CVE DISCOVERY AND REPORTING

Análise e Exploração de Vulnerabilidades

Ricardo Covelo - 102668
Telmo Sauce - 104428
José Alexandre - 118373
Rafael Oliveira - 117240

3/11/2023

Contents

1 Introduction **3**

2 Scope **3**

2.1 The Usage Environment and Area of the Assessment 3

2.2 Chosen programming languages 3

2.3 Area Of Assessment 3

2.4 ASVSS 4

3 Vulnerability exploration **6**

3.1 Scope 6

3.2 Analysis of AVSS 6

3.3 Brute Force Exploration 16

1 Introduction

To introduce this AEV project, we will focus on understanding and reporting the phases involved in finding vulnerabilities, and ultimately issuing a Common Vulnerabilities and Exposures(CVE).Throughout this project, we will gain a comprehensive understanding of the different phases involved in the vulnerability identification process. This will include learning how to recognize potential vulnerabilities, assess their severity and build an effective report that details the information needed for responsible disclosure.

2 Scope

2.1 The Usage Environment and Area of the Assessment

In this work, we will evaluate a plethora of GitHub projects, we will deploy the websites in our environments, and find vulnerabilities on those projects. Methods for attacking a website can include SQL injection, cross-site scripting, and brute force attacks.

2.2 Chosen programming languages

Given the overall ability of the group with the most used technologies, we chose to work with systems, and eventually solutions to the possible vulnerabilities that use the most common programming languages for Web Development. For the back end, we will use JavaScript, TypeScript, Python, and HTML (Used in Angular, and React).

2.3 Area Of Assessment

We'll work with many areas of assessment such as Access Control (or input validation), we will also look of bad programming and bad coding practices.

Access control is an important aspect of software security that prevents unauthorized access to sensitive data. Input validation is another crucial aspect of software development that ensures the integrity of data and prevents attacks like SQL injection. Bad programming practices such as using outdated programming languages and relying on untested code can undermine the software's functionality and security. Similarly, bad coding practices such as poor documentation and unclear variable names can make the program difficult to maintain and debug. Therefore, it is essential to perform thorough assessments of these areas to ensure a robust and secure software product.

2.4 ASVSS

Injection And Cross-site Scripting [1]

5.1 - Input Validation

Failure to validate input coming from the client or the environment before directly using it without any output encoding.

- 5.1.1 - Verify that the application has defenses against HTTP parameter pollution attacks.
- 5.1.3 - Verify that all input is validated using positive validation.
- 5.1.5 - Verify that URL redirects only allow destinations that appear on an allow list, or show a warning when redirecting to potentially untrusted content.

5.2 - Sanitization and Sandboxing

- 5.2.2 - Verify that unstructured data is sanitized to enforce safety measures such as allowed characters and length.
- 5.2.4 - Verify that the application avoids the use of dynamic code execution features. Where there is no alternative, any user input being included must be sanitized or sandboxed.

5.3 - Output Encoding and Injection Prevention

Output encoding is crucial for application security and should be located near the interpreter in use. While not typically saved, encoding is used to ensure safe output in the desired context. Neglecting proper encoding can result in a significant security flaw.

- 5.3.4 - Verify that data selection or database queries use parameterized queries, ORMs, entity frameworks, or are otherwise protected from database injection attacks.
- 5.3.5 - Verify that where parameterized or safer mechanisms are not present, context-specific output encoding is used to protect against injection attacks, such as the use of SQL escaping to protect against SQL injection.
- 5.3.8 - Verify that the application protects against OS command injection and that operating system calls use parameterized OS queries or use contextual command line output encoding.
- 5.3.9 - Verify that the application protects against Local File Inclusion or Remote File Inclusion attacks.

6.1 - Data Classification

The data processed, stored, or transmitted by an application is the key asset. We need to always conduct a privacy impact assessment to accurately classify the protection requirements of stored data.

- 6.1.1 - Verify that regulated private data is stored encrypted while at rest, such as Personally Identifiable Information.

6.2 - Algorithms

- 6.2.3 - Verify that encryption initialization vector, cipher configuration, and block modes are configured securely using the latest advice.

12.3 - File and command Execution

Failure to prevent commands from a user being executed in the Server operating System can result in problems like data theft, and Denial of service.

- 12.3.3 - Verify that user-submitted filename metadata is validated or ignored to prevent the disclosure or execution of remote files via Remote File Inclusion or Server-side Request Forgery attacks.
- 12.3.5 - Verify that untrusted file metadata is not used directly with system API or libraries, to protect against OS command injection.
- 12.3.6 - Verify that the application does not include and execute functionality from untrusted sources, such as unverified content distribution networks, JavaScript libraries, node npm libraries, or server-side DLLs.

Brute Force [1]

2.1 - Password Security

Passwords, or "Memorized Secrets" by NIST 800-63, encompass a range of security measures including PINs, unlock patterns, image selection, and passphrases. These are typically classified as "something you know" and used as single-factor authenticators. It is essential for individuals to use strong and unique passwords, frequently change them, and consider combining them with additional security measures for enhanced protection.

- 2.1.1 - Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined).
- 2.1.3 - Verify that password truncation is not performed. However, consecutive multiple spaces may be replaced by a single space.
- 2.1.5 - Verify users can change their password.
- 2.1.6 - Verify that password change functionality requires the user's current and new password.
- 2.1.7 - The application checks if passwords are compromised during registration, login, and change using either a local list of common passwords or an external API. If an API is used, it protects the password using a zero-knowledge proof method. If compromised, it requires the user to set a new secure password.
- 2.1.8 - Verify that a password strength meter is provided to help users set a stronger password.

2.2 - General Authentication Security

Failure to ensure general future-proof safety mechanisms.

- 2.2.1 - The system confirms the efficacy of anti-automation measures against credential testing, brute force, and account lockout attacks. These include blocking common breached passwords, using soft lockouts, rate limiting, CAPTCHA, delayed attempts, IP restrictions, and risk-based controls. Additionally, it restricts failed attempts to 100 per hour for a single account.

13.1 - Generic Web Service Security

Ensure that a verified application that uses trusted service layer APIs.

- 13.1.3 - Verify API URLs do not expose sensitive information, such as the API key, session tokens etc.

14.2 - Dependency

Failure to keep up to date with outdated or insecure dependencies is the root cause of the largest and most expensive attacks to date.

- 14.2.1 - Verify that all components are up to date, preferably using a dependency checker during build or compile time.

14.5 - HTTP Request Header Validation

- 14.5.3 - Verify that the Cross-Origin Resource Sharing (CORS) Access-Control-Allow-Origin header uses a strict allow list of trusted domains and subdomains to match against and does not support the "null" origin.

3 Vulnerability exploration

3.1 Scope

During our thorough investigation of various websites for vulnerabilities on GitHub, we discovered an Airbnb Clone project(github.com/AMS006/AirBnbClone/) that utilizes React and MongoDB as its key technologies.

3.2 Analysis of AVSS

Injection And Cross-site Scripting

- 5.1.1 - Upon inspection, no vulnerabilities or weaknesses related to HTTP parameter pollution attacks were detected within the application's codebase. The application seems to have robust mechanisms in place to mitigate and prevent such attacks.

- 5.1.3 - The Website does not clearly define the expected patterns for each type of input (e.g., usernames, passwords, email addresses, etc.).

- 5.1.5 - Currently, the website is unable to show a warning when redirecting to potentially untrusted content and does not have an allow list with safe destinations.
- 5.2.2 - After we explore the website, we could see that there are restrictions for special characters and length but we can insert negative values, both in the price and in the guests.

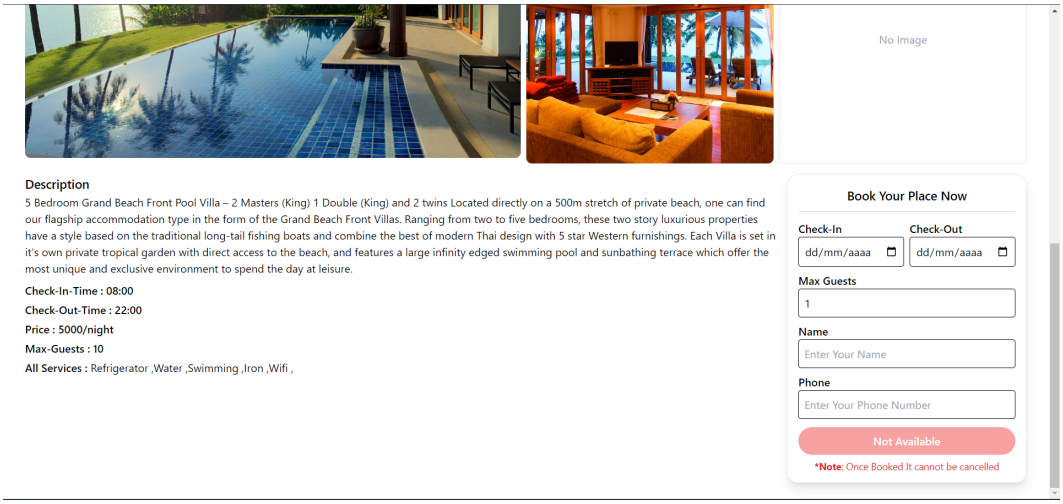

Figure 1: Booking a place

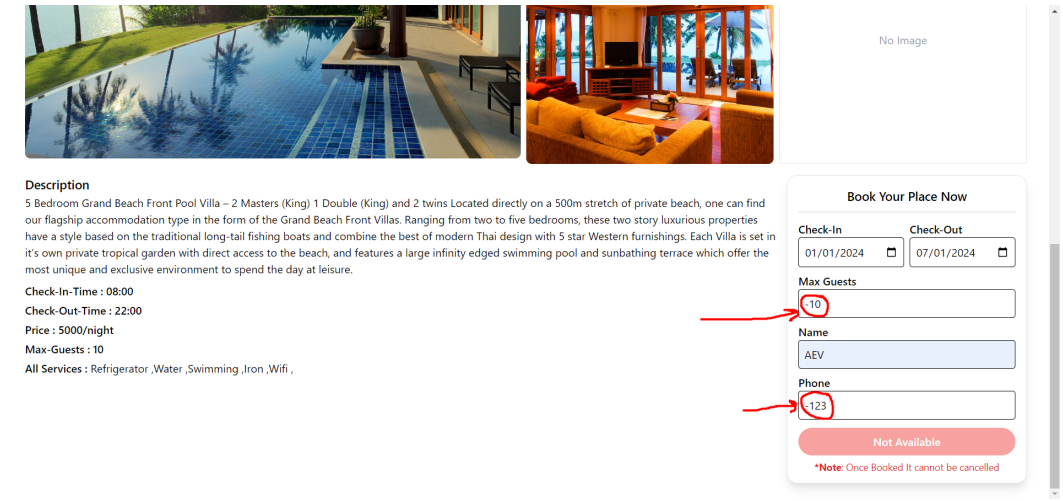

Figure 2: Entering negative values

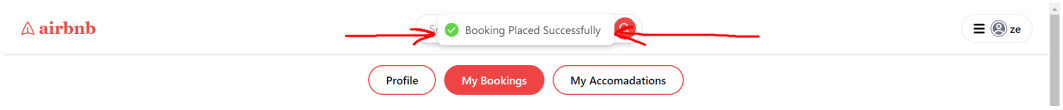

Figure 3: Booking confirmation

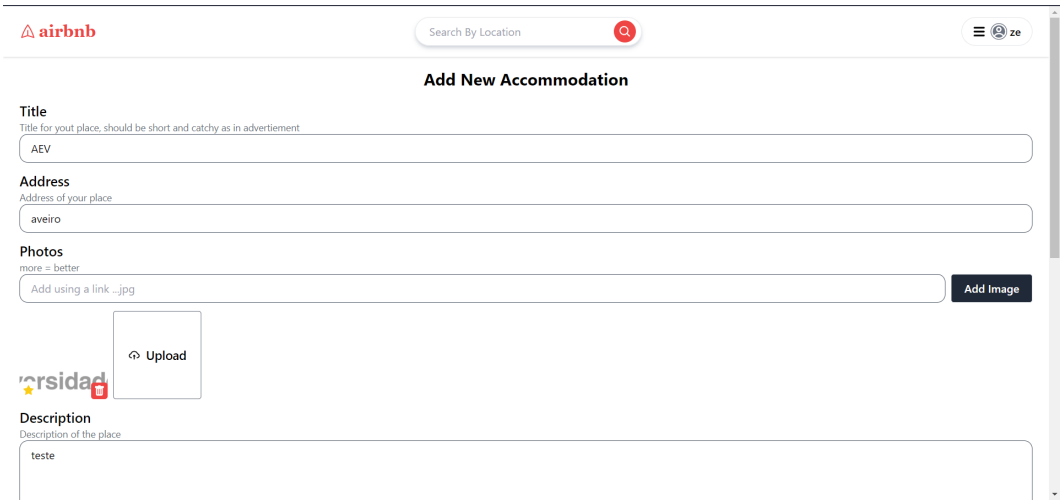

Figure 4: Creating an offer

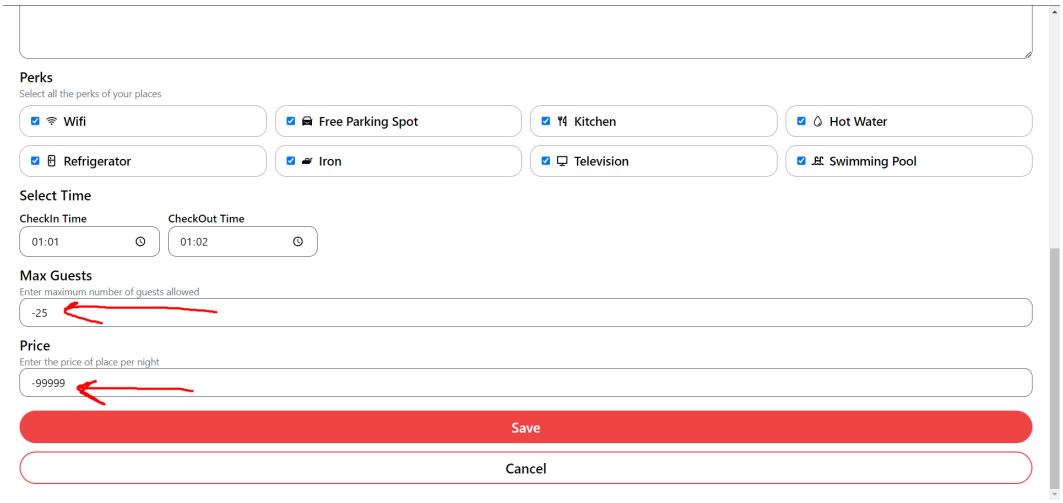

Figure 5: Entering negative values

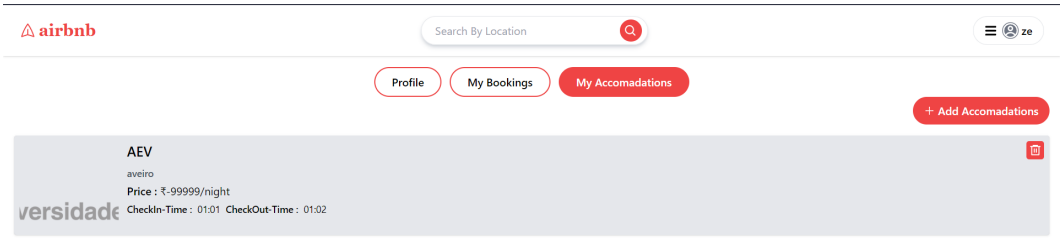

Figure 6: Offer confirmation

- 5.2.4 - The code does not use dynamic code execution features such as eval() or exec(). React itself doesn't directly support server-side dynamic code execution features.

- 5.3.4 - After the review, it's confirmed that the implemented code utilizes secure methodologies, such as the use of Mongoose ORM in conjunction with bcrypt for password hashing, ensuring protection against potential database injection attacks. Parameterized queries or direct protection mechanisms from such attacks were not found to be explicitly implemented, as the code relies on these secure practices provided by the utilized libraries

```
    password: bcrypt.hashSync(password,bcryptSalt)});
const mongoose = require('mongoose')
const userShema = new mongoose.Schema({
    name:{
        type:String,
        required:true
    },
    email:{
        type:String,
        required:true,
        unique:true,
    },
    password:{
        type:String,
        required:true
    }
})

module.exports = mongoose.model('users',userShema)
```

Figure 7: Preventing injection attacks

- 5.3.5 - After a comprehensive review, it's confirmed that the code implements parameterized or secure mechanisms where necessary. For example, in the Account Page:
  - React's {} curly braces are used for rendering dynamic content in JSX. This helps prevent XSS attacks by automatically escaping the content.

```
<p className='text-center text-xl mt-4 font-semibold'>Name: <span>{user.name}</span></p>
<p className='text-center text-xl font-semibold'>Email:  <span>{user.email}</span></p>
```

Figure 8: Preventing XSS (Cross-Site Scripting)

In the above code, user data (user.name and user.email) is interpolated within JSX using curly braces, which ensures that the data is safely rendered.

  - React's state and props are used for managing and passing data instead of directly manipulating the DOM. This helps in maintaining a declarative and secure approach to rendering UI.

```
<Link to={'/account/profile'} className={createClass('profile')}>Profile</Link>
```

Figure 9: Avoiding direct manipulation of the DOM

The Link component from react-router-dom is used for navigation, and the createClass function is used to dynamically set the class of the link based on the current subpage.

– The handleLogout function uses the preventDefault method to prevent the default form submission behavior. This is a good practice to ensure that the logout action is triggered only when intended, preventing unwanted navigation.

```
const handleLogout = (e) => {
    e.preventDefault();
    dispatch(logoutUser())
    navigate('/')
}
```

Figure 10: Handling user input securely

– The getUserPlaces action is dispatched within the useEffect hook. This is a good practice for fetching user data after the component has mounted, and it helps in managing the component lifecycle without introducing security risks.

```
useEffect(() => {
    dispatch(getUserPlaces())
}, [dispatch])
```

Figure 11: Handling user input securely

- 5.3.8 - The website allows file uploads, ensuring that uploaded files are properly validated, and never execute commands based on file names or contents. In this case, we only can insert images. Any other file wont load and the offer wont validate.

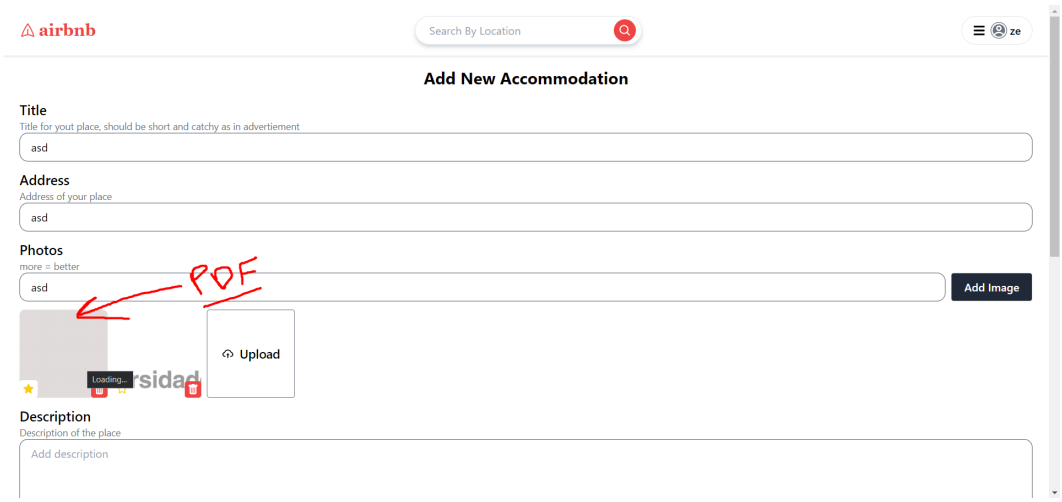

Figure 12: PDF wont load but the image does

- 5.3.9 - This website uses security headers, such as "X-Content-Type-Options" to enforce that browsers interpret files with the expected content type.

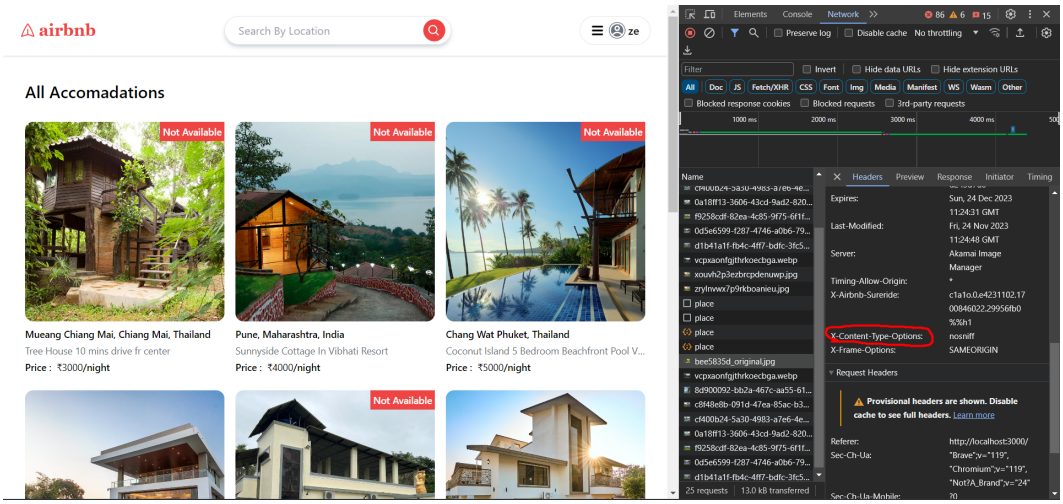

Figure 13: X-Content-Type-Options

- 6.1.1 - Upon review, no instances were found where untrusted file metadata was directly utilized with system APIs or libraries, mitigating the risk of potential OS command injection vulnerabilities.

- 6.2.3 - After thorough examination, it's confirmed that the code doesn't explicitly handle or expose encryption initialization vector, cipher configuration, or block modes. The utilized libraries, such as 'bcrypt' for password hashing and 'jsonwebtoken' for token generation, abstract away these low-level encryption settings, ensuring secure defaults without direct developer intervention.

Figure 14: Secure Encryption Configuration

- 12.3.3 - There is no evidence whatsoever that user-submitted filename metadata is not validated or ignored. So, the website is ready to prevent the disclosure or execution of remote files via Remote File Inclusion or Server-side Request Forgery attacks.
- 12.3.5 - This website only allows valid characters, rejecting any input that may be interpreted as a command.
- 12.3.6 - After reviewing the code and use tools like snyk, we could see that the Website does not include and execute functionality from untrusted sources. But we can do regular security audits to identify and address potential vulnerabilities.

Brute Force

- 2.1.1 - The Website currently doesn't make any restriction on the length of the passwords they can have only one character, as shown on the figure 15.

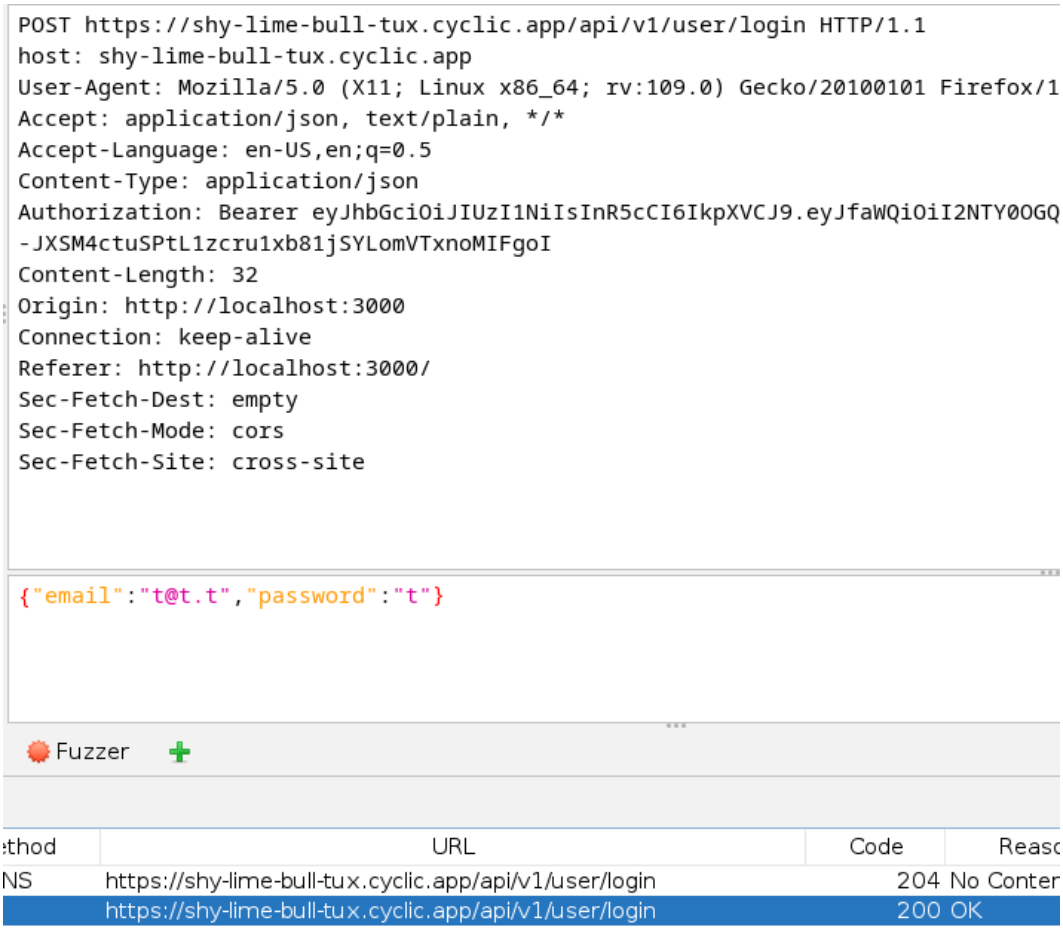

Figure 15: Valid Short Password

- 2.1.3 - The truncation of passwords is not performed, however the passwords with multiple spaces(figure 16) are not replaced by one, as we can see on the figure 17.

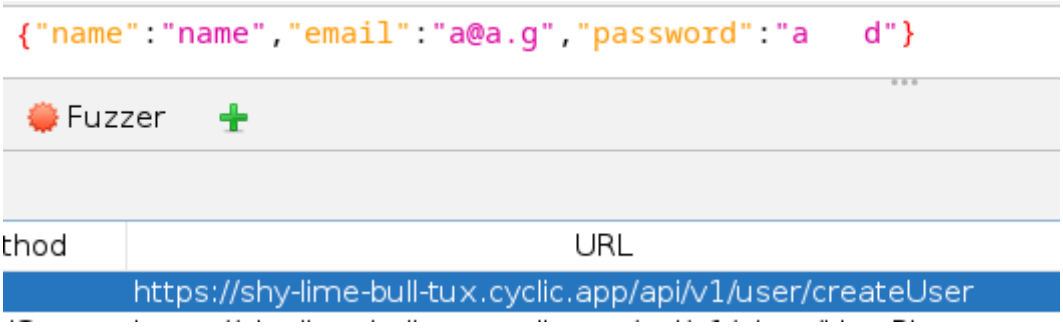

Figure 16: Multiple spaces register

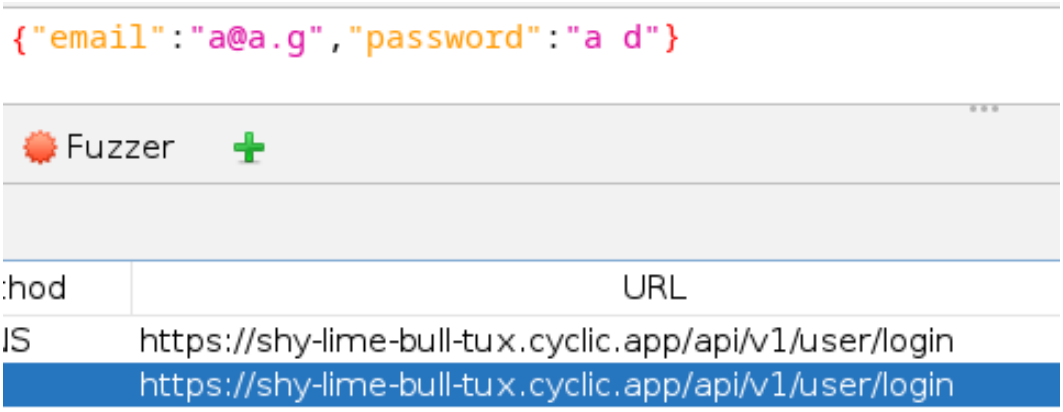

Figure 17: Single Space login

- 2.1.5 and 2.1.6 - There is no place where a user can change it's password.
- 2.1.7 - The application doesn't do any verification on the security of the password used by the user, neither with a local list or a API.
- 2.1.8 - There is not guidance to encourage users to use a strong password on their accounts.
- 2.2.1 - The System only has in place a rate limit, however this is really small, which doesn't affect much the brute force attacks.
- 13.1.3 - We could see that the API Request ID is being exposed in some items from the website, like in the image below.

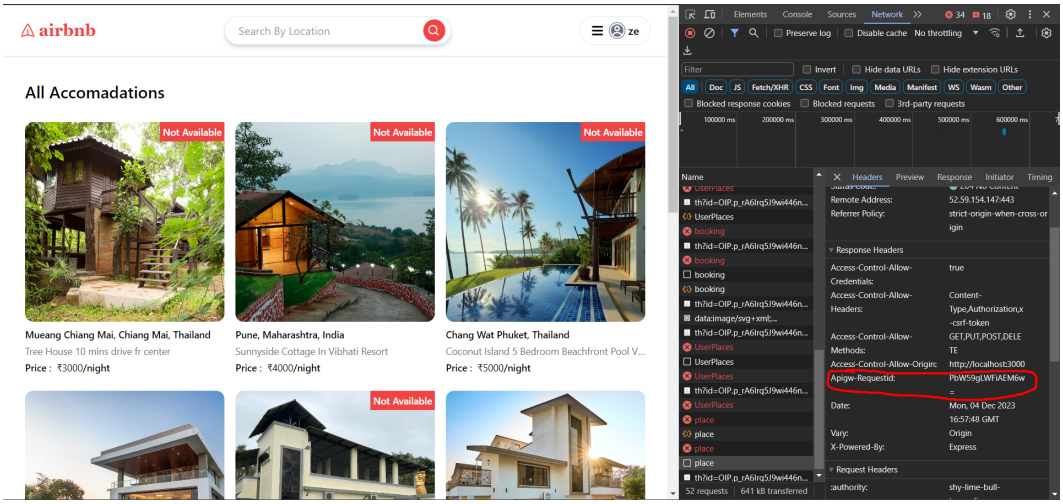

Figure 18: API Request ID exposed

- 14.2.1 - The System as outdated dependencies which have vulnerabilities and some are already fixed in newer versions as shown on the figures 19 and 20.

```
┌──(sauce㉿sqlsrv-cl2)-[~/AirBnbClone/client]
└─$ npm outdated

Package                      Current  Wanted  Latest  Location                                  Depended by
@testing-library/jest-dom      5.17.0  5.17.0   6.1.4  node_modules/@testing-library/jest-dom    client
@testing-library/react        13.4.0  13.4.0  14.1.2  node_modules/@testing-library/react       client
@testing-library/user-event   13.5.0  13.5.0  14.5.1  node_modules/@testing-library/user-event  client
date-fns                       1.30.1  1.30.1  2.30.0  node_modules/date-fns                     client
web-vitals                      2.1.4   2.1.4   3.5.0  node_modules/web-vitals                   client

┌──(sauce㉿sqlsrv-cl2)-[~/AirBnbClone/client]
└─$ npm audit

# npm audit report

nth-check  <2.0.1
Severity: high
Inefficient Regular Expression Complexity in nth-check - https://github.com/advisories/GHSA-rp65-9cf3-cjxr
fix available via `npm audit fix --force`
Will install react-scripts@3.0.1, which is a breaking change
node_modules/svgo/node_modules/nth-check
  css-select  ≤3.1.0
  Depends on vulnerable versions of nth-check
  node_modules/svgo/node_modules/css-select
    svgo  1.0.0 - 1.3.2
    Depends on vulnerable versions of css-select
    node_modules/svgo
      @svgr/plugin-svgo  ≤5.5.0
      Depends on vulnerable versions of svgo
      node_modules/@svgr/plugin-svgo
        @svgr/webpack  4.0.0 - 5.5.0
        Depends on vulnerable versions of @svgr/plugin-svgo
        node_modules/@svgr/webpack
          react-scripts  ≥2.1.4
          Depends on vulnerable versions of @svgr/webpack
          Depends on vulnerable versions of resolve-url-loader
          node_modules/react-scripts

postcss  <8.4.31
Severity: moderate
PostCSS line return parsing error - https://github.com/advisories/GHSA-7fh5-64p2-3v2j
fix available via `npm audit fix --force`
Will install react-scripts@3.0.1, which is a breaking change
node_modules/resolve-url-loader/node_modules/postcss
  resolve-url-loader  0.0.1-experiment-postcss || 3.0.0-alpha.1 - 4.0.0
  Depends on vulnerable versions of postcss
  node_modules/resolve-url-loader

8 vulnerabilities (2 moderate, 6 high)
```

Figure 19: Client dependencies outdated

```
┌──(sauce㉿sqlsrv-cl2)-[~/AirBnbClone/server]
└─$ npm outdated

Package   Current   Wanted   Latest  Location              Depended by
aws-sdk   2.1502.0  2.1503.0 2.1503.0 node_modules/aws-sdk  server
mongoose    7.6.5     7.6.5    8.0.1  node_modules/mongoose server
nodemon    2.0.22    2.0.22    3.0.1  node_modules/nodemon  server

┌──(sauce㉿sqlsrv-cl2)-[~/AirBnbClone/server]
└─$ npm audit

# npm audit report

semver  7.0.0 - 7.5.1
Severity: moderate
semver vulnerable to Regular Expression Denial of Service - https://github.com/advisories/GHSA-c2qf-rxjj-qqgw
fix available via `npm audit fix --force`
Will install nodemon@3.0.1, which is a breaking change
node_modules/simple-update-notifier/node_modules/semver
  simple-update-notifier  1.0.7 - 1.1.0
  Depends on vulnerable versions of semver
  node_modules/simple-update-notifier
    nodemon  2.0.19 - 2.0.22
    Depends on vulnerable versions of simple-update-notifier
    node_modules/nodemon

3 moderate severity vulnerabilities
```

Figure 20: Server dependencies outdated

3.3 Brute Force Exploration

Explanation

A brute force attack is made possible by the lack of sufficient restrictions and policies on the website to prevent users from creating weak passwords, CWE-521 , as discussed in section 3.2. Additionally, there are no limitations on the number of login attempts or penalties for entering incorrect passwords multiple times, CWE-307.

The only deterrent against brute force attacks is a "rate limit" that restricts the number of login requests from a particular entity, such as a user, IP address, or application, within a specific timeframe. This limitation became apparent during our attempt to perform a brute force attack using ZAP.

To address this vulnerability, we have developed a Python script that retries any password that receives a status code of 500 or 429. This enables us to continue testing passwords that haven't been blocked by the rate limit.

The vulnerability mentioned is highly significant for the system, considering its CVSS score and exploit severity of 8.2, as evaluated by the NVD calculator. To enhance the website's security and mitigate brute force attacks, it is imperative to enforce stronger password policies and implement restrictions and penalties for incorrect login attempts.

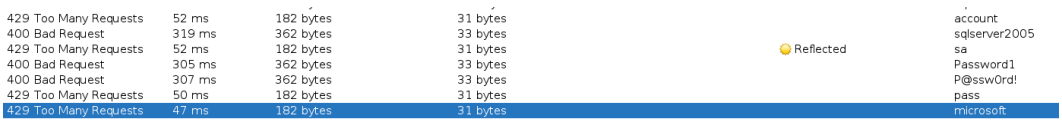

Figure 21: Brute Force Attack on ZAP

Brute Force Script

The script is developed in Python and utilizes the requests library. The main section of the code focuses on modifying the payload to change the password and sends a request to the server. If the server responds with a status code of 429 or 500, it indicates that the script has been stopped due to rate limiting. In such cases, the password is stored in a list for future testing.

However, if the payload returns a status code of 200 at any point, the script is terminated, and the password is displayed. This indicates that the password has been successfully found.

With this approach, the script automates the process of systematically testing passwords until a successful match is found, taking into consideration rate limiting which can occur with this approach passwords that returned 429 or 500 can be retested and are not discarded.

```python
# Initialize a list to store the passwords that resulted in a 500 or 429 status code
passwords_to_store = []

with open(wordlist_path, 'r') as file:
    passwords = file.readlines()

for password in passwords:
    password = password.strip()  # remove newline characters
    data = {"email": user, "password": password}

    response = requests.post(url, headers=headers, data=json.dumps(data))

    # If the status code is 500 or 429, store the password
    if response.status_code in [500, 429]:
        passwords_to_store.append(password)

    # If the status code is 200, store the password
    elif response.status_code == 200:
        print(f"User: {data['email']}\nPassword: {password}")
        return
```

Figure 22: First for loop of the payload

```python
for password in passwords_to_store:
    data["password"] = password

    response = requests.post(url, headers=headers, data=json.dumps(data))

    # If the status code is 200 for the retest, store the password
    if response.status_code == 200:
        print(f"User: {data['email']}\nPassword: {password}")
        return
```

Figure 23: Retest passwords with code 429 and 500

References

[1] OWASP, "Application security verification standard 4.0.3," 2021.