



# Vulnerabilities in software products

Security in Informatics and Organizations

## Authors:

- > Daniel Madureira, n.º 107603
- > Diana Miranda, n.º 107457
- > Miguel Pinto, n.º 107449
- > Pedro Ramos, n.º 107348



universidade  
de aveiro



**deti**  
universidade de aveiro



# INDEX

INTRODUCTION .....	3
OUR IMPLEMENTATION .....	4
PROJECT DESCRIPTION .....	4
SIMPLE ARCHITECTURE OVERVIEW .....	5
FRONTEND .....	5
BACKEND .....	5
DATABASE .....	6
DOCKERS .....	6
APPLICATION FEATURES AND CAPABILITIES .....	7
USER FEATURES .....	7
ADMIN FEATURES .....	7
HOW TO RUN THE PROJECT .....	8
DOCKERIZED FRONTEND .....	8
DOCKERIZED BACKEND .....	8
FRONTEND .....	9
BACKEND .....	9
VULNERABILITIES .....	10
TABLE OF VULNERABILITIES .....	10
VULNERABILITY DESCRIPTION .....	11
CWE 79 CROSS-SITE SCRIPTING .....	11
CWE 89 SQL Injection .....	13
CWE 20 Improper input validation .....	16
CWE 521 Weak password requirements .....	19
CWE 434 Unrestricted upload of a file with dangerous type .....	20
CWE 862 Missing authorization .....	21
CWE 256 Plaintext storage of a Password .....	22
CWE 287 Improper Authentication .....	24
CWE 201 Insertion of sensitive information into sent data .....	25
FINAL OVERVIEW AND CONCLUSION .....	26



# INTRODUCTION

By abstracting away considerations of popularity, the potential user base of a particular informatic system expands significantly each year. With this growing user base, any well-constructed application should have the capacity to efficiently handle the performance requirements necessary to maintain the system's capabilities while still meeting expected response times and error rates.

One of the most frequently overlooked aspects of a given implementation is its defense against unforeseen attacks, whether originating from malicious actors or arising from natural events, such as earthquakes or power outages.

In this project, we aim to focus on the first source. We have developed a mostly complete mock-up implementation of a product with the intended of testing vulnerabilities to determine just how easily one can manipulate a given system and how challenging it is to safeguard it.

As programmers, we must bear in mind that end users will often discover ways to disrupt something, even unintentionally. Therefore, it is our responsibility to prevent the occurrence of persistent issues in our implementations due to improper usage or targeted attacks.



# OUR IMPLEMENTATION

## PROJECT DESCRIPTION

This project involves the creation of two similar web-based applications designed to simulate a small web store that sells DETI related memorabilia. The shop should have a variety of products, including mugs, T-shirts, etc. Both applications should operate without errors or inconsistent behaviours, and should be visually and functionally similar, but one should contain hidden security vulnerabilities, while the other should prevent abuses from these weak points as much as possible.

These vulnerabilities should not be easily found by the normal user but should be significant enough for malicious actors to identify them.

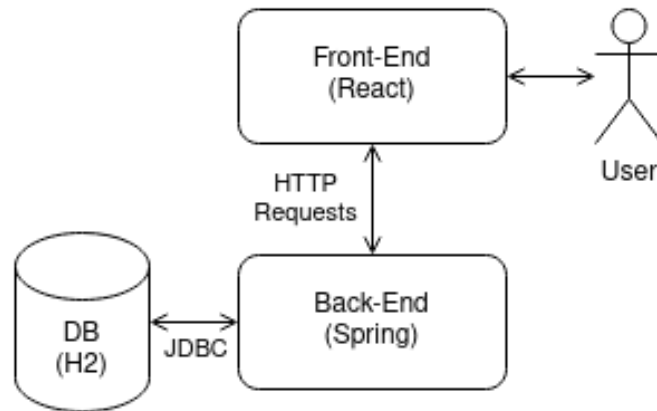
The application itself is composed by a web-based user interface that allows users to select and buy desired items.

The users are categorized into two roles, 'user' and 'admin'. Administrative accounts possess all base functionalities of user accounts plus product management functions, which we will discuss later. All accounts must be able to view all the products, add them to a cart and purchase them.

The frontend communicates with a separate backend server through a set of HTTP Requests, maintaining a clear separation between the user interface and the core functionality of our project. The backend then manipulates the database accordingly.

# SIMPLE ARCHITECTURE OVERVIEW

Our architecture can be divided into three separate layers:



## FRONTEND

The user interface of the application was developed using React.

We opted for this technology due to its numerous advantages in modern web development. React allows us to build reusable components, facilitate page updates, enhance performance through its Virtual DOM, and take advantage of its extensive ecosystem of libraries and tools. This translates to a more seamless user experience, streamlined development, and the ability to create a highly scalable web application.

Additionally, some members of our development team have prior experience with React from previous projects and are well-versed in its usage.

## BACKEND

Our application's backend was built using Spring Boot. This choice enabled us to create the moderately complex REST API required to provide all the necessary functionality for the shop's operation to end-users.

Once again, some members of our development team already had prior experience with this tool, allowing us to leverage their knowledge and expand our backend to meet the requirements of the frontend application.

The backend is responsible for translating and parsing requests from the frontend to the database layer. Since it operates entirely on the server side, it serves as a critical barrier to thwart most potential attacks.

Given that HTTP requests are fundamentally insecure (limited authentication, encoding, etc.), the backend has been prepared with corresponding security measures and a defensive mentality in mind.

All the requests are categorized into groups with similar contexts (user management, product management, reviews, etc.) and all administrative functions require an admin account to access.

# DATABASE

The final layer of our application is the database layer.

We tested various databases (PostgreSQL, MariaDB, MySQL, etc.) and determined that the in-memory H2 database was the most suited for testing a production system like ours.

H2 eliminates the need for defining schemas or users in Spring, and since the database resets each time we reload Spring, we can confidently experiment with SQL attacks without worrying about breaking anything. For testing purposes, we pre-load the database with some data every time it is restarted.

The only layer that interacts with the database is the spring backend, so we must ensure Spring is capable of both identifying and blocking potentially harmful SQL queries.

# DOCKERS

To facilitate testing and experimentation with the application while minimizing the risk of compromising our own system, as well as ensuring that everyone can run our programs without having to deal with complex installation procedures and dependency issues, we containerized both the front-end and the back-end.

These containers run completely separate from each other.



# APPLICATION FEATURES AND CAPABILITIES

## USER FEATURES

For optimal and more creative testing, we implemented these shop features available to all users:

- User management:
  - Create users;
  - See user profiles;
  - Update user password;
  - Check User logins.
- Product Catalog:
  - List products;
  - Filter the list of products based on category;
  - Search for products based on the name;
  - Keep track of product availability and stock.
- Shopping Cart:
  - Add, remove and update shopping cart items;
  - Get the total cost of the order;
  - Save the cart between sessions.
- Checkout:
  - Order the current shopping cart;
  - Minimal payment processing;
  - Update product quantities;
  - Order confirmation and receipt generation;
- Order history:
  - See all previously made orders;
- Comments and reviews:
  - See all reviews for a given product;
  - Post comments/reviews with ratings;
  - See average rating for the products.

## ADMIN FEATURES

An administrative account has access to the same methods as user accounts, with a few additional features such as:

- Product Management:
  - Change update availability and stock;
  - Add new products to the shop.



# HOW TO RUN THE PROJECT

As per the assignment, our project has two versions. Their run commands are the same but executed in their respective directories.

## DOCKERIZED FRONTEND

Navigate to the desired version:

```
# Assuming you're in the root of the repository  
cd <app | app_sec>
```

Go to frontend directory:

```
cd frontend
```

Build the container:

```
docker build -t react-docker:latest .
```

Run the container:

```
docker run -p 5173:5173 react-docker:latest
```

## DOCKERIZED BACKEND

Navigate to the desired version:

```
# Assuming you're in the root of the repository  
cd <app | app_sec>
```

Go to frontend directory:

```
cd backend
```

Build the application:

```
./mvnw install
```



Build the container:

```
docker build --build-arg JAR_FILE=target/shop_backend-0.0.1-SNAPSHOT.jar  
-t com/shop_backend .
```

Run the container:

```
docker run -p 8080:8080 com/shop_backend
```

## FRONTEND

Navigate to the desired version:

```
# Assuming you're in the root of the repository  
cd <app | app_sec>
```

Go to frontend directory:

```
cd frontend
```

Install dependencies:

```
yarn  
- ou -  
npm install
```

Run project:

```
yarn dev  
- ou -  
npm run dev
```

## BACKEND

Navigate to the desired version:

```
# Assuming you're in the root of the repository  
cd <app | app_sec>
```

Go to frontend directory:

```
cd backend/Backend
```

Run project:

```
./mvnw spring-boot:run
```



# VULNERABILITIES

When examining the insecure application, many vulnerabilities are not immediately obvious to the end customer, but even an inexperienced malicious actor can readily exploit some of the shop's functions.

In this topic we will highlight and evaluate the most critical vulnerabilities identified in the insecure version of the application, taking into account their complexity and the potential damage they could inflict on the system if exploited by someone with malicious intent.

## TABLE OF VULNERABILITIES

For the purposes of this report, we have grouped all the discovered vulnerabilities into the following table.

N.	CWE	Description	Correction
1	CWE 79	Cross-Site Scripting	Reacts 'escape hatch' mechanisms.
2	CWE-89	SQL Injection	Use parameterized queries.
3	CWE 20	Improper input validation	Add checks to all the inputs in the backend and frontend.
4	CWE 521	Weak password requirements	Implement password checks and stronger requirements.
5	CWE 434	Unrestricted upload of a file with dangerous type	Restrict file input to only accept file types of jpg, png, jpeg, tiff, and webp.
6	CWE 862	Missing authorization	Remove the ability to change permissions based on URL parameters and add more authorization checks.
7	CWE 256	Plaintext storage of a Password	Implement secure password salt and hashing. Never save or provide the plaintext password.
8	CWE 287	Improper Authentication	Implement a Token based authentication system. Never save credentials in any easy accessible location.
9	CWE 201	Insertion of sensitive information into sent data	More output validation and outgoing information selection on the backend.

# VULNERABILITY DESCRIPTION

## CWE 79 CROSS-SITE SCRIPTING

Reflected and Stored Cross-Site Scripting (XSS) vulnerabilities pose significant threats to web security. In the case of Reflected XSS, attacks occur when an attacker manipulates input data to execute malicious scripts in the user's browser, compromising data integrity and confidentiality. On the other hand, Stored XSS allows attacks to persist as malicious scripts are to be stored on the server, affecting multiple users.

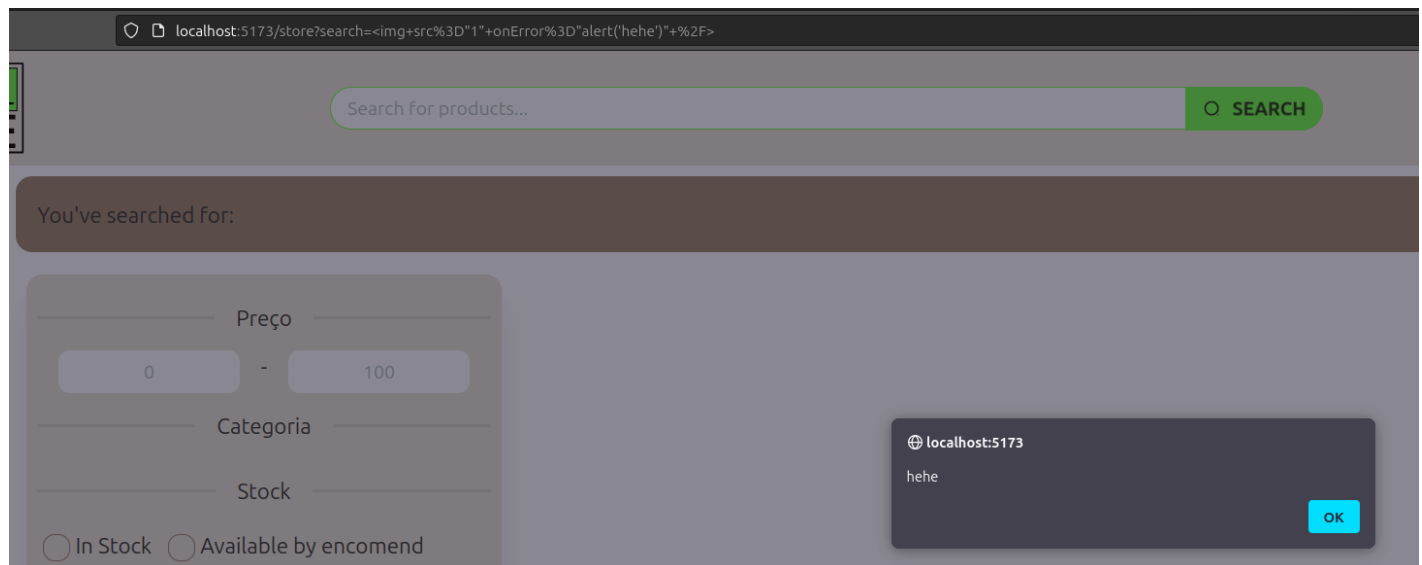
### REFLECTED XSS

If a user searches for a product and inputs:

```

```

this alert will execute, making the website vulnerable.



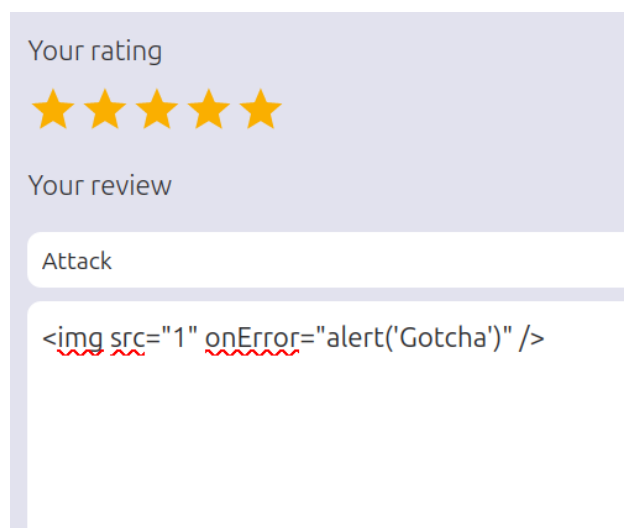
### REFLECTED XSS

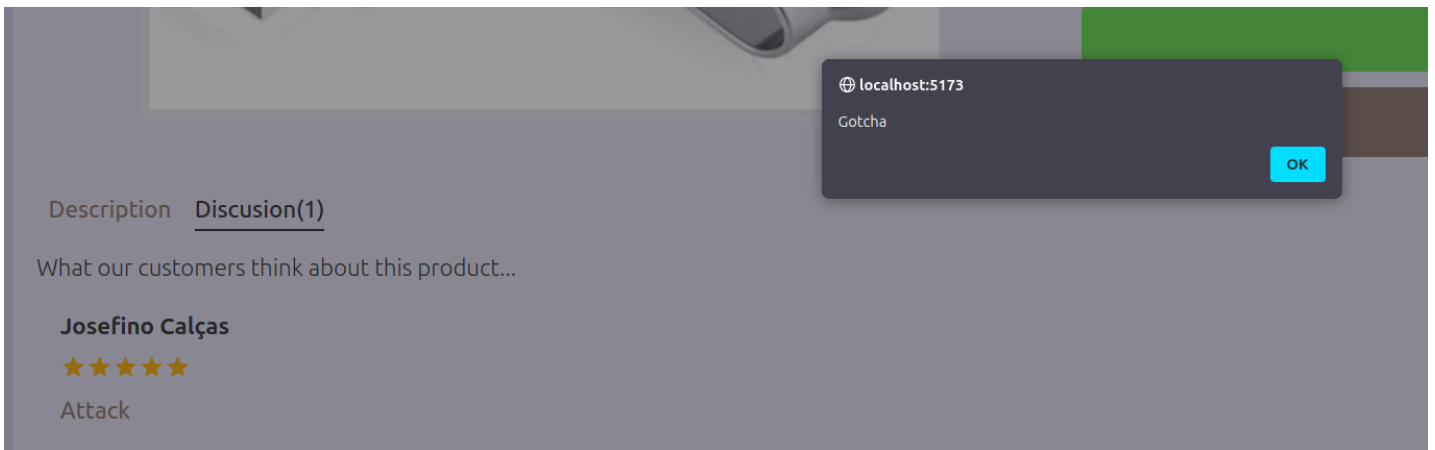
In this attack, the user posts a comment on a product with the payload.

```

```

This comment is stored, and whenever any user views comments on that product, the injected alert is triggered. This allows the attack to be executed.





## RESOLUTION

To address Cross-Site Scripting (XSS) vulnerabilities in the secure application, we have chosen to use React's 'escape hatch' mechanisms instead of relying on native JavaScript techniques like `document.element.innerHTML` to insert dynamic data into the HTML. By employing these React's mechanisms, we ensure that data is properly escaped, treated as plain text, and not interpreted as JavaScript code. This prevents the execution of malicious scripts in the user's browser and maintains consistency in rendering dynamic data, reducing the XSS attack surface and enhancing application security.

```
<h3>
  You've searched for: {search_query}
</h3>
```

instead of

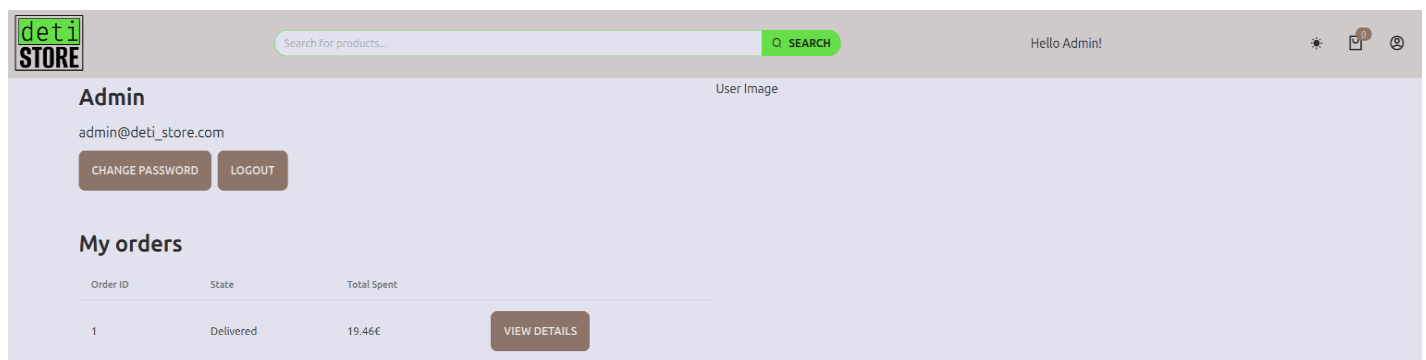
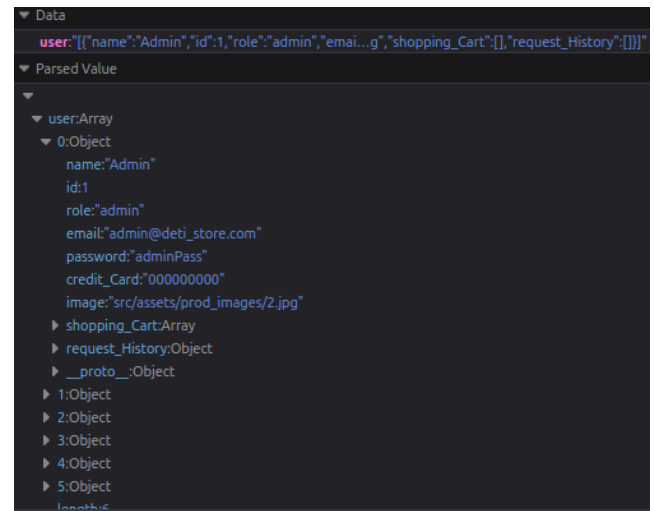
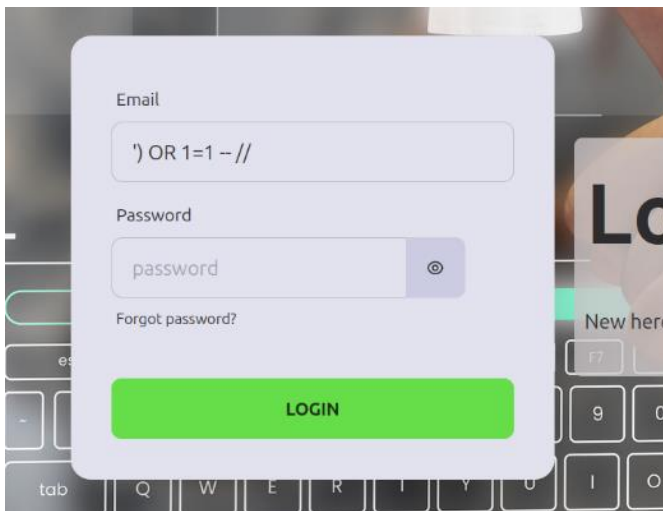
```
document.getElementById("show_query_results").innerHTML = search_query;
...
<h3>
  You've searched for:{ " "}
  <span className="font-extrabold" id="show_query_results"></span>
</h3>
```

# CWE 89 SQL INJECTION

For this CWE, our found vulnerabilities were Basic SQL Injection, SQL Second Order, and Blind SQL Injection.

## BASIC SQL INJECTION

In this scenario, the attacker didn't specify any user and still managed to make the application ignore the rest of the query. The outcome of this attack is exceptionally severe. Not only did the attacker successfully log in as an admin (or any account based on their email address), but they also gained access to all user data, since this information was received from the server and saved into their local storage.



## SQL SECOND ORDER ATTACKS

In this attack, the user created an account with the username `' UNION SELECT 1, table_name, column_name FROM information_schema.columns WHERE table_name = 'APP_USER' -- //`.

After successfully logging in and attempting to change the password, the operation fails due to an error. However, the final query was executed with the malicious username included, granting access to the table schema information.



# Register

Name

Email

Password

Card Number

Image

BROWSE... aa.jpg

REGISTER

## Change Password!

New Password

Confirm New Password

SUBMIT CLOSE

```

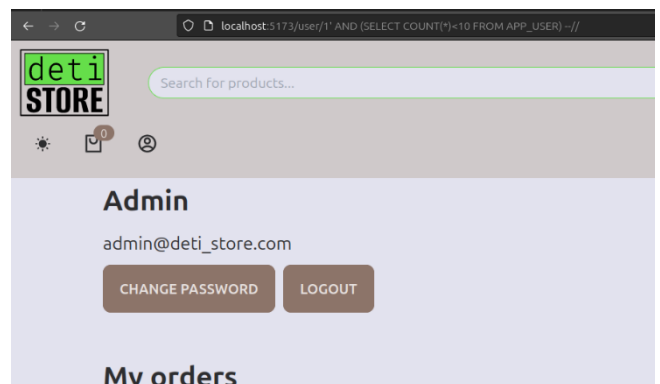
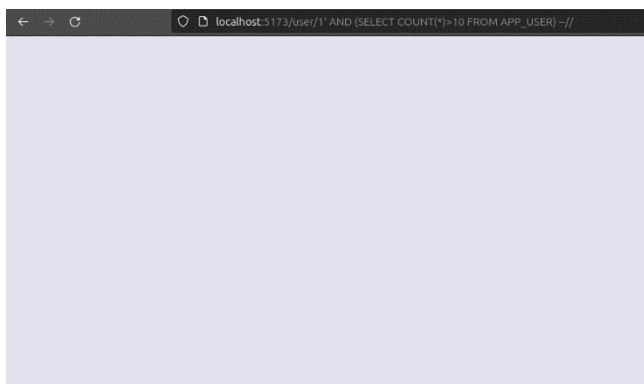
▼ Array(7) [ (3) [...], (3) [...], (3) [...], (3) [...], (3) [...], (3) [...], (3) [...]]
  ▶ 0: Array(3) [ 1, "APP_USER", "CREDIT_CARD" ]
  ▶ 1: Array(3) [ 1, "APP_USER", "EMAIL" ]
  ▶ 2: Array(3) [ 1, "APP_USER", "ID" ]
  ▶ 3: Array(3) [ 1, "APP_USER", "IMAGE" ]
  ▶ 4: Array(3) [ 1, "APP_USER", "NAME" ]
  ▶ 5: Array(3) [ 1, "APP_USER", "PASSWORD" ]
  ▶ 6: Array(3) [ 1, "APP_USER", "ROLE" ]
    length: 7
  ▶ <prototype>: Array []

```

## BLIND SQL INJECTION

In this attack, the user attempted to determine how many registered users exist on the website. To achieve this, he modified a typical `/user/?id` URL by adding `' AND (SELECT COUNT(*)>10 FROM APP_USER) --//`. Due to the website's lack of proper protection, this query was executed, returning 'False' and causing an error, which resulted in nothing being displayed on the screen.

Subsequently, the attacker changed the query to `' AND (SELECT COUNT(*)<10 FROM APP_USER) --//`, which returned 'True', allowing the website to continue its normal operation. This allowed the attacker to confirm that there were fewer than 10 users registered on the site. With a few more attempts, he could easily determine the exact number.



# RESOLUTION

To address all these SQL injection issues, we opted to use parameterized queries in the backend, which makes it impossible to maliciously manipulate the application through injected SQL statements. This solution involves rewriting our SQL queries to use placeholders for user input. These placeholders are then filled with user-supplied data in a safe and controlled manner, ensuring that any malicious SQL code provided by users is treated as data and not executed as SQL commands. By adopting this approach, we enhance the security of our application, safeguard sensitive data, and minimize the potential impact of SQL injection vulnerabilities.

We implemented this programmatically by using a Spring-supported database management system called Hibernate. Hibernate is capable of safely parsing and blocking potentially dangerous SQL commands, as well as having a native safe builder for queries to completely avoid using unsafe dynamic queries.

```
public interface App_UserRepo extends CrudRepository<App_User, Integer> {
    @Query(value="SELECT * FROM app_user usr", nativeQuery=true)
    List<App_User> listapp_users();

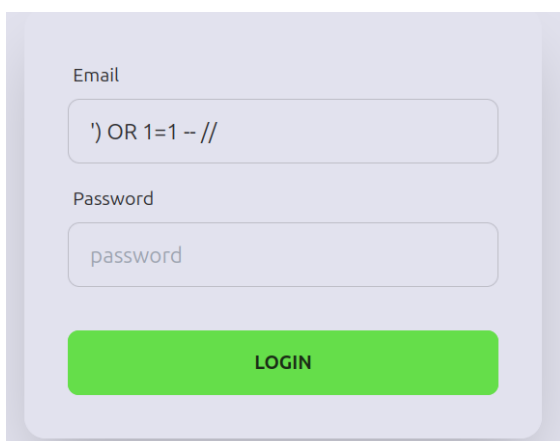
    @Query(value="SELECT * FROM app_user usr WHERE usr.Role = :type",
nativeQuery=true)
    List<App_User> listapp_usersByType(@Param("type") String type);

    @Query(value="SELECT COUNT(id) FROM app_user prod", nativeQuery=true)
    String getNumberOfapp_users();

    @Query(value="SELECT * FROM app_user WHERE id = :id", nativeQuery=true)
    App_User findapp_userByID(@Param("id") Integer id);

    @Query(value="SELECT * FROM app_user WHERE email = :email", nativeQuery=true)
    App_User findapp_userByEmail(@Param("email") String email);

    @Query(value="SELECT * FROM app_user WHERE email = :email AND password = :pass",
nativeQuery=true)
    App_User findapp_userByEmailAndPassword(@Param("email") String email,
@Param("pass") String pass);
}
```



Email

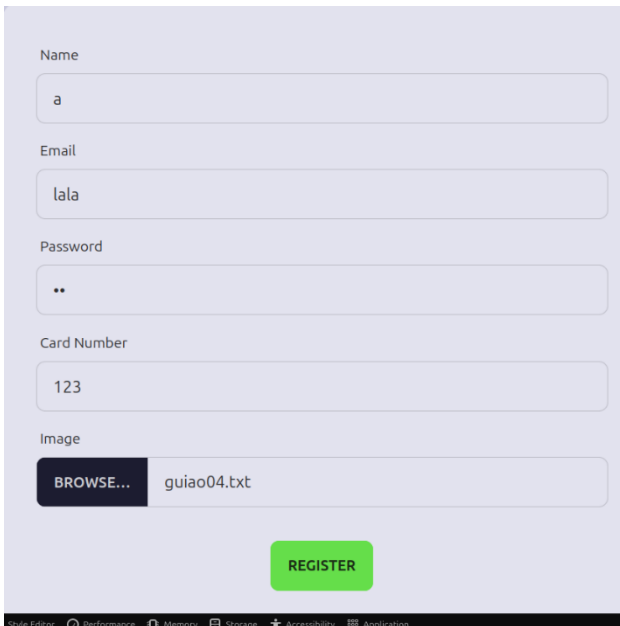
Password

LOGIN

```
! ▶ Error during API call TypeError: response is undefined
    handleLogin LoginPage.jsx:23
    ▶ React 23
    <anonymous> main.jsx:9
```

# CWE 20 IMPROPER INPUT VALIDATION

This vulnerability is also serious, as the insecure app lacks validation on user inputs. This can lead to security issues, such as data injection, information corruption, data leakage, unauthorized execution, and compromise of system integrity.



Name  
a

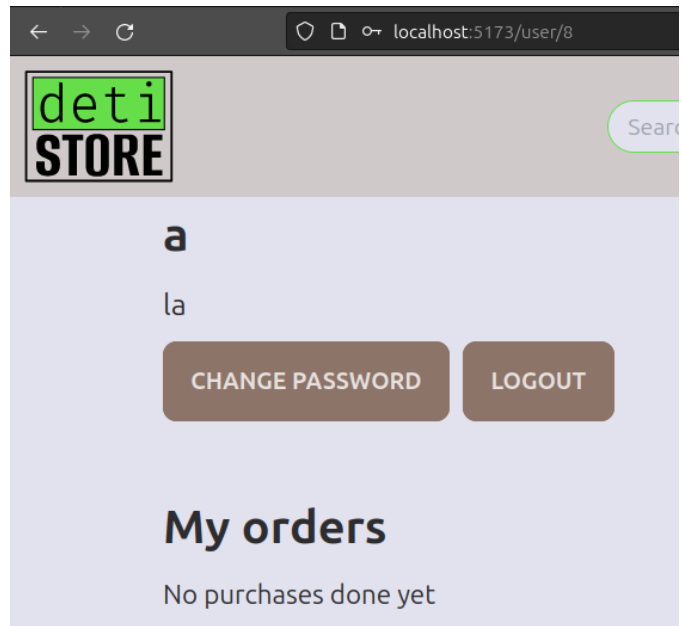
Email  
lala

Password  
..

Card Number  
123

Image  
BROWSE... guiao04.txt

REGISTER



In this example, the user was allowed to create an account with an obviously fake email and name, along with an extremely weak password.

## RESOLUTION

To prevent incorrect and dangerous processing of information provided by user input, such as empty strings or Stored SQL injections, it's crucial to validate all user input to make sure that no variable can carry malicious code that might be used later in the application's execution. The most effective approach we have identified is to parse and validate the inputs as soon as they reach the backend and before they are used in the frontend.

Any rejected data must be immediately discarded, and an error must be thrown back to the source of such data.

### Frontend solution

```
{
    setShowAlertName(true);
} else {
    setShowAlertName(false);
}
if (password !== newPassword) {
    setShowAlertSamePass(true);
} else {
    setShowAlertSamePass(false);
}
if (!emailRegex.test(email)) {
    setShowAlertEmail(true);
}
```

```

} else {
    setShowAlertEmail(false);
}
if (!cardNumberRegex.test(cardNumber)) {
    setShowAlertNumberCard(true);
} else {
    setShowAlertNumberCard(false);
}
if (!imageRegex.test(image.name) || image.size > 5000000) {
    setShowAlertImage(true);
} else {
    setShowAlertImage(false);
}

if (
    showAlertPass ||
    showAlertName ||
    showAlertSamePass ||
    showAlertEmail ||
    showAlertNumberCard ||
    showAlertImage
) {
    return;
}
...
};

```

Name

Asd

Email

fakemail

⚠ Warning: Email must be valid!

Password

...

⚠ Warning: Password must have at least 8 characters, 1 uppercase, 1 lowercase, 1 number and 1 special character!

Confirm Password

...

Card Number

678

⚠ Warning: Card Number must have 12 numbers!

Image

CHOOSE FILE

logging.c

⊗ Error: Image must be a .jpg, .jpeg, .png or .webp file and the size should be less than 5mb!

REGISTER

## Backend solution

```
// Check the given email is already associated with another user
if (app_userRepository.findapp_userByEmail(email) != null) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "A user
with this email already exists!");
}

String emailRegex = "[a-zA-Z0-9_+&*-]+(?:\\.[a-zA-Z0-9_+&*-]+)*@" +
    "([a-zA-Z0-9-]+\\.)+[a-z" +
    "A-Z]{2,7}$";
Pattern pat = Pattern.compile(emailRegex);
// Check if the email is valid
if (!pat.matcher(email).matches()) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY,
        "The provided Email must be valid!");
}

String passwordRegex = "(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=.*[a-zA-
Z])(?=.*[!@#$%^&*()_+]).{8,}$";
pat = Pattern.compile(passwordRegex);
// Check if the password is valid
if (!pat.matcher(password).matches()) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY,
        "The provided Password must have more than 8 characters, 1 lowercase, 1
uppercase, 1 number and 1 special character!");
}

// Check if the card number has 12 characters in lenght
if (cartao.length() != 12) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY,
        "The Card number must have twelve digits!");
}

if (img != null) {
    String OGfileName = img.getOriginalFilename();
    String fileExtention = OGfileName.substring(OGfileName.lastIndexOf(".") +
1);
    String[] a = {"png", "jpeg", "jpg", "tiff", "tif", "webp"};

    // Check file type
    if (!Arrays.asList(a).contains(fileExtention)) {
        throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY,
            "The image file must be of the png, jpeg, jpg, tiff, tif or webp
type!");
    }
}
}
```



# CWE 521 WEAK PASSWORD

## REQUIREMENTS

Inadequate password requirements present a significant security threat. It allows users to choose weak and easily guessable passwords, making their accounts vulnerable to brute-force attacks and the risk of unauthorized access. Weak password complexity makes it much easier for attackers to guess passwords, significantly increasing the likelihood of successful breaches.

As demonstrated in the prior vulnerability example, it was possible to create an account with a password containing only two characters, which is an extremely weak password and easily guessable.

## RESOLUTION

To address this vulnerability, it is crucial to implement password requirements that promote strong passwords and educate users on best practices for password selection. With this in mind, we have introduced password validation in the secure application that enforces the following criteria:

- Passwords must be eight characters or longer.
- Passwords must include at least one lowercase letter.
- Passwords must include at least one uppercase letter.
- Passwords must include at least one special character.

Additionally, we have added a password confirmation field to ensure users do not make mistakes while entering their passwords.

```
const handleRegister = async (event) => {
  ...
  const passwordRegex =
    /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[a-zA-Z])(?=.*[!@#$%^&*()_+]).{8,}$/;
  ...
  if (!passwordRegex.test(password)) {
    setShowAlertPass(true);
  } else {
    setShowAlertPass(false);
  }
  ...
  if (password !== newPassword) {
    setShowAlertSamePass(true);
  } else {
    setShowAlertSamePass(false);
  }
  ...
};
```

# CWE 434 UNRESTRICTED UPLOAD OF A FILE WITH DANGEROUS TYPE

In the example of CWE 20, it is evident that there was no problem when the user uploaded a text file in a field where an image should have been inserted. This is a significant security concern because unverified file uploads can lead to the execution of malicious scripts, code injection attacks, data leakage, and misuse of server resources.

## RESOLUTION

The most straightforward approach to dealing with potentially dangerous file types is to restrict the allowed file extensions in input fields, permitting only specific types like image files (e.g., .png, .jpg).

By limiting the file extensions, we can prevent malicious users from uploading potentially harmful files to our server. Additionally, we have imposed restrictions on image size, allowing only images with a maximum size of 5MB.

This security method should be implemented as a "Allow none but with a few exceptions" policy, which means that all file formats should be blocked with very few exceptions.

User education on secure uploading practices and the importance of not sharing sensitive information through uploaded files is also crucial. Combining strict checks and user education is essential to ensure the security and integrity of the website.

### Frontend solution

```
const handleSubmit = async (event) => {
  ...
  const imageRegex = /\.(jpe?g|tiff?|png|webp)$/i;
  if (!imageRegex.test(image.name) || image.size > 5000000) {
    setShowAlertImage(true);
  } else {
    setShowAlertImage(false);
  }
  ...
};
```

### Backend solution

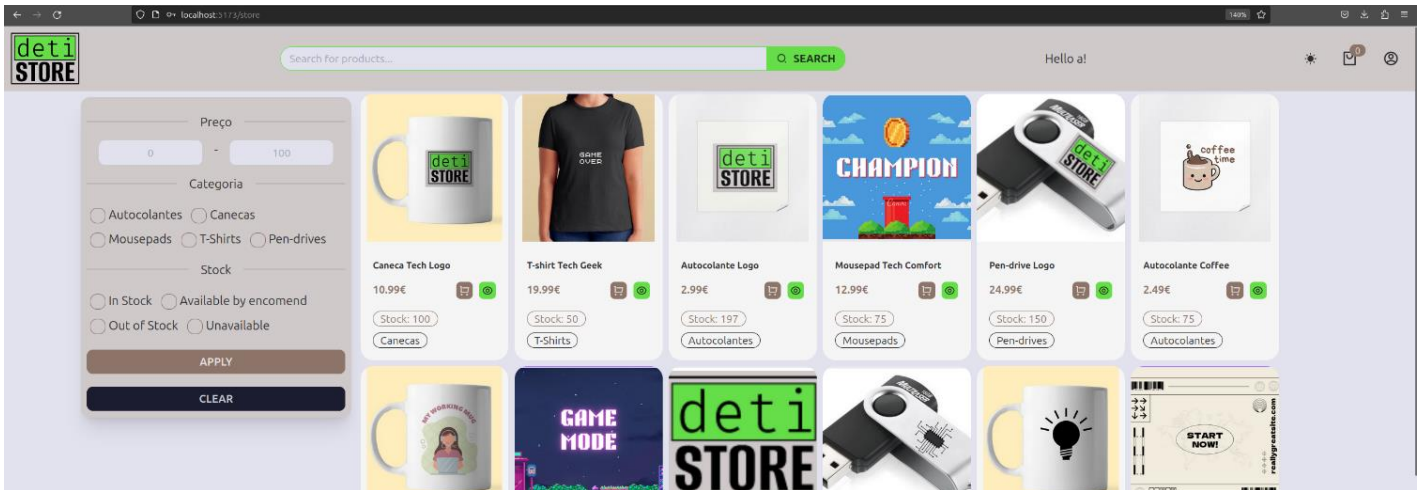
```
if (img != null) {
  String OGfileName = img.getOriginalFilename();
  String fileExtension = OGfileName.substring(OGfileName.lastIndexOf(".") + 1);
  String[] a = {"png", "jpeg", "jpg", "tiff", "tif", "webp"};

  // Check file type
  if (!Arrays.asList(a).contains(fileExtension)) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY,
      "The image file must be of the png, jpeg, jpg, tiff, tif or webp type!");
  }
}
```

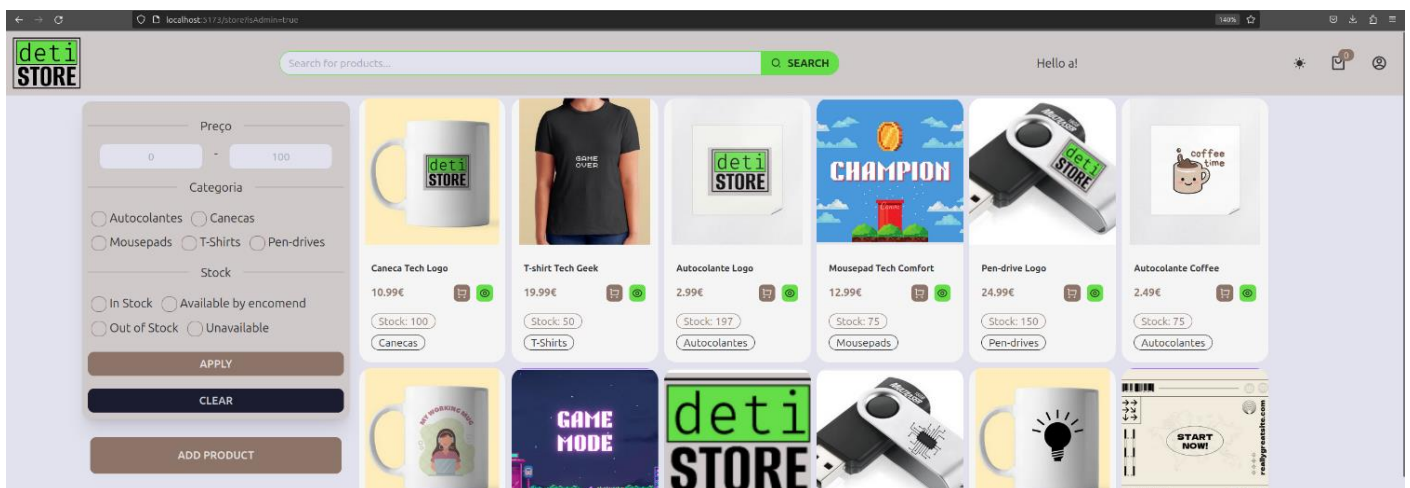
# CWE 862 MISSING AUTHORIZATION

This vulnerability occurs when proper authorization controls are not applied, allowing users to access resources or perform actions without the appropriate permission. This can lead to unauthorized access, privacy breaches, privilege escalation, and data manipulation. In our insecure app, this occurs when a regular user discovers that to gain access to admin permissions, all he needs to do is manipulate the URL by adding the parameter `'isAdmin=True'`.

Store page with a regular authenticated user:



The same user, but with `?isAdmin=True` in the URL. He acquired admin privileges, enabling him to add new products and perform action they were not supposed to:

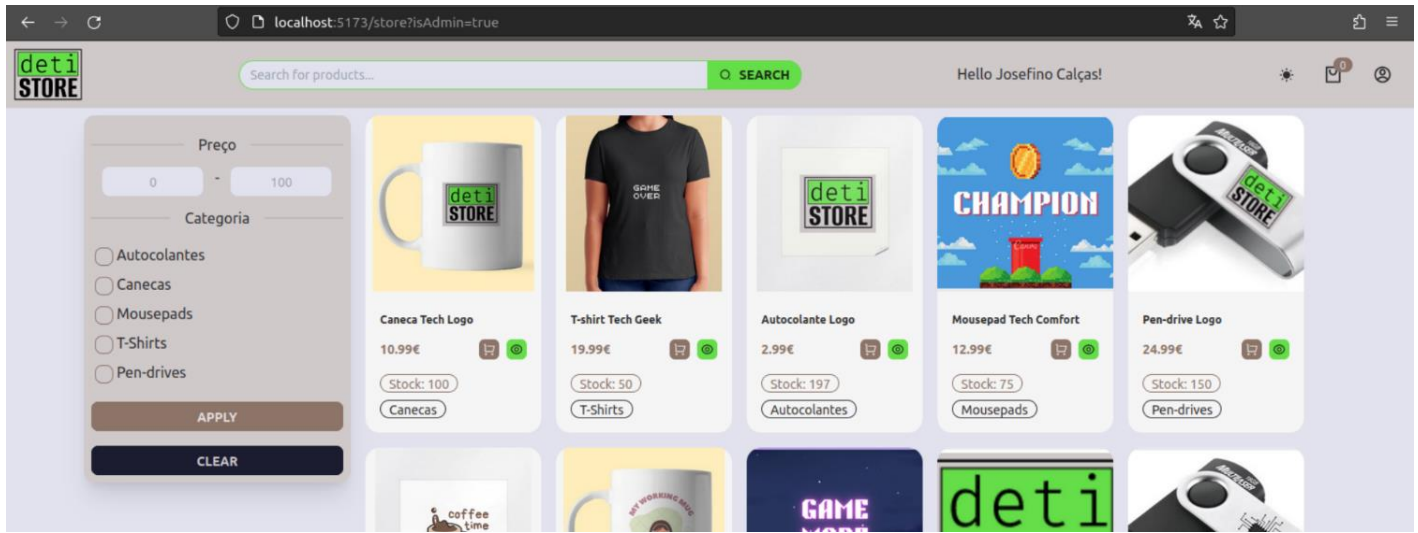


Managing this issue in our frontend is particularly challenging because our application's user interface is the same for both regular users and administrators. The pages dynamically adapt to the user's role, displaying more or fewer options accordingly. Since this adaptation occurs locally in the user's browser, any user can easily enter "admin mode" by manipulating internal website variables to make themselves appear as higher-privileged accounts.

## RESOLUTION

To prevent non-administrative accounts from accessing privileged information, the backend must be capable of blocking unauthorized access to these resources without compromising the access of administrative accounts.

This is achieved by hiding the admin functionality inside more complex code and by altering the privileged methods so that they can only be accessed if an authentic account's credentials are provided (userID + Token). This way, even if a regular user gains access to the admin website he will not be capable of executing the privileged methods. Without the correct credentials the API methods simply return errors without disclosing any sensitive information.



## CWE 256 PLAINTEXT STORAGE OF A PASSWORD

In the insecure app, no password processing is applied, neither in the backend nor in the frontend. As a result, passwords are stored in plain text, without any form of encryption or hashing. This practice is highly insecure and leaves the passwords readily accessible to anyone with access to the system, including potential attackers.

Storing passwords in plain text is a significant security risk as it allows malicious actors to easily read and misuse user credentials, compromising the confidentiality and integrity of user accounts.

```
// Create and save a new app_user object to the repository (database)
@PostMapping(path = "/add", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public @ResponseBody String addapp_user
    (@RequestParam String name, @RequestParam String email,
     @RequestParam String password, @RequestParam String cartao,
     @RequestParam String role, @RequestParam(required = false) MultipartFile img)
```

```

// Register the App_User Object
try {
    App_User usr = new App_User();
    usr.setName(name);
    usr.setEmail(email);

    usr.setPassword(password);

    usr.setCredit_Card(cartao);
    usr.setRole(role);

    . . .

    app_userRepository.save(usr);

    return "Saved";
} catch (Exception e) {
    e.printStackTrace();
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Internal
processing error!");
}
}

```

## RESOLUTION

As a last resource in case of information leaks (internal or external), the use of salted and hashed passwords is critical to maintain the security of the user's accounts. This ensures that if a malicious attacker manages to obtain an account's login information, he cannot simply use it for our and other applications, since it will first require heavy decoding and enormous processing power to obtain the original plaintext password.

To mitigate these risks, the password is now hashed using a SecureRandom number and salted with a random unique number for each account. The password is also never given back to the user at any API response, being replaced by a generated Token that we will discuss next.

```

Encoder encoder = Base64.getUrlEncoder().withoutPadding();
// Generate the random number
SecureRandom random = new SecureRandom();
// Generate the salt
byte[] salt = new byte[16];
random.nextBytes(salt);
// Generate the salted key
KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 65536, 128);
// Generate the final hashed + salted key
SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
byte[] hash = factory.generateSecret(spec).getEncoded();

```



# CWE 287 IMPROPER AUTHENTICATION

To properly utilize the application, the user must correctly prove that he is, in fact, the proper owner of the account he is trying to utilize and that he has access to the methods he is trying to execute.

In the insecure website, the user information is locally saved and most methods that require a complete user authentication (email + password) get the credential values from local storage.

All this means that if the user accesses a compromised website or someone gains access to the user's machine, they can access the local storage or cookie data and steal the user's credentials, which they can then change to lock the user out of his account and gain complete control to it.

Since users commonly use the same email and password for other sites, their other accounts are also compromised, and this could lead to extreme cases of theft.

```
user -> ▼ Array(1) ⓘ
  ▼ 0:
    credit_Card: "123123123123"
    email: "jose@fino.com"
    id: 2
    image: "src/assets/prod_images/11.jpg"
    name: "Josefino Calças"
    password: "Jose-Fino123"
    ▶ request_History: []
    role: "user"
    ▶ shopping_Cart: []
    ▶ [[Prototype]]: Object
    length: 1
    ▶ [[Prototype]]: Array(0)
```

## RESOLUTION

To ensure the user's identity is consistently verified throughout his shopping process, he must provide the credentials at every point of interaction with the system. However, asking for the credentials every time he navigates to a new page is inconceivable, and saving the password in local storage or cookies can easily allow them to be stolen by a simple scripting attack or if someone gets access to the user's computer.

To prevent this type of information leak, we switched to a Token-based user authentication, where the user receives a Token every time he logs in the application. The user should present the Token at every request or transaction and the Token should be completely regenerated at every login.

The Token is generated using a SecureRandom function and is then converted to a string type value.

If someone got access to the user's machine, the Token could be stolen but the user's login credentials would not, and at the next login that the user makes the attacker would lose access to the account, since he won't have an authorized Token.

The attacker can still access the user account, but now the risk of email and password leaks is much more reduced.

Further research into the usage of Token based authentication could potentially diminish even more the effectiveness of such attacks.

```
"name": "Josefino Calças",
"id": 2,
"active_Token": "830MVg0JXdINUdJ--10zEZCRmw85WXDbpqZiBDktla9wMMYTykS7zp2dh-qtbWe8zC1jJ7YCfr0rWEiV9Yko6Q",
"shopping_Cart": [],
"request_History": [],
"image": "src/assets/prod_images/11.jpg"
```

## CWE 201 INSERTION OF SENSITIVE INFORMATION INTO SENT DATA

This vulnerability occurs when sensitive data, such as passwords, financial information or personal details is inadvertently included in transmitted information. In our insecure app, the API returns the entire user object, exposing data like passwords and banking information with inadequate protection. This can lead to unintentional exposure of sensitive data to third parties, posing a significant threat to users' privacy and security.

When a user logs in, all their data is returned from the API:

```
userInfo -> ▼ Object { name: "Josefino Calças", id: 2, role: "user",
  credit_Card: "123123123123"
  email: "jose@fino.com"
  id: 2
  image: "src/assets/prod_images/11.jpg"
  name: "Josefino Calças"
  password: "123"
  ▶ request_History: Array []
  role: "user"
  ▶ shopping_Cart: Array []
  ▶ <prototype>: Object { ... }
```

## RESOLUTION

Sometimes sensitive information can be obtained by simple analysing the responses from the API server, since instead of manually parsing and building the response, the server can just automatically generate, for example, a JSON representation of an object without taking precautions like to stop sensitive information from being dispensed at the output.

To mitigate the possibility of leaking this sensitive information, the outputs of the API calls are now built manually with only the selected data values, so instead of returning the full user object, we return only a small portion of the required data that is essential for the operations of the system's frontend.

```
user -> ▼ Object {
  email: "jose@fino.com"
  id: "2"
  image: "src/assets/prod_images/11.jpg"
  name: "Josefino Calças"
  ▶ request_History: [{-}]
  ▶ shopping_Cart: []
  token: "UIfBdCWuHQxkloC57GxRz7uVQ1ackKIR5HNYsvt14Qrx3DmFB17cIER0EnIJ1gq_mgpAVpePd7tl_zFXXAwQww"
```



# FINAL OVERVIEW AND CONCLUSION

The development of this project has contributed to increase our awareness of the necessity to prioritize information security when creating websites, apps and services. It has demonstrated the consequences of neglecting security considerations by highlighting the vulnerabilities that can emerge when not taking this aspect seriously. We've learned that we cannot trust that all users will use the system as intended, and thus, security should be a top priority.

One of the key lessons we've gained is that code running inside the user's own machine, such as the frontend website, is exceptionally vulnerable to variable and method manipulation, whether by a malicious user or even an inexperienced regular user who makes mistakes. Web-based implementations are notoriously easy to alter locally. For instance, even relatively inexperienced users can access the "Inspect Element" function in most web browsers and modify the code or the values of variables stored in local storage.

As a result, we've realized that the frontend should be designed to be simple and user-friendly to prevent regular users from inadvertently introducing security risks. Simultaneously, the backend should be robust enough to thwart the manipulation of requests by bad actors and safeguard information from permanent damage or potential leaks.

We've concluded that security is not just a matter of implementing a few argument and input checks. It requires continuous development to prevent the abuse of unprotected methods and protect less informatically knowledgeable users.

Another vital insight is that security considerations should be integral to the application's development from the outset. Adding security features to an already existing service can be much more challenging than incorporating them from the beginning. This underscores the importance of coding defensively and with security in mind from the very start of a project.