

# Secure Coding and Vulnerability Detection - DDSS Assignment #2

José Donato, donato@student.dei.uc.pt, 2016225043

**Abstract**—The number of internet users have increased by more than 500% since 2000 and it is a number that doesn't seem to be stopping anytime soon. Also, there are more than 3 billion internet users worldwide [1]. More than ever web applications are being developed and used. Although the best secure coding practices aren't being followed and it results in vulnerable applications with endangerment not only to the users but also to the owners of the app. To solve this problem the best practices must be used and developers must be aware of the risks when coding an application that will be available to every 3 billion internet users.

## I Introduction

The goal of this report is to show the relevance of secure coding practices and why testing and analysing the app developed is important. It is divided in two parts.

In the first part, I explain what practices I followed to develop the secure parts of the application and what vulnerabilities I left open in the vulnerable parts. I start by explaining each part of the application in a separate subsection and then I talk about some important features of the application (like Session and Multifactor authentication). I finish this section talking about the attacks and their prevention with ready to use exploits including proof of concept code.

In the second part, the results of testing and analysing tools to detect vulnerabilities are exposed and discussed.

The application is live on the following url: <https://ddss-assignment2.herokuapp.com/>.

## II Part 1 - Secure Coding Practices & Vulnerabilities

### II-A Introduction

Right after I receive the resources, I converted the NodeJS application from JavaScript to TypeScript <https://www.typescriptlang.org/>. Although I am not a long experienced TypeScript developer, its advantages are obvious. It brings optional static type-checking and the latest ECMAScript features always being compatible with JavaScript, i.e., JavaScript programs are also valid TypeScript programs. With a type-safe language we are less likely to make simple errors when coding.

The application can be accessed live in the url with HTTPS protection: <https://ddss-assignment2.herokuapp.com/> or running docker-compose up -d in the root folder of the project. All the private source code is available in the following url: <https://github.com/jose-donato/ddss2019>.

When the user enters the application he is presented with 4 different parts:

- **Part 0:** One register form where users can create an account
- **Part 1:** Two login forms (one vulnerable and one secure) where users can login with their credentials
- **Part 2:** Chat with two forms (one vulnerable and one secure) to submit messages and see other users' messages
- **Part 3:** Two forms (one vulnerable and one secure) to search books created in the application

### II-B Sign Up & Sign In

As said before, when the users enter the /part0, they can create an account. This form is safe and is similar to other forms in the application. It contains three input fields: one for username, one for email and another for the password.

- is protected against Cross-site request forgery (CSRF) attacks with the library: <https://github.com/expressjs/csrf>
- is protected against brute force attacks with the middleware: <https://github.com/AdamPflug/express-brute#readme>
- with express-validator (<https://express-validator.github.io/docs/>) I do a first input validation. I check if the username is a string and it has a length between 3 and 31 characters. The same is done to the password but with a length between 8 and 128 characters. The email is also validated with the same library (`req.body.email.isEmail()`).
- before creating queries to the database, I do another validation of the username and password
- with a parameterized query (from pg library <https://github.com/brianc/node-postgres>) I see if the user already exists. If not, I calculate the hash of the password using argon2id (with the library argon2 <https://github.com/ranisalt/node-argon2>) and do another parameterized query to insert the new user in the table users

When the users are in the sign up page, they have a suggestion of a safe password. This password is randomly generated with the help of the secure-random-password library (<https://www.npmjs.com/package/secure-random-password>) that claims to generate passwords using a cryptographically-secure source of entropy. I found this solution interesting since afterwards they have a safe password and can store it on a password manager such as bitwarden (<https://bitwarden.com>). Some companies like PortSwigger (<https://portswigger.net/>) are already starting to use this approach (provide the safe passwords and users have to save it in a password manager).

If the users choose not to use this safe password, the password that they write in the input field will go through another validation with the help of the password-validator library (<https://github.com/tarunbatra/password-validator>). I set up a schema to check if the password provided has at least eight characters including:

- uppercase
- lowercase
- number
- symbol
- zero spaces

I decided to use this library despite other solutions such as a simple regex verification because of Regular Expression Denial of Service (ReDoS) attacks. If I used a regex verification to validate the passwords it could lead to a ReDoS attack.

If all input fields are correct, the account is created, a verification email is sent and the users are redirected to the next part: part1.

Before jumping into the sign in page, let's talk about the verification email. It is sent with nodemailer (<https://nodemailer.com/about/>) and inside has a confirmation link that has the following format:

[https://ddss-assignment2.herokuapp.com/confirmation/\\${emailToken}](https://ddss-assignment2.herokuapp.com/confirmation/${emailToken})

The emailToken is a JSON Web Token (this report have a detailed section about them below). When the users click the link, their email is confirmed and they are redirected to the login page (/part1).

In /part1 we have the login forms. To take care of the authentication part, I followed the pseudocode suggestion provided in the OWASP cheatsheet [2]:

#### Algorithm 1: Login Flow

```
if USER_EXISTS(username) then
  password_hash=HASH(password);
  IS_VALID=LOOKUP_CREDENTIALS_IN_DB(username,
  password_hash);
  if NOT IS_VALID then
    | RETURN Error("Invalid Username or Password!")
  else
    | //If reaches here, successful login, send JWT
  end
else
  | RETURN Error("Invalid Username or Password!")
end
```

Note that both when user doesn't exist and credentials are wrong it returns the same message, not providing an attacker any information about the users in the web application. If an attacker wanted to know which usernames are registered in application it is impossible because it will return always "Invalid Username or Password!" and not something similar to "User already taken".

In the correct (safe) form, I setup a protection against brute force attacks (with express-brute). This way, if malicious attackers use automated tools to try different combinations of passwords they will get blocked from accessing the web application.

On the other hand, the vulnerable form isn't protected against brute force. I made a simple script in nodejs to bruteforce this vulnerable form and get the correct password for a certain user (explained in later section).

In the safe form, if the user confirmed his email and the credentials are correct, he is presented with another level of protection: the two factor authentication (explained in later section).

If the user succeeds to prove his identity gets a token cookie (JSON Web Token) associated to his requests and is redirected to the next part: part2. In case the user fails any of this two steps, the attempts are registered in the system and sent to the user by email.

As I already said, the vulnerable login form isn't protected against bruteforce attacks which is a big vulnerability. Also, although it is not possible to login using simple SQL injections such as "' OR 1=1-" the form is not protected against other SQL injections. Since the query associated with this specific form isn't parameterized, an attacker can easily send SQL commands that will be executed in the database such as: "' DROP TABLE users-". An attacker wouldn't know the name of the database tables without more attacks but it is easy to guess some common table names. In case the attacker guesses the right table name, the losses would be catastrophic. A possible mitigation for this problem is suggested in the section **II-F. Changes in Database**.

In this vulnerable form, if the user prompts the correct credentials, gets a token cookie and directly redirects to the next part without passing the two factor authentication and without the need of having the email verified.

## II-C Part 2 - Insert Message Forms

In this second part of the application, the user is presented with protected contents, i.e., the user needs to be authenticated to see this (part2) and the next part (part3). This is achieved using JSON Web Tokens (JWT). Its implementation is explained in later sections. Very briefly, if a request has a valid JWT associated, the user is authenticated. Otherwise he is not and gets redirected to the login page.

In the /part2, the user is presented again with two forms: one vulnerable and one correct. In both forms, any logged user can submit messages and see those messages in the table below the forms. The differences between the two forms are similar to the ones in the login forms. The safe one is protected against:

- 1) SQL injection (SQLi): using pg parameterized queries.
- 2) Cross-site scripting (XSS): all the messages that comes from the safe form are escaped before going into the html with a library named escape-html (<https://github.com/component/escape-html>). To achieve this, I select all the messages from the database and see if the message come from the safe or vulnerable form. Depending on the result, the message and the author are escaped or not:

```
10 import escapeHtml from "escape-html";
11 const getMessages = async (client: any) => {
12   const queryString = "SELECT * FROM messages";
13   pinoLogger.info("query in part2: ", queryString);
14   const result = await client.query(queryString);
15   if (result.rowCount > 0) {
16     return result.rows.map((m: any) => ({
17       ...m,
18       author: m.author.includes("_vulnerable")
19         ? m.author
20         : escapeHtml(m.author),
21       message: m.author.includes("_vulnerable")
22         ? m.message
23         : escapeHtml(m.message)
24     }));
25   }
26   return null;
27 };
```

The same effect could be achieved with the EJS library (<https://ejs.co/>) when presenting the results in the HTML. If I used `<%= result %>` instead of `<%- result %>`, the result would be automatically escaped.

- 3) Cross-site request forgery (CSRF): is an attack where with some social engineering, the attacker can make a legit authenticated user submit data to a form without noticing it. To protect against this type of attack I used the csrf library (<https://github.com/expressjs/csrf>). It is a middleware that runs on the routes with forms and generates a valid and unique CSRF token to insert in the form on the HTML code:

```
<input type="hidden" name="_csrf"
value="<%=csrfToken%>">
```

This way, only forms with valid csrfTokens will be executed.

Meanwhile, the vulnerable form is only protected against SQLi with the parameterized queries. It is vulnerable to both XSS and CSRF attacks. Later in this report there are sections with examples of this attacks.

## II-D Part 3 - Search Books Forms

In the last part of the application (/part3) we have again two forms: one vulnerable and one safe. Again, this part is protected, i.e., only

authenticated users can access it.

The forms allow the users to search for books that exist in the database. The logic to achieve the features in this forms was complex and involved SQL queries. Although, lot of parameters in the form were optional. Because of this, using parameterized queries from "pg" for all the parameters was not possible. Since they were optional, I see if the parameters actually exist and append it to a string that contains the whole SQL query.

In the safe form, to prevent SQLi attacks, the integer/float parameters such as the minimum price, maximum price or days of the date were converted to numbers with parseInt (or parseFloat in case of minimum and maximum prices) and the string parameters (title and authors) were added using "pg" parameterized queries. Note that before passing the parameters to the function that makes the SQL query, all of them go through an input validation with express-validator library.

Because of the complexity of the logic "Match with", it wasn't done using SQL queries. From the results after the SQL query, I filter the data with some recent ECMAScript functions (such as some(), filter() or map()) to achieve what was intended (all the operations are in the file /src/utills/operation.ts).

The vulnerable form, although is protected against CSRF attacks, it has SQL injection vulnerabilities. With simple "' OR 1=1--" in some input fields we can get all the results or do more catastrophic database operations such as table drops or changes.

## II-E Hashing Function - Argon2

To store the users' password in the database the winner of password hashing competition was used: Argon2 (<https://www.npmjs.com/package/argon2>) [3]. Instead of storing the plain text password (very bad practice) or using solutions with known vulnerabilities like bcrypt [4], I store a hash of that password using the hash function named Argon2 from the library node-argon2 (<https://github.com/ranisalt/node-argon2>).

To create the hash is as simple as:

```
const hash = await argon2.hash("password");
```

To compare the plaintext password to perform login operations we just need to use verify function from argon2 that return a boolean where the password provided by the unauthenticated user matches the hash stored in the database or not.

```
const IS_VALID = await argon2.verify(hash, "password");  
//check if plaintext password is equivalent to hash
```

## II-F Changes in Database

The database provided by the teachers suffered some changes. The first change was remove serial IDs and use UUIDs. Using serial IDs, i.e., auto incremental integers can be problematic. UUIDs, on the other hand, provide a solid solution [5].

Password length was also changed password in table users to match 128 as recommend in OWASP Cheatsheet [2], with the goal of preventing long password DOS attacks.

A 2FA key (twofa\_secret) was added to this table when implementing the two factor authentication system.

In order to add email verification and support, the fields email and confirmed were also added to the table.

Although I did not implement it, to hinder the attacker's work, random strings could be added to the table names in order to make hard to guess them. For example, the table users would be called "users\_sositdnnoh6xheiff\*whnQpnnjcmkdb".

## II-G Session, Email Verifications & JSON Web Tokens

Session is an important topic in web applications. It is the set of data structures that are used to track the state of the user's interaction with the application [6]. To handle session, I used JSON Web Tokens (JWTs).

Right after user signs in, it is sent from the server a cookie (this cookie has the httpOnly flag activated to prevent XSS attacks) that contains a JWT which includes the users' username. This way, we can identify a request and match it to the user that sends it.

If a request is sent without the cookie we already know that the user isn't logged in and redirect him directly to the login page.

Also, the case where a request has an invalid JWT (for example, a expired JWT) can happen. To prevent this, the jsonwebtoken library (<https://www.npmjs.com/package/jsonwebtoken>) has a method called verify that checks if a JWT is valid or not. When user logs out, the jwt cookie is cleared. Below there is the middleware that handles this logic:

```
1 import jwt from "jsonwebtoken";  
2 const verifyToken = (req: any, res: any, next: any) => {  
3   const token = req.cookies["token"];  
4   if (!token) {  
5     return res.render("pages/part1");  
6   }  
7  
8   if (verifyJWTFromCookie(token)) {  
9     next();  
10  } else {  
11    console.log("invalid jwt, redirecting to login");  
12    return res.render("pages/part1");  
13  }  
14 };  
15  
16 const authLogin = (req: any, res: any, next: any) => {  
17   const token = req.cookies["token"];  
18   if (!token) {  
19     next();  
20   }  
21  
22   if (verifyJWTFromCookie(token)) {  
23     console.log("user already logged in, redirecting to home");  
24     return res.render("pages/part2", { messages: [] });  
25   } else {  
26     next();  
27   }  
28 };  
29  
30 const verifyJWTFromCookie = (token: string): boolean => {  
31   try {  
32     jwt.verify(token, process.env.JWT_SECRET!);  
33     return true;  
34   } catch (err) {  
35     console.error(err);  
36     return false;  
37   }  
38 };  
39 export { verifyToken, authLogin };  
36 app.use("/part1", authLogin);  
37 app.use("/part2", verifyToken);  
38 app.use("/part3", verifyToken);
```

There are two possible cases:

- The user is authenticated: if he tries to visit login page, needs to be redirected to authenticated pages (verifyToken middleware)
- The user isn't authenticated: if he tries to visit protected pages, needs to be redirected to login (authLogin middleware)

In this application I used the JSON Web Tokens to generate the confirmation links for email verifications. Since we need a unique link for a user with a expiration date, this is another great use case for JWTs.

## II-H Multifactor Authentication

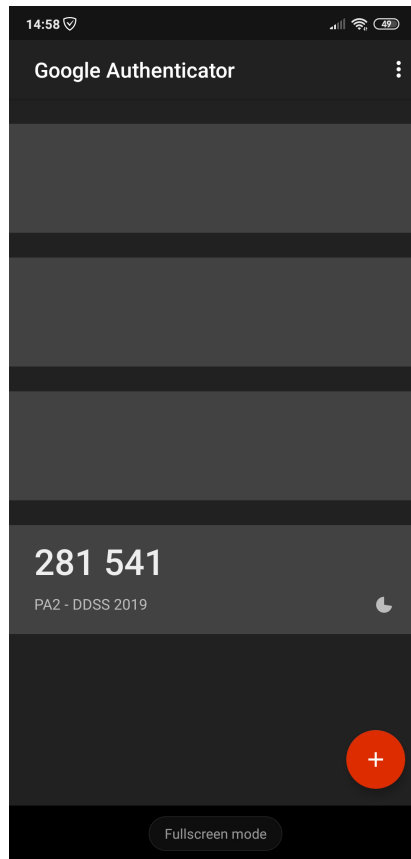
Authentication only with the association username/password is weak since anyone with knowledge of this credentials can access the account even if they aren't the owner of the account. That's why two factor authentication is essential. In my implementation, after a user types the username and password and press login button, they are redirected to another page with a QR Code.



Part 1.0 - 2FA Form	
Google Authenticator Token:	Enter Token
<div>Login</div>	

In this page, the user needs to scan the QR Code with an application that generates 2-step verification codes in their phone (for example Google Authenticator: <https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2&hl=en>).

After scanning the user receives a 6-digit code that renews every 60 seconds:



To complete the login, the user just needs to type the 6-digit code in the web application form. If the code is correct, the user will be redirected to the part2 of the application. This form is protected against all attacks mentioned so far.

## II-I SQL Injection Prevention & Attacks

As said before, the secure forms were protected against SQL injections using the default protections from the node-postgres library (parameterized queries). The cases where I couldn't use parameterized queries were the cases the input was integers. In order to protect it I converted the input to numbers using parseInt or parseFloat. This way, malicious inputs would be destroyed.

Although, as you know, there are some vulnerable forms in the application. In this ones, malicious inputs can result in SQL injections. "" OR 1=1-" and "" DROP table users-" are two of the many examples.

## II-J Brute Force Protection & Attacks

Lot of web apps are vulnerable to brute force attacks. In fact, the part1\_vulnerable form in my application is vulnerable to brute force attacks, i.e., this form hasn't any protection that limits a user from making requests.

Below there is a simple script that tests a list of more than five hundred common passwords (only four showed in the image but the file "bruteforce\_credentials.js" in the repository contains more than 500):

```

1 const http = require("http");
2
3 const username = "jose"
4
5 const passwords = [
6   "123456",
7   "password",
8   "12345678",
9   "1234",
10  ...
11 ];
12
13 passwords.forEach(pass => {
14   http
15     .get(
16       `http://localhost:3000/part1_vulnerable?v_username=${username}
17       &v_password=${pass}&v_remember=on`,
18     )
19     .on("data", chunk => {
20       data += chunk;
21     })
22     .on("end", () => {
23       if (data !== "Invalid Username or Password!") {
24         console.log("password is: ", pass);
25         process.exit();
26       }
27     })
28   );
29 }
30
31 // A chunk of data has been recieved.
32 // The whole response has been received. Print out the result.
33
34 resp.on("end", () => {
35   if (data !== "Invalid Username or Password!") {
36     console.log("password is: ", pass);
37     process.exit();
38   }
39 });
40
41 .on("error", err => {
42   console.log("Error: " + err.message);
43 });
44 });
45

```

For each password a get request is sent to check if the password, for example, for the user jose is the correct one or not. Since in this form there is no brute force protection, this small script will make over five hundred requests and test all the passwords.

If the password doesn't match the script passes to the next password in the list, if it matches the password is logged and the script exits. Having the password, we can log in the web application in the vulnerable form like a legit user.

As you can see, this is a big problem. To solve it I used a library called express-brute (<https://www.npmjs.com/package/express-brute>) that limits the requests that an user can make to the server. If an user is making too much requests their access is forbidden:

```

"error": { "text": "Too many requests in this time frame.",
"nextValidRequestDate": "2019-12-26T17:27:51.194Z" }

```

## II-K XSS Protection & Attacks

The solution to solve this type of attack was the easiest one. In fact, the files provided by the teachers already protected the application from XSS. In the EJS files, when using `<%= variable_from_server %>` escapes the information that comes from the server and makes it impossible to perform XSS.

In order to make it more interesting (and vulnerable) I changed from `<%= variable_from_server %>` to `<%- variable_from_server %>`. I already explained in section of Part2 my implementation to protect the application against XSS in the safe form.

Although, the vulnerable form can be attacked. If we insert `<h1>title </h1>` we can see rightaway that it is vulnerable.

Output Box
Vulnerable: Hi! I wrote this message using Vulnerable Form.
Correct: OMG! This form is so correct!!!
Vulnerable: Oh really?
kow89329@eveav.com vulnerable: okdwakdaw
kow89329@eveav.com correct: dwadwadwa
kow89329@eveav.com vulnerable:
<b>dwadwa</b>
josef vulnerable: ola
josef vulnerable:
<b>i was exploited</b>
josef correct: <h1>i wasn't</h1>

Even though, no critical information is saved on cookies without httpOnly flag activated or in localStorage/sessionStorage. So, no critical information from the user could be stolen using XSS.

## II-L CSRF Protection & Attacks

Again, in part2 I already explained how I protected my application against CSRF attacks with csrf library. Below we have two forms I created to perform CSRF attacks (the first one is vulnerable and the second is not):

```

1 <html>
2 <body>
3 <form
4   method="GET"
5   action="https://ddss-assignment2.herokuapp.com/part2_vulnerable?v_text=i_am_vulnerable"
6 >
7   <input type="hidden" name="v_text" value="i_am_vulnerable" />
8   <input type="submit" value="Submit" />
9 </form>
10 </body>
11 </html>

```

```

1 <html>
2 <body>
3 <form
4   method="POST"
5   action="https://ddss-assignment2.herokuapp.com/part2_correct"
6 >
7   <input type="hidden" name="c_text" value="im not vulnerable" />
8   <input type="submit" value="Submit" />
9 </form>
10 </body>
11 </html>

```

Note that the first form will aim at the vulnerable form in /part2 and the second one will aim the safe form in the same page of the application. Have in mind also that the vulnerable form isn't protected against this type of attacks because we don't have a csrf token associated. Therefore, any forms with no csrf token will be valid. On the other hand, in the safe form, if the form hasn't a valid csrf token associated it will result in an error.

If an authenticated user gets tricked into submitting the forms presented above two cases will happen:

- 1) the attack will be successful in the first case and the vulnerable message will be inserted:



```
josef_vulnerable:
i was exploited
josef_correct: <h1>i wasn't</h1>
josef_vulnerable: i am vulnerable
```

- otherwise, the attack will not be successful and the user will be redirected to a page with "INVALID CSRF TOKEN" because the attack is being executed against a secure form with CSRF protection.

## II-M Sensitive Credentials & dotenv

When coding web applications, storing sensitive credentials in the source code is considered a very bad practice. In all the application, whenever a sensible variable was needed (JWT secrets or database credentials for example) `process.env.VARIABLE_NAME` is used. This way, all the variables can be stored in a safe place separate from the rest of the code. In order to accomplish this, I used the library `dotenv` (<https://github.com/motdotla/dotenv>) when developing the app and then in production I used their heroku config vars.

## II-N Deploy & HTTPS

In order to deploy the application and have HTTPS for my application I used heroku (<https://www.heroku.com/>). The application is live in the url <http://ddss-assignment2.herokuapp.com/>. The source code of the application is in <https://github.com/jose-donato/ddss2019> and to run it locally you just need to execute the following command: "docker-compose up --build" in the root folder.

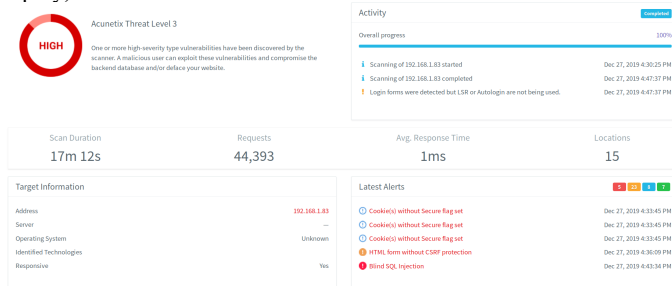
## III Part2 - Testing and Static Analysis

### III-A Web Vulnerability Scanner - Acunetix

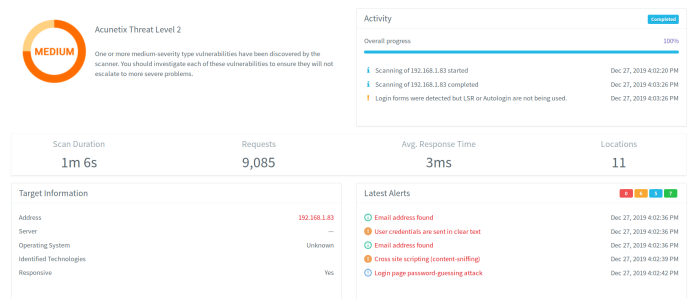
To perform blackbox tests, I used a web vulnerability scanner from acunetix (<https://www.acunetix.com/vulnerability-scanner/web-application-security/>). To get several results I made three different tests:

- All forms and no brute-force protections
- Only safe forms and some brute-force protections
- Only safe forms and a lot of brute-force protections

As expected, in the first one the results gave lot of high, medium and low vulnerabilities since the project included several vulnerable forms. The vulnerabilities include, in addition to others, CSRF, blind SQL injections and XSS. Other minor vulnerabilities were in the report such as sending credentials in clear text (because the vulnerable login form is sent in parameters with GET request) or warning about not having HTTPS (this is solved with the heroku deploy).

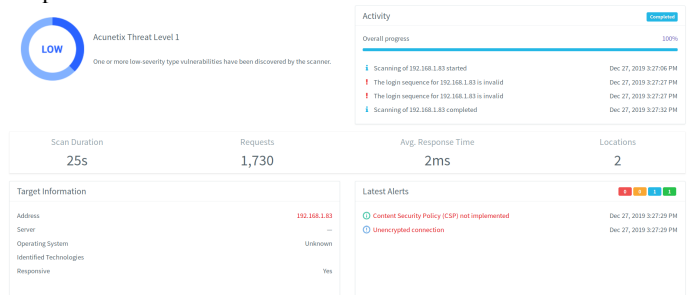


In order to get a better chance on analysing the results from this tool, I removed the vulnerable forms from the application. The results were the following:



I got zero high, 6 medium and 5 low vulnerabilities. Almost all the medium vulnerabilities were "Cross site scripting (content-sniffing)". I tried to find why this vulnerability was reported although without success since the free version of acunetix doesn't show it and I wasn't able to track it when analysing the code. The low vulnerabilities were regarding to HTTPS and "Email not found". The last one I think it is related to some dependency that I used because I didn't find any email on all my code. Also, the "Login Page password-guessing attack" shows in the report but I think it is a false positive because of the protections this form has (brute force prevention and password validation in sign up).

With the goal of restricting even more the power of this tool, I put brute-force prevention on all routes of the application. The results are presented below:



As you can see, the tool was only able to perform 1730 requests and only found two very low vulnerabilities. The first one was Content Security Policy (CSP) not implemented which was later solved by setting a header in the responses ("res.setHeader("x-frame-options", "deny");"). The other vulnerability was regarding HTTPS which was solved with the heroku deploy.

If not limited by some brute-force library, this tool is extremely powerful and detects a lot of vulnerabilities making it a viable testing tool when developing safe web applications.



### III-B Vulnerability Finder - Snyk

Since the project was done in nodejs environment I used a lot of open source libraries available in npm (<http://npmjs.com/>). It is always important to check if any of the libraries has known vulnerabilities. To test this, I used snyk (<https://snyk.io/>). Snyk is a command line tool to find vulnerabilities in open source dependencies in a certain project. To use the tool you just need to install it and then run "snyk monitor".

The results are the following:

## ddss-webapp

Snapshot taken by cli 18 seconds ago.

Vulnerabilities	1 via 1 paths	Dependencies	271
Taken by	CLI/CLI	Hostname	josedonato-XPS-13-9360
Imported by	 zmcdonato@gmail.com zmcdonato@gmail.com	Project owner	 Add a project owner

Analysing this, only one known vulnerability (express-brute library) was found from 271 dependencies in the project. This is the Common Vulnerability Scoring System of the vulnerability:

CVSS SCORE

5.6

MEDIUM SEVERITY

ATTACK VECTOR

Network

ATTACK COMPLEXITY

High

PRIVILEGES REQUIRED

None

USER INTERACTION

None

SCOPE

Unchanged

CONFIDENTIALITY

Low

INTEGRITY

Low

AVAILABILITY

Low

Since this is a vulnerability with no known exploit, high complexity and low impact on Confidentiality, Availability and Integrity I chose to keep this useful library in the application although, in a bigger or more complex project I would remove this library right away.

In my opinion, with close to zero effort to setup it (only two commands: one to install and another to run it) this tool provides crucial information and its use should be required when developing web applications.

### III-C Static Analysis - SonarQube

To do the static analysis of the code, I used the SonarQube (<https://www.sonarqube.org/>) community edition with the typescript plugin. This tool analysed over 800 lines in my project:

## About This Project

No tags

XS 825

Lines of Code

JavaScript 825

## Project Activity



First I got some (around 10) code smells (code that is confused and difficult to maintain), one bug (coding error that will break the code and needs to be fixed immediately) and one vulnerability (because one database credential was being stored in plaintext instead of in some safe place). An example of a code smell was this:

```
const resultInsert = await client.query(queryInsert, [
```

Remove the declaration of the unused 'resultInsert' variable.

See Rule

14 days ago L26

 Code Smell

 Minor

 Open

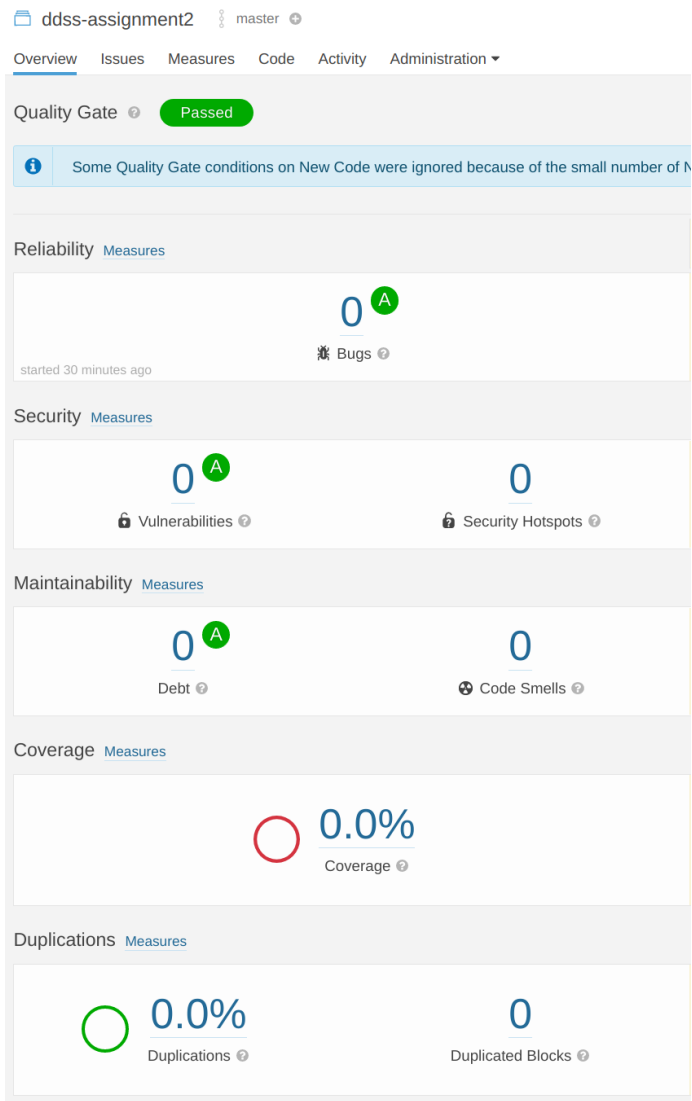
 Not assigned

 5min effort

 Comment

 unused

As you can see, the fixes were easy and the final report was the following:



Achieved zero bugs, vulnerabilities and code smells. Although, I noticed that zero of my code was test covered because I didn't perform any type of tests. To solve this issue, I used a framework called cypress to perform end to end (e2e) testing. I will talk more about this in next section.

### III-D End to End Testing - Cypress

Cypress (<https://www.cypress.io/>) is a JavaScript End to End Testing framework. And although it is a framework made to use with javascript, they support typescript out of the box.

The tests I did are stored inside the folder `nodejs/cypress/integration` and are divided into four different files:

- part0.ts: cover the sign up form in the part0

```
Part0
  ✓ Sign up form loads
  ✓ Sign up with same username gives error
```

- part1.ts: cover the vulnerable login form in part1

```
Part1 Vulnerable Form
  ✓ Sign in forms load
  ✓ Sign in with correct credentials redirects to part2 and receives jwt token
  ✓ Sign in with wrong credentials gives error
```

- part2.ts: cover the vulnerable chat form in part2

```
Part2 Vulnerable Form
  ✓ Message forms load
  ✓ Submit message is included in Output Box
```

- part3.ts: cover the vulnerable search form in part3

```
Part3 Vulnerable Form
  ✓ Book search forms load
  ✓ search existing book gives book in output
  ✓ search non existing book gives Zero books found
```

Because the lack of time I wasn't able to test all the code and only some parts of the vulnerable forms. But this showed the power of this testing library and how it can be useful to test all different parts of the application. If changes are made to test covered code, we can just run the tests again and see if it broke something or everything is fine.

## IV Conclusion

With this assignment I was able to develop a fully functional application having in mind the best security practices. I also had the opportunity to position myself in the attacker perspective and try to exploit my own application (and then protect it).

I learned about different useful technologies, frameworks and libraries that will be useful in the future when developing web applications.

## References

- [1] Zappedia. How Many People Use the Internet? <https://zappedia.com/global-internet-access/>.
- [2] OWASP. OWASP Cheat Sheet Series. [https://cheatsheetseries.owasp.org/cheatsheets/Authentication\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html).
- [3] password hashing. Password Hashing Competition. <https://password-hashing.net/>.
- [4] Snyk. bcrypt vulnerabilities. <https://snyk.io/vuln/rubygems:bcrypt>.
- [5] Clément Delafargue. Why Auto Increment Is A Terrible Idea. <https://www.clever-cloud.com/blog/engineering/2015/05/20/why-auto-increment-is-a-terrible-idea/>.
- [6] Marcus Pinto Dafydd Stuttard. The Web Application Hacker's Handbook. <https://portswigger.net/web-security/web-application-hackers-handbook>.