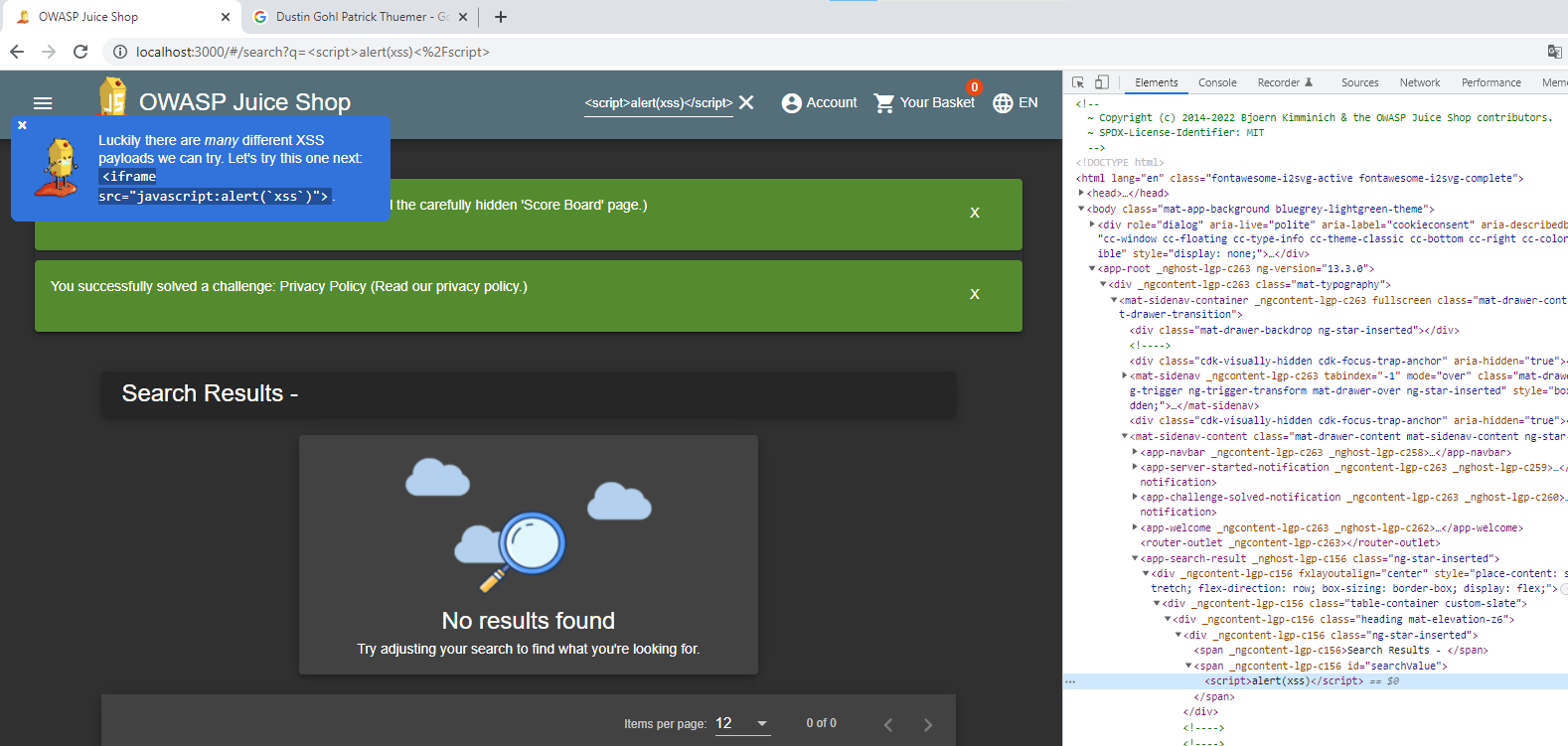


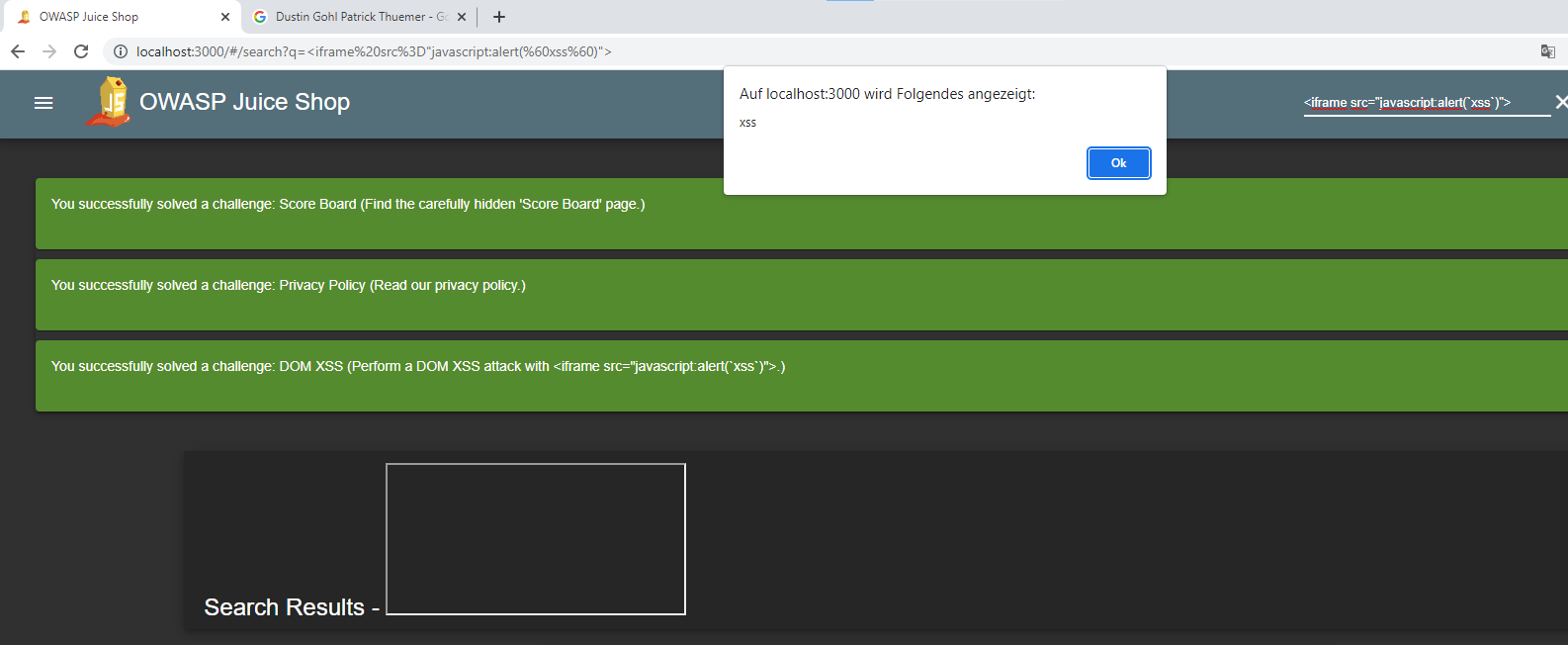
Afterwards, all owasp related products of the shop are shown. The search term is displayed right above all the products. Because of this, it might be possible to do a Cross-Site-Scripting attack, by injecting HTML or JavaScript code in the code.

We tested this by inputting “<h1>owasp”´. Now we had to check if the injection worked and the code is in the application.



We can check this by right-clicking on owasp and pressing “Inspect”. Afterwards, the DevTools opens and we automatically get to the element right-clicked. There we see that our input got injected and a new <h1> got added.

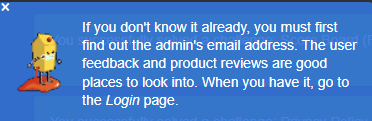
Now we tried to inject JavaScript. We input <script>alert(xss)</script> and searched. The script did not work but by inspecting the page, we can see that the script got added, but is not getting executed.

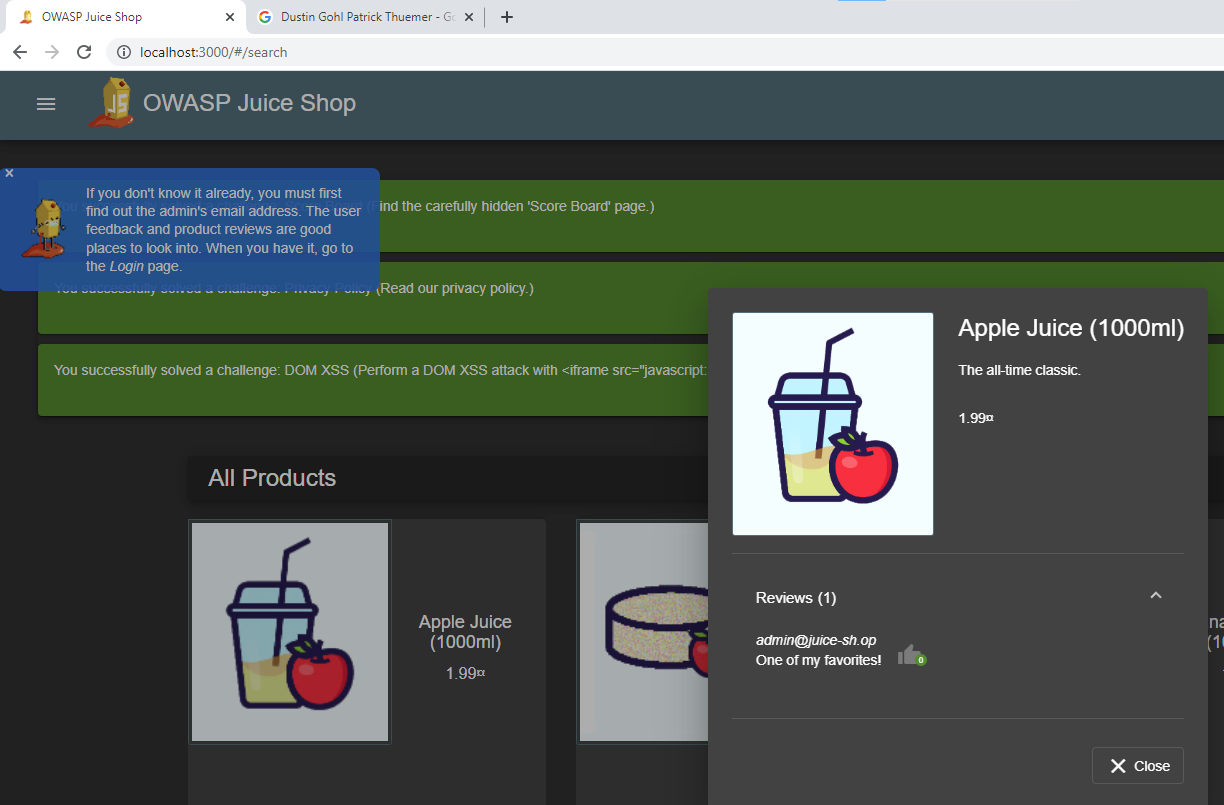
Next, we tried out another XSS payload by inputting <iframe src=”javascript:alert(‘xss’)”>. Now the input gets executed and we get an alert. We have injected code and carried out an XSS DOM attack.

In the real world, this vulnerability could be a problem because an attacker would be able to embed a malicious script in the application. The browser of a victim could execute the script. As an example, the attacker could steal data from the user or download files on the victim's side.

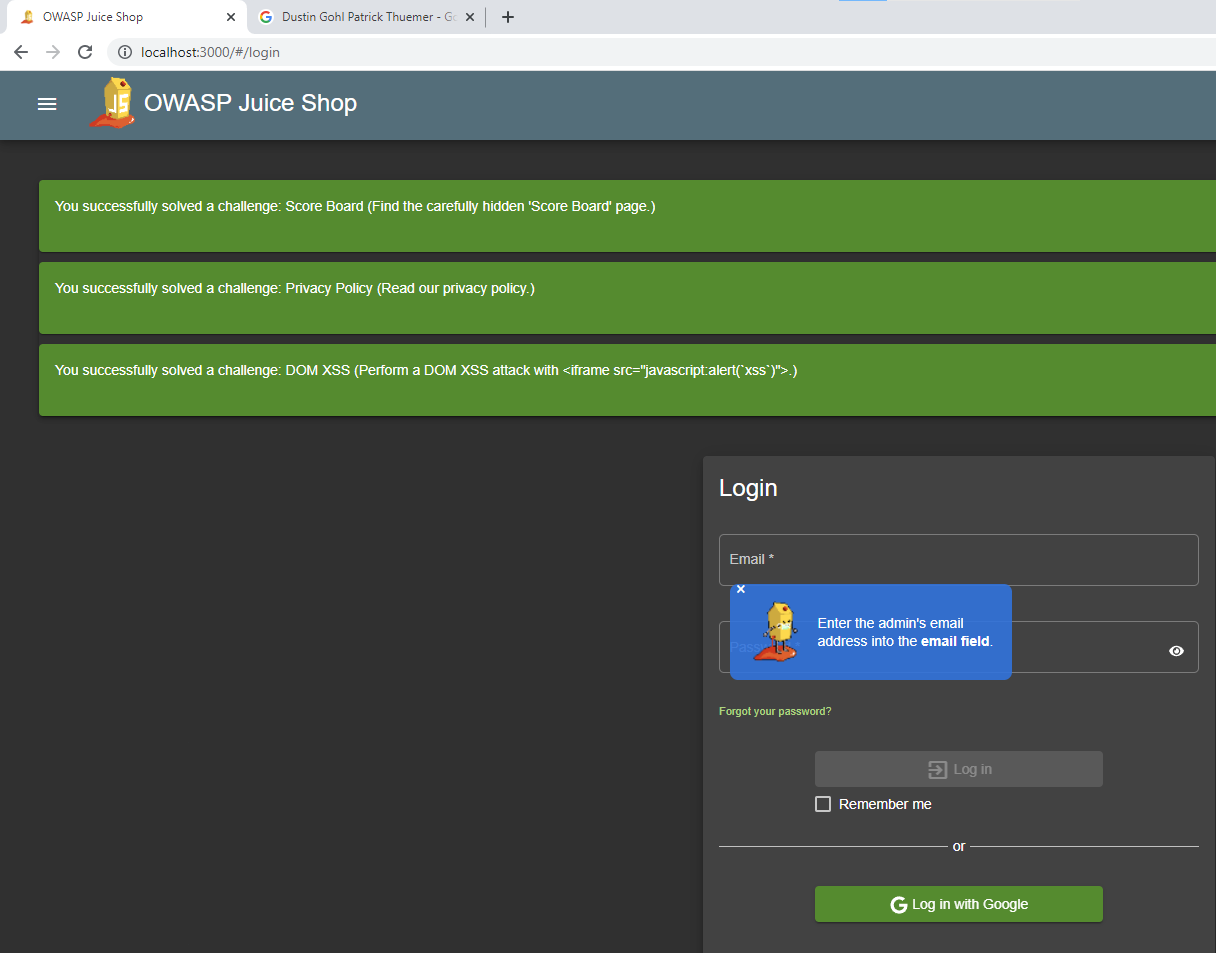
Password Strength Problem

The next vulnerability we choose was the 2-star problem Password Strength. In this attack, we wanted to exploit the vulnerability to login as an admin.

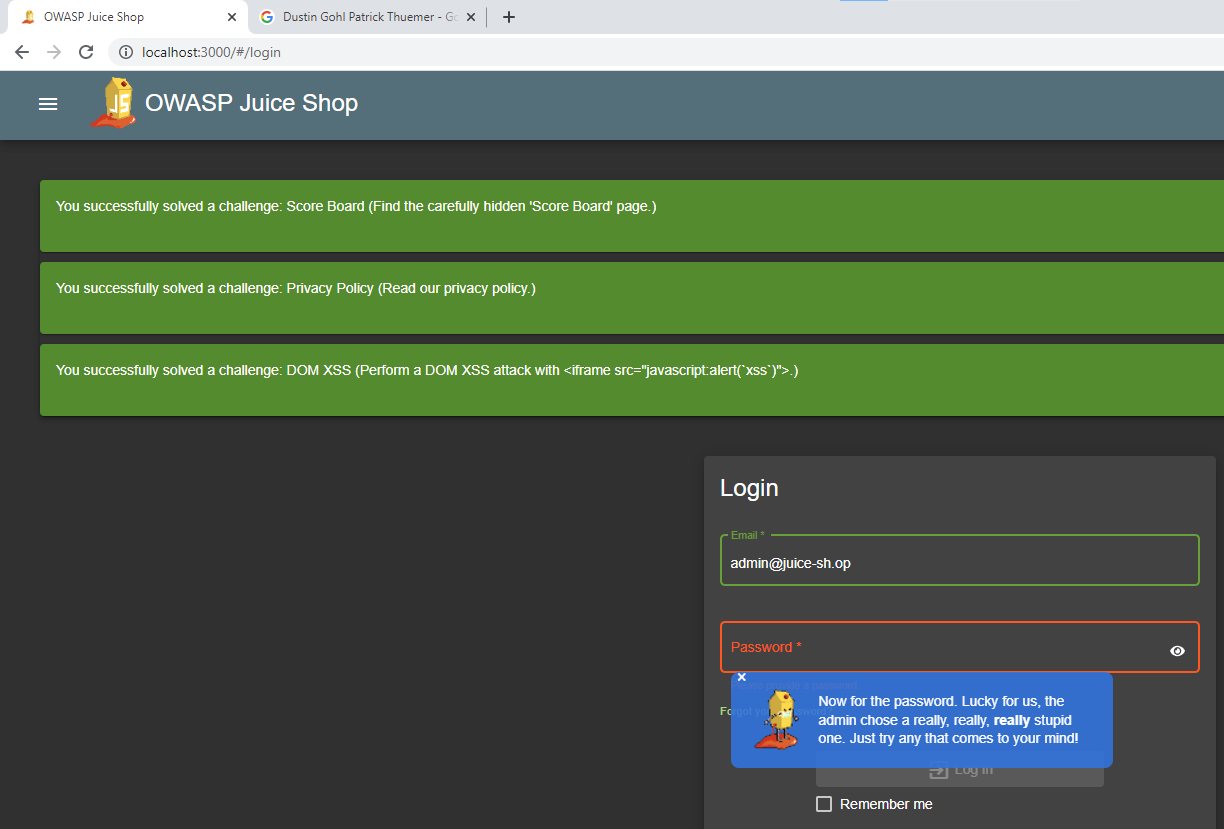




First, we needed the e-mail of the admin to be able to log in. By searching through the website, we found the admin’s email address in a review of one of the products.



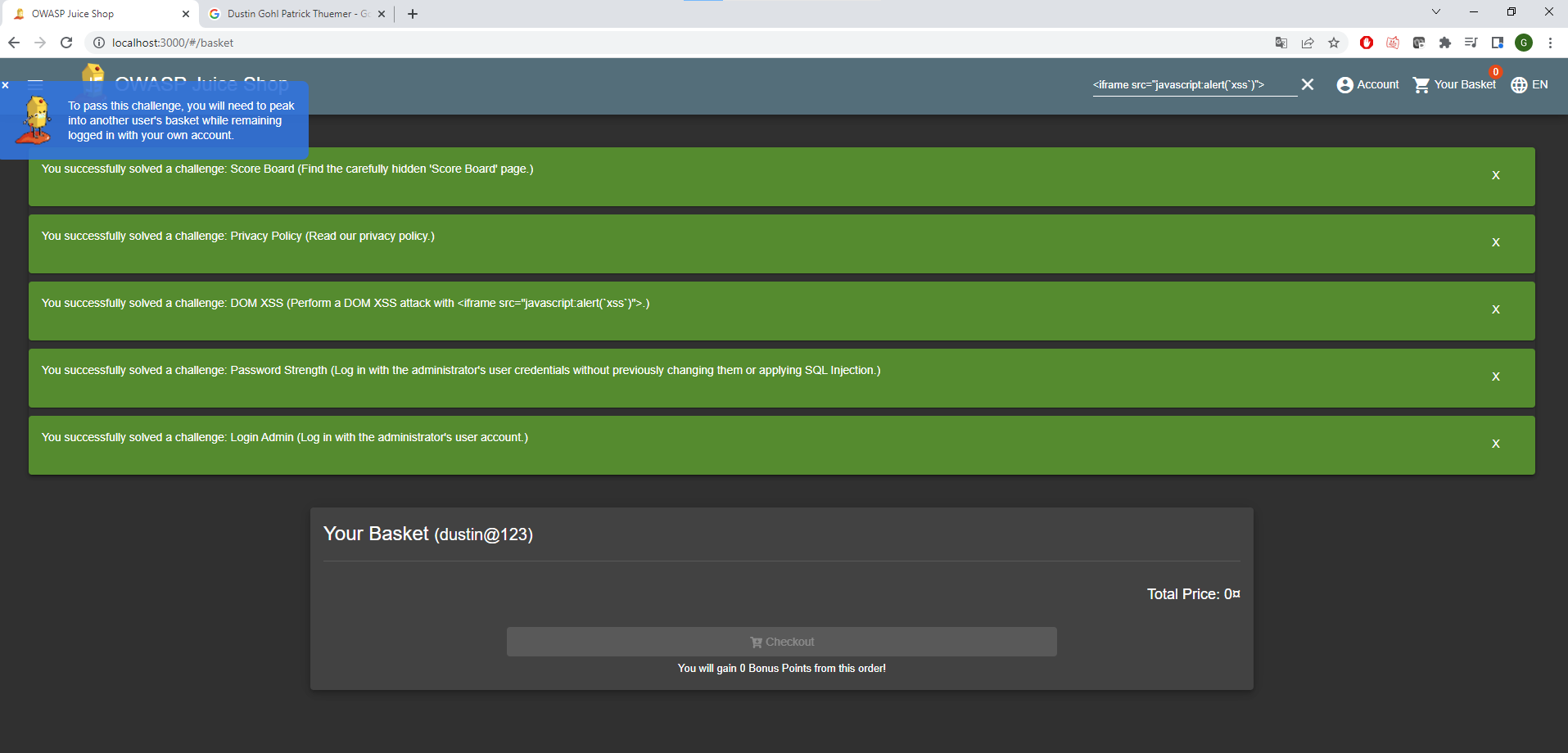
Afterwards, we went to the login section and input the email address.

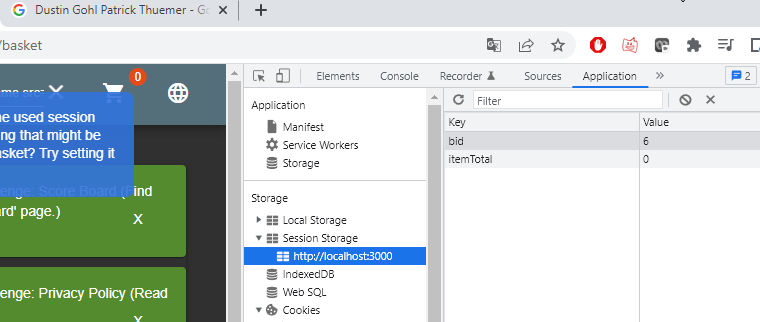
In the next step, we got a hint by the tutorial that we had to guess the password. First, we tried out a row of numbers like “12345”, “123” or “00000”. Since the numbers did not work, we tried combinations of numbers and text. Finally, we got the password “admin123” and were able to login in to the admin account and complete the task. During this task, we completed the 2-star problem “Login Admin” as well.

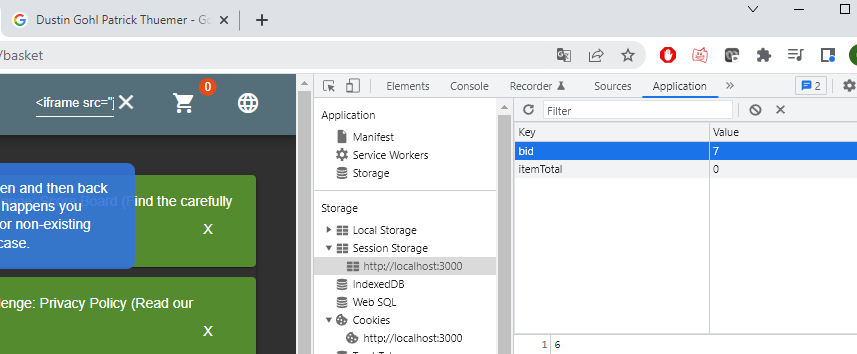
This vulnerability could also be a problem in a real-world application because an attacker could almost do anything in the application with the access of an admin account. Given the different functions the admin has in the web-application the attacker could take the application down or change parts of the application.

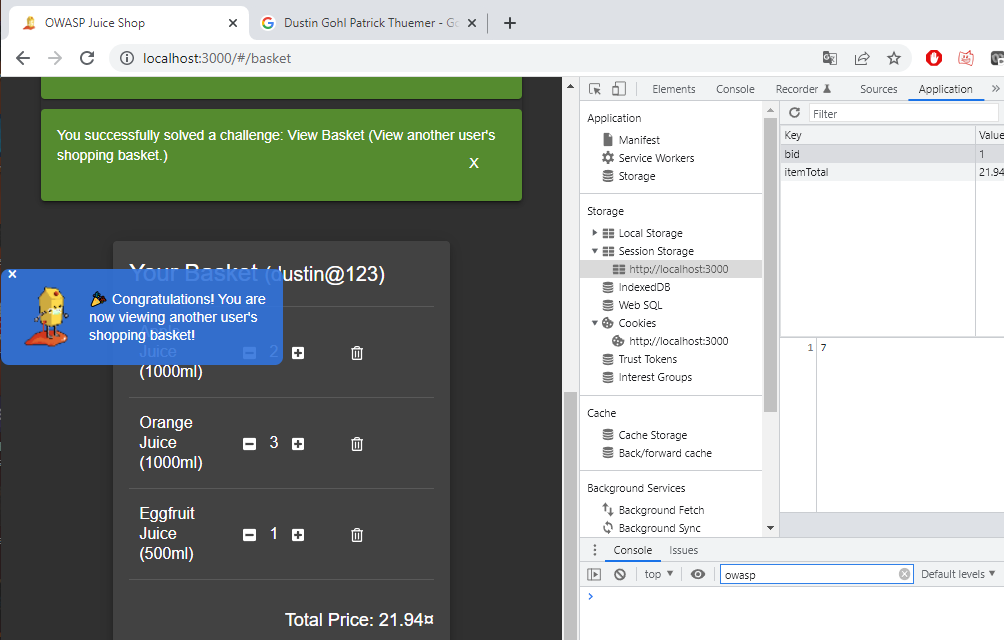
View Basket

The last task we completed was the 2-star challenge “view Basket”. In this challenge, we want to access data or basket of another user.

First, we went to our Basket, so we got a comparison for other baskets to check if the basket changed. The logged in account had an empty basket.

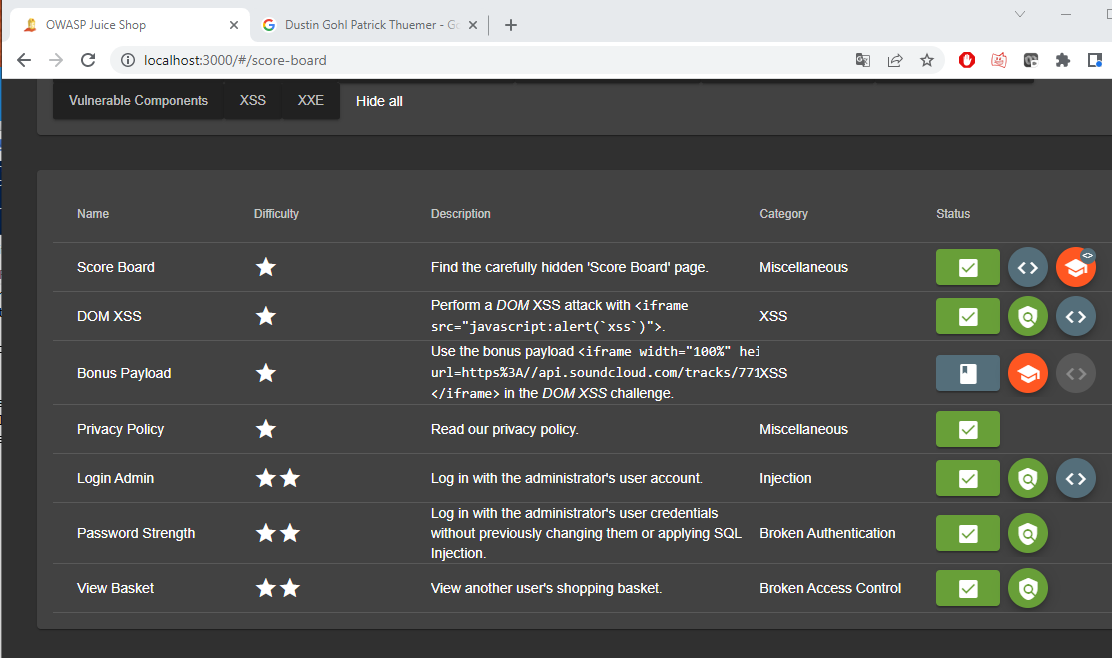
Now we looked for references of the basket in the application. A stored reference might be a possible attack vendor. We opened the Dev Tools and accessed the Session Storage tab in the section application. In here, data is stored in key/value pairs.

In the Session Storage Tab, we found the key “bid” that might be the ID of the basket. By changing the bid value, we may be able to access a basket of another user.

We tried multiple bid values and after changing the value, we had to go to another screen and back to basket every time. If nothing changed, we have set a non-existing bid. Eventually, we input the bid “1” and got access to the basket of another user.

This vulnerability could be a huge problem in an application. In our case we only accessed the data of a shop basket, but in a real-world application this data could contain personal information's of a user like credit card information or passwords.

Scoreboard after the tasks

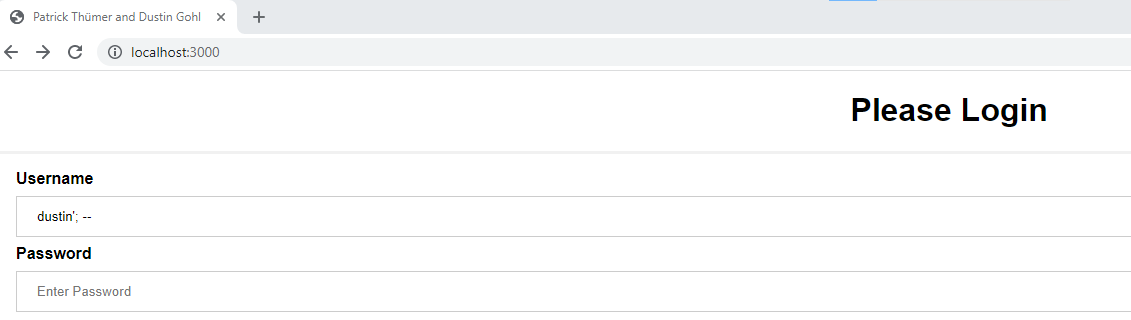
After completing all the tasks our scoreboard looked like this.

Part 2: Build Your Own Vulnerable Web Application

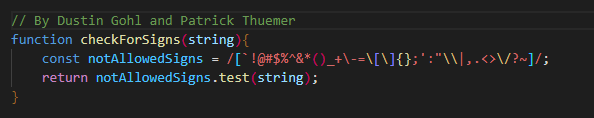
We decided to create a web application for the creation of notes. The application has a register and login, so a user is able to authenticate and login to the application. After the login, the user gets send to the home page. On the home page, all notes of the user get outputted and the user can create a new note by inputting a text in an input field and pressing the button “Save”. The accounts and notes are saved in a MySQL database. To be able to open the application in a browser, we created a node.js server in JavaScript that works with the framework express.js.

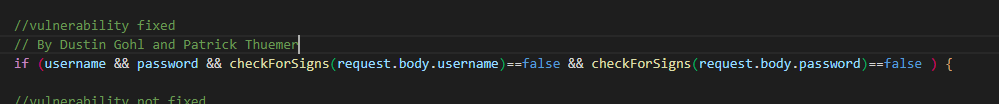
SQL Injection

The SQL Injection is a vulnerability, a User is able to modify the SQL Queries that are used in the application. With the injection, the attacker might be able to get full access to the database and it´s data. In our application, it is possible to do a SQL Injection in the login section. It is possible to inject in the box of the username input.

One example is to inject into the username box, so no password is needed and the user gets access to the account without knowing the password. We knew that the username of the account was ”dustin”, but we didn’t know the password. By adding “’; -- “we changed the SQL query of the application and were able to login without knowing the password.

The characters “-- “comment everything behind it, so in our application no password is queried. Similarly, it is possible to add any SQL Code the user wants in to the username box. This vulnerability creates many risks and a user could modify and access the database. An example for this would be, that a user adds SQL code to drop a table of the database and deletes all the data within it.

To mitigate this vulnerability, we would have to filter the input of the boxes and check if the input is ok to be used. A function that tests if a string contains one of the specific characters of “notAllowedSigns”. If it contains one, the function returns true and if it contains none it returns false.

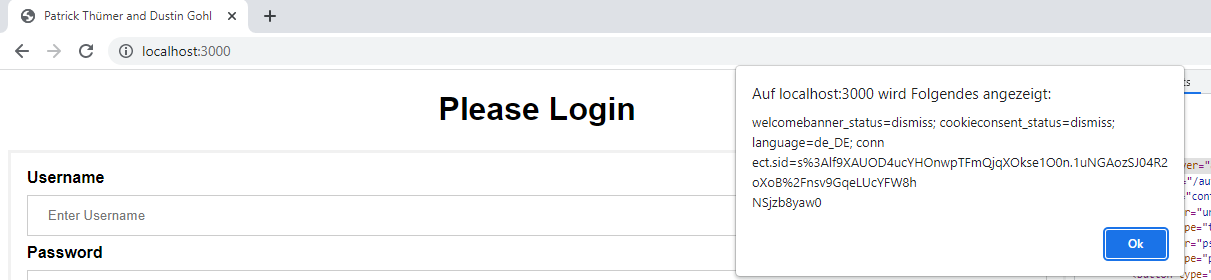
With this function, it would be possible to check the inputted username and password and check for not allowed signs. If it contains one of the signs, the user gets redirected to the login screen and has to input again.

XSS attack

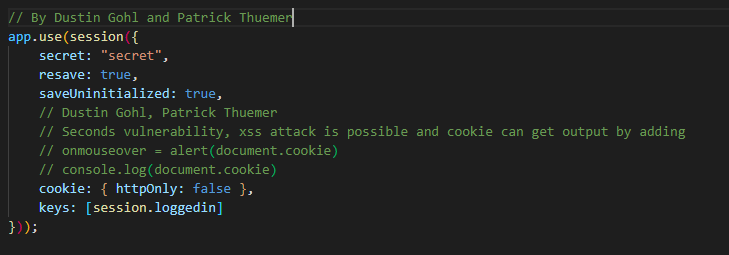
For the transfer of the backend variables to the frontend, we used the response.render() function in express. This function gets often used in combination with the file format .ejs, which allows the transfer of data to different sites. The setup of a .ejs file is quite similar to the one of a normal HTML site. With the syntax <% %> backend variables can be accessed in the frontend. The render() function as well as the .ejs format include multiple safety aspects against XSS attacks. As an example, client signs like “</>” get outputted as strings and don't get recognized as JavaScript commands by the browser. For the creation of notes, we used the render() function, so attacks would get more difficult. For the implementation of the XSS attack, we used the send() function of express in some cases. The function does not include any security aspects for XSS attacks. On the other site the transfer of variables is more difficult with this function.



To exploit this vulnerability, someone could change the element of one of the elements of the site, so the site outputs the cookies when the user hovers over the element. We exploited this vulnerability by changing the properties of the headline “Please Login”.



After hovering over the headline, the site outputs the application cookie “connect.sid” and the vulnerability got exploited.

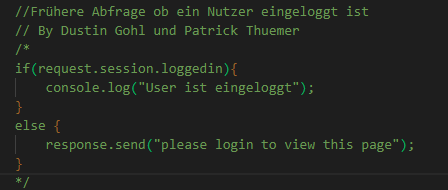


To be able to exploit this vulnerability, we first had to disable one of the security aspects of express. We had to set the variable “httpOnly” of the cookie on false. After this change, it was possible to access the cookie “conned.sid” and to exploit the problem. The mitigation for this vulnerability would be to set the Variable back to true, so it gets protected from these attacks.

Broken Access Control

In our application, it is possible to exploit the vulnerability broken access control by changing the URL to open a specific site without being logged in.

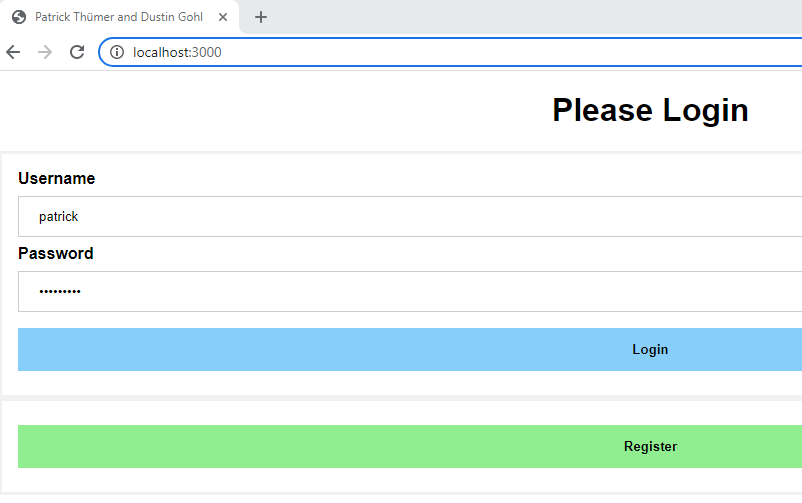
By opening the URL localhost:3000/home the user is able to access the home site without being logged in. The attacker is able to create notes on the site and can add them into the database without even having an account.



To mitigate this vulnerability, we would have to add a query on every site of the application that checks if a user is currently logged in. If there is no user logged in, the site should get blocked and the application sends the user back to the login or it notifies him that he is not logged in and therefor can't see the content of the site. A simple query to check if a user is logged in could look like the code above. Instead of the console.log we would have to fill it with the functioning code of the site.

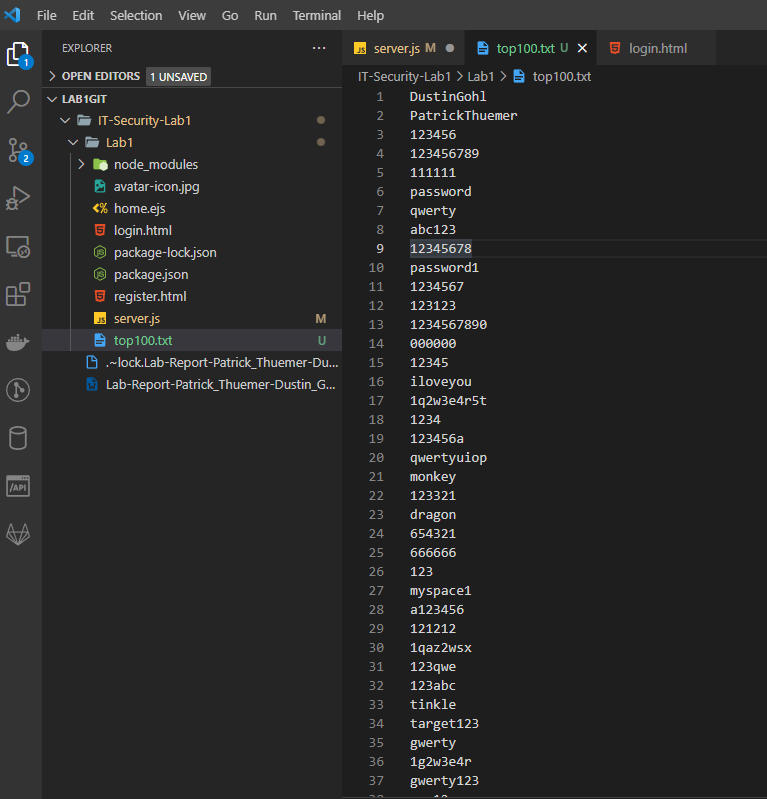
Broken Authentication

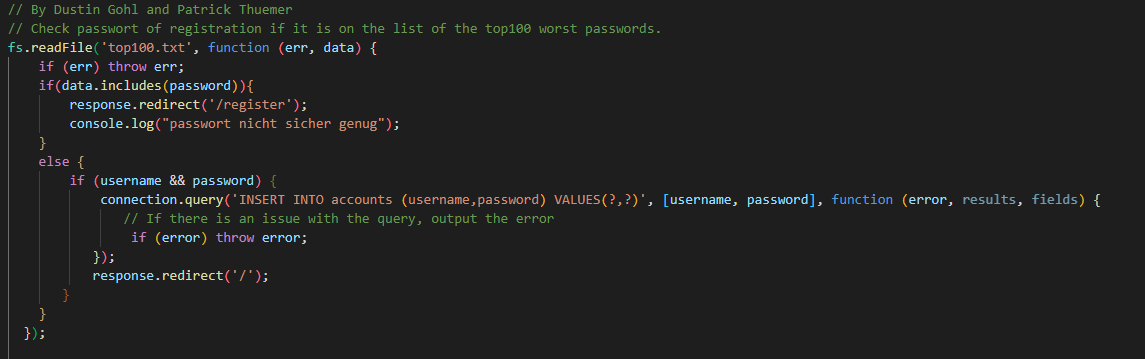
We exploited this vulnerability by abusing the possible bad password of a user. In our case we had the username “patrick” of an account, but we didn’t know the password. To exploit the broken authentication, you could run a script, that tries logs in by trying a list of passwords of the top 100 worst passwords.

In this project, we went manually through the top 10 list and found that the password is “password1”.

These lists can be found on GitHub:

<https://github.com/danielmiessler/SecLists/tree/master/Passwords>



The application does not check new passwords at the registry. Therefore, it is possible that a user might use an awful or simple password that is easy to guess. To mitigate this vulnerability, we have to restrict the password a user can use at registration. The password has to be stronger and should not be on the top lists of worst and most used passwords. To filter the password, we wrote a function that compares the password with a list of a passwords. If the password is not on the list, the user is allowed to use it. If the password is on the list, he has to use a better password.

CVSS Ranking of the vulnerabilities

Ranking was done with the CVSS calculator on the website <https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator>.

