



# **UNIVERSIDADE DE AVEIRO**

*DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA*

**42573 - Segurança Informática e Nas Organizações**

João Ferreira 103625

Arthur 102667

**1st-project-group\_37**

2023

# Contents

<b>1</b>	<b>Shop Introduction</b>	<b>5</b>
<b>2</b>	<b>CWEs</b>	<b>6</b>
2.1	CWE-79 . . . . .	6
2.2	CWE-89 . . . . .	6
2.3	CWE-209 . . . . .	6
2.4	CWE-256 . . . . .	6
2.5	CWE-284 . . . . .	6
2.6	CWE-613 . . . . .	6
2.7	CWE-620 . . . . .	6
2.8	CWE-756 . . . . .	7
<b>3</b>	<b>Vulnerability Exploit</b>	<b>8</b>
3.1	CWE-79 . . . . .	8
3.1.1	Where does this occur? . . . . .	8
3.1.2	Connect section . . . . .	8
3.1.3	What does the attacker gain? . . . . .	8
3.2	CWE-89 . . . . .	10
3.2.1	Where does this occur? . . . . .	10
3.2.2	Login . . . . .	10
3.2.3	What does the attacker gain? . . . . .	10
3.3	CWE-209 . . . . .	11
3.3.1	Where does this occur? . . . . .	11
3.3.2	Change Pass Function (for example) . . . . .	11
3.3.3	What does the attacker gain? . . . . .	11
3.4	CWE-256 . . . . .	12
3.4.1	Where does this occur? . . . . .	12
3.4.2	What does the attacker gain? . . . . .	12
3.5	CWE-284 . . . . .	13
3.5.1	Where does this occur? . . . . .	13
3.5.2	What does the attacker gain? . . . . .	13
3.6	CWE-613 . . . . .	14
3.6.1	Where does this occur? . . . . .	14
3.6.2	What does the attacker gain? . . . . .	14
3.7	CWE-620 . . . . .	15
3.7.1	Where does this occur? . . . . .	15
3.7.2	Login . . . . .	15
3.7.3	What does the attacker gain? . . . . .	15
3.8	CWE-756 . . . . .	16
3.8.1	Where does this occur? . . . . .	16
3.8.2	What does the attacker gain? . . . . .	16
<b>4</b>	<b>Vulnerability fixes</b>	<b>17</b>
4.1	CWE-79 . . . . .	17
4.2	CWE-89 . . . . .	18
4.3	CWE-209 . . . . .	19
4.4	CWE-256 . . . . .	20

4.5	CWE-284 . . . . .	21
4.6	CWE-613 . . . . .	22
4.7	CWE-620 . . . . .	23
4.8	CWE-756 . . . . .	24
<b>5</b>	<b>Workload and General Information</b>	<b>25</b>

## List of Code Blocks

1	Unsecured Query . . . . .	10
2	Not Generic Error Message . . . . .	11
3	A query executed in the secured app . . . . .	17
4	A query executed in the insecure app . . . . .	18
5	A query executed in the secure app . . . . .	18
6	Generic Error Message . . . . .	19
7	Password encode . . . . .	20
8	Is User Authorized to Comment Function . . . . .	21
9	Logout Session . . . . .	22
10	Session timeout defined . . . . .	22
11	Error pages fix . . . . .	24

# 1 Shop Introduction

Our project is dedicated to creating a straightforward online store website. Within this platform, users have the ability to register new accounts and log in, preserving their session. This means that once a user logs in, the website recognizes them as long as the server is operational. Additionally, users can leave comments on selected items, with the author field being automatically populated by the backend using the session. It's essential to note that if a user isn't logged in, they will be redirected to the authentication page when trying to leave a comment.

The website also provides other features, such as the option to change a user's current password. In our `app.py`, which is the less secure version, there's a lack of input sanitization, making it possible for SQL injection to occur in the login sections. The text fields in article comments are also susceptible to security issues due to the absence of sanitization. Another concern with this app is that it exposes backend code when user input triggers a 500 status error.

Conversely, our `app_sec.py`, which is the secure version, allows us to utilize all of the application's functions without introducing any vulnerabilities.

## **2 CWEs**

### **2.1 CWE-79**

CWE-79, commonly known as 'Cross-Site Scripting (XSS),' involves the improper neutralization of input during web page generation. This vulnerability allows attackers to inject malicious scripts into web pages viewed by other users. These scripts can be used to steal session tokens, redirect users to malicious sites, or perform unauthorized actions on behalf of the victim, leading to a compromise of user data and overall security.

### **2.2 CWE-89**

CWE-89, commonly known as 'SQL Injection,' pertains to improper neutralization of special elements used in SQL commands. This vulnerability is often utilized by attackers to inject unwanted code into database queries, potentially revealing protected information, removing relevant content from tables, and altering protected data.

### **2.3 CWE-209**

This vulnerability occurs when sensitive information about the system is revealed through error messages. This can include internal details such as database table names, column names, or internal structures that may be exploited by an attacker.

### **2.4 CWE-256**

CWE-256 relates to storing passwords in plaintext, meaning that the database contains passwords without any form of protection. Consequently, if an attacker gains access to the database, they can directly access user accounts.

### **2.5 CWE-284**

This vulnerability arises when access control to resources or functionalities is not properly implemented. It can allow unauthorized users to access or modify data that they should not be able to.

### **2.6 CWE-613**

CWE-613, also known as "Insufficient Session Expiration," refers to a security weakness where an application or system does not effectively terminate user sessions after a specified period of inactivity or after a user logs out. This vulnerability can lead to unauthorized access or misuse of an active session by an attacker who gains control over it.

### **2.7 CWE-620**

CWE-620, also known as "Unverified Password Change," refers to a security vulnerability where a system's mechanism for changing passwords lacks proper verification processes. This weakness allows attackers to change a user's password without adequate verification, such as confirming the user's identity or requiring authentication. As a result, unauthorized individuals

can potentially gain control over user accounts and access sensitive information, thereby compromising the security and integrity of the system. Proper verification steps, such as multi-factor authentication or confirmation through email, are essential to prevent unauthorized password changes and protect user accounts.

## **2.8 CWE-756**

CWE-756 addresses the absence of an error page, which is a page presented to the user when some type of error occurs (a page with status code 400, 500, etc.). When this page is not implemented, error page generation becomes the responsibility of the library used for the backend, resulting in descriptive errors with the display of sensitive information (code used for queries).

## 3 Vulnerability Exploit

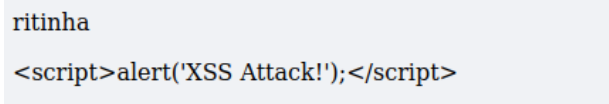
### 3.1 CWE-79

This vulnerability significantly increases the shop's susceptibility to other exploits, such as CWE-352 and CWE-601. Attackers often inject malicious scripts by exploiting the lack of proper input validation, which can be done using tags like `script` or by embedding harmful JavaScript code directly into the web page content.

#### 3.1.1 Where does this occur?

#### 3.1.2 Connect section

By inserting a script tag such as `<script>alert('XSS Attack!');</script>` into a user input field on the application, an attacker can exploit an XSS vulnerability to execute arbitrary JavaScript in the context of another user's browser session. This type of attack can have several severe consequences.



```
ritinha
<script>alert('XSS Attack!');</script>
```

Figure 1: Comment Section

As demonstrated in the previous image, a seemingly simple alert on a web page can easily escalate into a serious security issue, potentially leading to data breaches, identity theft, and other forms of cyber exploitation.

#### 3.1.3 What does the attacker gain?

By injecting malicious code into the application, the attacker can gain unauthorized access to sensitive information and disrupt the normal operation of the site. This includes:

- **Script Execution:** The injected script runs within the victim's browser, executing arbitrary code. For example, a simple script like `<script>alert('XSS Attack!');</script>` can display an alert box. However, more malicious scripts can steal session cookies, capture keystrokes, or perform actions on behalf of the user.
- **Fake Login Forms:** Attackers can use XSS to inject fake login forms or phishing pages into the application. When users interact with these fake forms, their credentials are sent to the attacker rather than being processed by the legitimate application. This can lead to unauthorized access and data theft.
- **Data Theft and Account Compromise:** By executing malicious JavaScript, attackers can exfiltrate sensitive information such as session tokens, login credentials, or personal data from the user's session. This can lead to significant security breaches and account compromise.
- **Impact on User Trust:** Successful XSS attacks can undermine user trust in the application. Users may feel their data is not secure, leading to reputational damage and loss of user confidence.



To mitigate XSS vulnerabilities, it's essential to sanitize user inputs, encode outputs properly, and employ security libraries and frameworks that automatically handle these concerns.

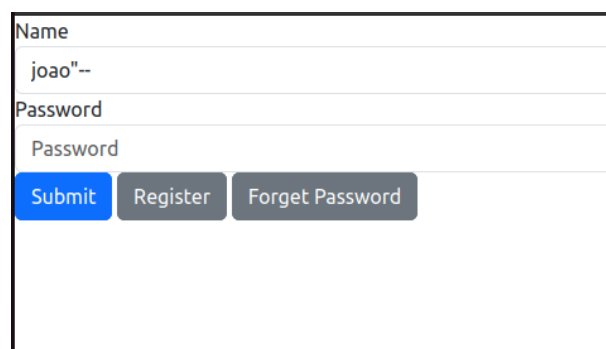
## 3.2 CWE-89

This particular vulnerability makes the shop highly susceptible because it serves as a gateway for many other exploits, including CWE-256 and CWE-756. Attackers predominantly use quotation marks (") and the combination of ("–) to comment out the query.

### 3.2.1 Where does this occur?

#### 3.2.2 Login

By inserting <username>"– into the username field on the login page, the attacker can log in without needing a password since the injected code comments out the part of the program that retrieves the password.



A screenshot of a login page. It features two input fields: 'Name' and 'Password'. The 'Name' field contains the text 'joao"–'. Below the 'Password' field, there are three buttons: 'Submit' (blue), 'Register' (grey), and 'Forget Password' (grey).

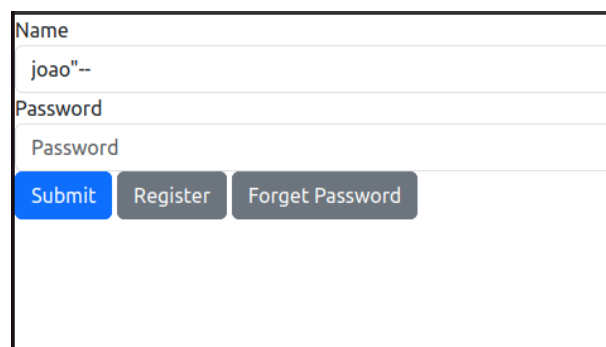
Figure 2: login\_page

#### Code Block 1: Unsecured Query

```
1 res = conn.execute(f'SELECT 1 FROM users WHERE username="{name}" AND  
2 password="{password}"')
```

### 3.2.3 What does the attacker gain?

By injecting some of the described lines of code, the attacker can obtain information to which they should not have access and disrupt the proper functioning of the shop. This includes direct (through login) and indirect (by discovering the password) access to user accounts.



A screenshot of a login page, identical to Figure 2. It features two input fields: 'Name' and 'Password'. The 'Name' field contains the text 'joao"–'. Below the 'Password' field, there are three buttons: 'Submit' (blue), 'Register' (grey), and 'Forget Password' (grey).

Figure 3: login\_page

### 3.3 CWE-209

This vulnerability significantly compromises the security of the application because it exposes sensitive system details through error messages. This exposure can provide attackers with critical insights into the system's internal structure and operations, potentially leading to further exploits. For instance, error messages revealing database schema information or specific error codes can aid attackers in crafting more targeted attacks. By disclosing internal error details, the application inadvertently assists attackers in identifying and exploiting other vulnerabilities, thereby increasing the risk of a successful breach.

#### 3.3.1 Where does this occur?

#### 3.3.2 Change Pass Fuction (for example)

In the `change_pass` function, if an error occurs while updating the password, the error message could reveal detailed information about the issue.

Code Block 2: Not Generic Error Message

```
1 Error updating password: SQL error near 'username': no such column:
2 username
```

#### 3.3.3 What does the attacker gain?

In the context of CWE-209, detailed error messages can provide attackers with valuable information that aids in compromising the application's security. Specifically, an attacker gains:

- **Identification of Internal Structure:** Detailed error messages may expose information about the database schema, file paths, or internal logic of the application. This information can help attackers understand the application's architecture and design.
- **Exploitation of Weaknesses:** By understanding specific error conditions and application behavior, attackers can identify potential vulnerabilities or misconfigurations that can be exploited for unauthorized access or other malicious activities.
- **Facilitation of Further Attacks:** Knowledge gained from error messages allows attackers to refine their techniques and craft more precise attacks. This increased understanding can lead to successful exploitation of other vulnerabilities, such as those related to authentication or data access.

In summary, detailed error messages expose internal details that can be leveraged by attackers to exploit vulnerabilities and compromise the security of the application.

### **3.4 CWE-256**

This vulnerability becomes particularly interesting when combined with SQL injection (CWE-89). With CWE-89, it is possible to retrieve plaintext passwords of one or more users. If passwords are stored in plaintext, the attacker can view the password in clear text. However, if a secure hash is used (e.g., SHA-256), the attacker would only see a hashed representation, which appears as random characters and is not easily reversible.

#### **3.4.1 Where does this occur?**

This occurs in all fields where SQL injection is possible, in the case of the insecure application, specifically in the login page. It also happens whenever someone with database access attempts to view passwords. This is because passwords should not be visible to anyone, even if they are the shop owner or a system administrator.

#### **3.4.2 What does the attacker gain?**

Naturally, in this scenario, the attacker becomes aware of the users' passwords they desire. Consequently, they can access their accounts, enabling them to create comments in the name of those users. They can also change the user's account password, denying access to it.

## 3.5 CWE-284

When CWE-284 is present, attackers can exploit the lack of proper access control to gain unauthorized access to sensitive data or administrative functions. For instance, if an application allows any authenticated user to access an admin panel or modify data that should only be accessible to privileged users, attackers could exploit this flaw to escalate their privileges or manipulate critical system settings.

### 3.5.1 Where does this occur?

In the `submitComment` function, there is no verification to check if the `postId` corresponds to an existing post. Although any user is allowed to comment on posts, the lack of validation on the `postId` can still pose a risk. For example, if a malicious user manipulates the request to alter the `postId`, they could attempt to comment on a non-existent post, which could potentially lead to unexpected behavior or exploitation of other vulnerabilities.

### 3.5.2 What does the attacker gain?

In the context of the `submitComment` function, even though any user can comment on a post, the lack of proper validation on the `postId` still leaves room for potential exploitation:

- **Commenting on Non-Existent Posts:** The attacker could submit comments with manipulated `postId` values, potentially causing errors or unexpected behavior within the application. This could disrupt the normal operation of the commenting system or expose other underlying issues in the code.
- **Insertion of Invalid Data:** By using invalid or non-existent `postId` values, the attacker could insert comments linked to posts that do not actually exist. This could clutter the database with invalid data, potentially leading to difficulties in data management or even application crashes.
- **Exploration of Other Vulnerabilities:** Although any user can comment, the ability to manipulate parameters without proper validation can sometimes be used in conjunction with other vulnerabilities to gain unauthorized access or perform privilege escalation.
- **Potential for System Disruption:** While the direct impact might be limited, consistently injecting invalid `postId` values could be part of a broader strategy to disrupt the normal functioning of the application, which could degrade user experience or compromise system integrity.

This revised text reflects the fact that any user can comment on posts while still highlighting the potential risks associated with not validating `postId`.

## **3.6 CWE-613**

CWE-613, also known as "Insufficient Session Expiration," refers to a security weakness where an application or system does not effectively terminate user sessions after a specified period of inactivity or after a user logs out. This vulnerability can lead to unauthorized access or misuse of an active session by an attacker who gains control over it.

### **3.6.1 Where does this occur?**

It typically arises when a system fails to terminate or expire user sessions adequately after a period of inactivity or after a user logs out. In our application, the session does not have an appropriate timeout. A user can log out without effectively terminating their session, which means that someone with access to the user's computer or device can resume the session without the need for authentication.

### **3.6.2 What does the attacker gain?**

The attacker can access a legitimate user's account without needing to authenticate again, which can lead to privacy breaches, access to confidential information, and malicious actions on behalf of the user.

### 3.7 CWE-620

This vulnerability exploits the change of a user's account password without requiring the entry of the current password. In essence, what we have here is a design flaw on the password change page.

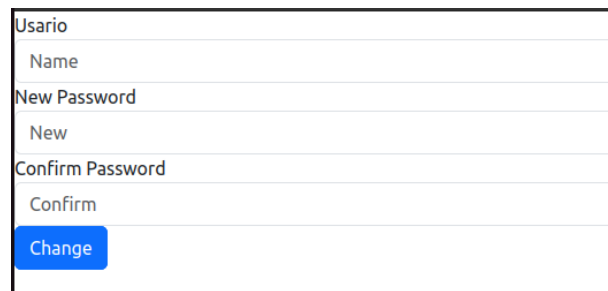
#### 3.7.1 Where does this occur?

#### 3.7.2 Login

This occurs exclusively on the user's change password page, where the user has a button that allows them to fill out a form and change their current password.

#### 3.7.3 What does the attacker gain?

What the attacker gains is access to the user's account to whom the password was changed, denying access to the legitimate user.



Usario
Name
New Password
New
Confirm Password
Confirm
Change

Figure 4: change\_password\_page

### 3.8 CWE-756

As previously explained, this vulnerability exploits the viewing of sensitive information, in this case, the viewing of code executed by the backend for making database queries. This facilitates SQL injection attacks because the attacker knows how the query is constructed.

### 3.8.1 Where does this occur?

This type of attack can occur anywhere we access the database with user-supplied data.

To trigger the query error, the user needs to input a quotation mark (") at the login page to close one of the query parameters and generate a formatting error.

Name

Password

Password

Submit

Register

Forget Password

Figure 5: How to attack

[illegible]

Figure 6: Error after attack

### 3.8.2 What does the attacker gain?

When an error page similar to the image above is displayed, the amount of information provided to the attacker is substantial. They immediately gain insight into how the query is constructed, that it is operating on a table named "table\_name" and the fields used in the query execution. This makes, for instance, a SQL injection attack (CWE-89) much easier to carry out.



## 4 Vulnerability fixes

### 4.1 CWE-79

To prevent Cross-Site Scripting (XSS) attacks, it is crucial to sanitize user inputs to neutralize any potentially malicious code. This involves ensuring that special characters are properly encoded so that they do not get interpreted as executable code by the browser.

One effective way to sanitize inputs is to use libraries designed for this purpose, such as bleach. The bleach library provides functions to safely clean and escape user-generated content, preventing malicious scripts from being executed. By applying such libraries, we can ensure that user inputs are rendered harmless.

Below is an example of how user input should be handled:

Code Block 3: A query executed in the secured app

```
1  
2 comment_text = bleach.clean(request.form['comment_text'])
```

In the example above, `bleach.clean()` is used to sanitize the input, ensuring that potentially harmful script tags are neutralized. This method transforms the dangerous input into a safe format that does not pose a risk to users or the application.

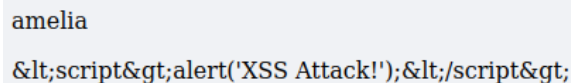


Figure 7: XSS Fix

By incorporating proper input sanitization techniques and using libraries like bleach, we can effectively mitigate the risk of XSS attacks and maintain the security and integrity of web applications.

## 4.2 CWE-89

To ensure that an attacker cannot execute a SQL injection, we need to sanitize the input, meaning that certain critical characters, such as ”—, single quotes (’), and double quotes (”), have to be replaced with characters that do not enable the attack. We also need to adjust the way we construct queries.

Code Block 4: A query executed in the insecure app

```
1  
2 res = conn.execute(f'SELECT 1 FROM users WHERE username="{name}" AND  
password="{password}"')
```

Code Block 5: A query executed in the secure app

```
1  
2 res = c.execute("SELECT * FROM users WHERE username=?", (name,))
```

This way, we neutralize the query attack.

### 4.3 CWE-209

To mitigate this vulnerability, avoid exposing detailed error messages. Instead, provide generic error messages to users while logging detailed error information internally for analysis.

Code Block 6: Generic Error Message

```
1  
2 return redirect(url_for('changePass_sec_page', error='An unexpected  
   error occurred'))
```

## 4.4 CWE-256

To fix this vulnerability, we had to encrypt the user account passwords at the time of registration. As a result, the password stored in the database is already encrypted. Therefore, in the event an attacker gains access to the database, they will not be able to determine the user passwords.

During the login process, the entered password for validation is encrypted and then compared with the value already stored in the database associated with the user. If the entered password is correct, the value returned by function must match the value already stored in the database.

Code Block 7: Password encode

```
1  
2 if bcrypt.checkpw(password.encode("utf-8"), data[3]):
```

## 4.5 CWE-284

To mitigate the risk of unauthorized actions, ensure that user input is validated properly. Specifically, verify that the `postId` corresponds to an existing post and that the user is allowed to interact with it. Additionally, implement robust validation checks to prevent users from manipulating parameters that could lead to unintended behavior or access to resources they should not modify.

A function, `is_user_authorized_to_comment`, is used to verify the validity of the `postId` provided by the user. This function checks whether the `postId` corresponds to an actual post within the database, ensuring that users can only comment on existing posts. Additionally, it ensures that the user has appropriate permissions to interact with the specified post. By performing these checks, the function helps prevent the insertion of comments linked to non-existent or unauthorized posts, thereby maintaining the integrity of the application's commenting system.

Code Block 8: Is User Authorized to Comment Function

```
1
2 def is_user_authorized_to_comment(title, user_id):
3     # Connect to the database
4     conn = sqlite3.connect("database/shop_sec.db")
5     c = conn.cursor()
6
7     # Execute the query to select all columns from the posts table
8     c.execute("SELECT * FROM posts")
9
10    # Fetch all results
11    all_posts = c.fetchall()
12
13
14    # Close the database connection
15    conn.close()
16
17    # Check if any post's title matches the provided title
18    for post in all_posts:
19        if post[1] == title: # Assuming the title is in the second
20            column
21            return True
22
23    # Return False if no matching title is found
24    return False
```

## 4.6 CWE-613

To address this vulnerability, we had to implement a session timeout for user sessions. In other words, if the session expired, the user would have to log in again. Additionally, after the user logs out, their session is now terminated, which was not the case previously.

Code Block 9: Logout Session

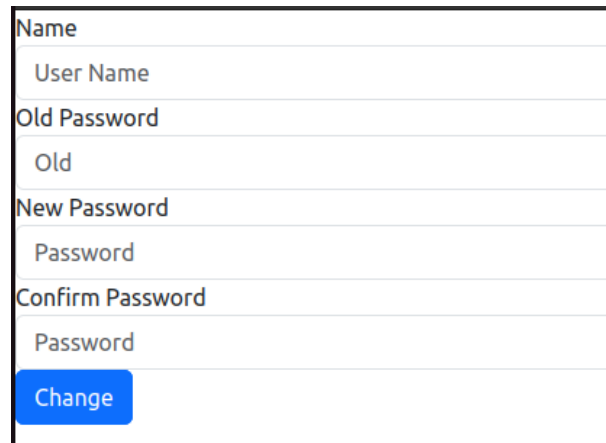
```
1
2 def logout():
3
4     if 'user_id' in session:
5         session['user_id'].pop()
6     return redirect("/")
```

Code Block 10: Session timeout defined

```
1
2 app.permanent_session_lifetime = timedelta(minutes=10)
```

## 4.7 CWE-620

The rectification of this vulnerability is clear and easily executable. When changing the password, a new parameter is added to allow password change. The added parameter is the current password, thus maintaining the feature that allows a user to change the password of an account they are not logged into, while adding an authenticity check. Once this pseudo-login is verified, and if the new password complies with the requested format, the password for the respective user is changed.



Name
User Name
Old Password
Old
New Password
Password
Confirm Password
Password
<input type="button" value="Change"/>

Figure 8: Old Password Requirement

## 4.8 CWE-756

The correction of this vulnerability requires the creation of an error page (responses with status codes of 400, 500, etc.). This page should not contain any sensitive information (source code) to provide as little information as possible to the user about the system.

In the event of any error, the application should not display the error page generated by the backend library but rather the error page created by the system programmer.

Code Block 11: Error pages fix

```
1
2 @app.errorhandler(404)
3 def page_not_found(error):
4     return render_template("Error/404.html", error_code=404), 404
5
6 @app.errorhandler(Exception)
7 def internal_server_error(error):
8     return render_template("Error/500.html", error_code=500), 500
```



## 5 Workload and General Information

The repository for the project is [https://github.com/detiuaveiro/1st-project-group\\_37/](https://github.com/detiuaveiro/1st-project-group_37/).  
Each team member made an equitable contribution to the project's resolution:

- João Ferreira (100%)