

DETI STORE

Project 2 - SIO

Index

- [1. Introduction](#)
- [2. Overview](#)
- [3. Vulnerabilities](#)

1. Introduction

This project was developed for the "Informatics Security and Organizations" course as part of the Bachelor's degree in Informatics Engineering at the University of Aveiro.

2. Overview

The application is a simple online store that allows users to browse and purchase products. The application is divided into two main components: the client and the server. The client is a web application that allows users to browse and purchase products. The server is a Flask application that provides the client with the necessary data to display the products and allows users to purchase products.

The main objective of this project is to identify and mitigate vulnerabilities in the application. The vulnerabilities are divided into various categories following the ASVS standard.

A careful analysis of the application was performed in order to identify vulnerabilities. Out of the 280 ASVS requirements our system complied with 72 at first. After the analysis, we were able to mitigate 24 vulnerabilities, resulting in a total of 96 ASVS requirements complied. Almost all of the level 1 requirements were met, and some level 2 requirements were also met.

The vulnerabilities were mitigated by using the following techniques:

- Communications Security Requirements (V9.1.1)
 - Use HTTPS to encrypt the data transmitted between the client and the server.
- File Upload Requirements (V12.1.2)
 - Ensure that the uploaded files are not too large.
- Session Logout and Timeout Requirements (V3.3.2)
 - Terminate the user's session when the user logs out or when the session expires.
- Error Handling (V7.4.1)
 - Display a generic error message to the user.
- Log Content Requirements (V7.1.1) & (V7.1.2)
 - Remove all the logs that contain information about the application, user's credentials, payment details, etc.
- Dependency (V14.2.1)
 - Use a dependency checker to ensure that all the dependencies are up to date and that there are no vulnerabilities in the dependencies.

- Password Security Credentials (V2.1.1) & (V2.1.2) & (V2.1.3) & (V2.1.4) & (V2.1.9)
 - Enforce a password policy that ensures that the passwords are secure.
- Password Security Credentials (V2.1.7)
 - Check if the password is breached.
- Password Security Credentials (V2.1.8) & (V2.1.12)
 - Display the strength of the password to the user.
 - Allow the user to view the password.
- General Authenticator Requirements (V2.2.1)
 - Prevent automated attacks by checking if the user is a human or a robot, used google recaptcha.
- General Authenticator Requirements (V2.2.2)
 - Allow users to log in using their Google credentials.
- General Authenticator Requirements (V2.2.3)
 - Ensure a secure and reliable communication channel for critical account activities.
- Credential Recovery Requirements (V2.5.5)
 - Notify the user when an authentication factor is changed or replaced.
- Out of Band Verifier Requirements (V2.7.2)
 - Verify the user's identity by sending a verification code to the user's email.
- Out of Band Verifier Requirements (V2.7.3)
 - Verification code is valid for a single use.
- Data Classification (V6.1.1) & (V6.1.3)
 - Encrypt sensitive data.
- Algorithms (V6.2.1)
 - Use strong encryption algorithms.

We were asked to only solve 8 vulnerabilities, but we decided to solve more than 8 in order to improve the overall security of the application and to learn more about security. Nonetheless, we consider that the following vulnerabilities were the most important to solve:

- Communications Security Requirements (V9.1.1)
- Data Classification (V6.1.1) & (V6.1.3)
- Algorithms (V6.2.1)
- General Authenticator Requirements (V2.2.1)
- Password Security Credentials (V2.1.1) & (V2.1.2) & (V2.1.3) & (V2.1.4) & (V2.1.9)

In the following sections, we will describe the vulnerabilities that were identified and the techniques that were used to mitigate them.

3. Vulnerabilities Solved

3.1 Communications Security Requirements (V9.1.1)

"Verify that secured TLS is used for all client connectivity, and does not fall back to insecure or unencrypted protocols." ([C8](#))

Nowadays, it's almost mandatory to have a secure connection between the client and the server. This is done by using HTTPS to encrypt the data transmitted between the client and the server. This ensures that the data is not intercepted by third parties and that the data is not modified during transmission.

HTTPS is a protocol that uses TLS/SSL to encrypt the data transmitted between the client and the server. TLS/SSL uses asymmetric cryptography to establish a secure connection between the client and the server. This is done by using a public key and a private key. The public key is used to encrypt the data and the private key is used to decrypt the data. This ensures that only the server can decrypt the data transmitted by the client and vice versa.

Our application uses HTTPS to communicate with the client. This is done by using a load balancer that redirects all HTTP requests to HTTPS. This is done by using the following nginx configuration:

```
upstream web {
    server web:5000;
}

server {
    listen 80;
    server_name localhost;
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl;
    server_name localhost;

    ssl_certificate cert.pem;
    ssl_certificate_key key.pem;

    location / {
        proxy_pass https://web;
        proxy_set_header Host "localhost";
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

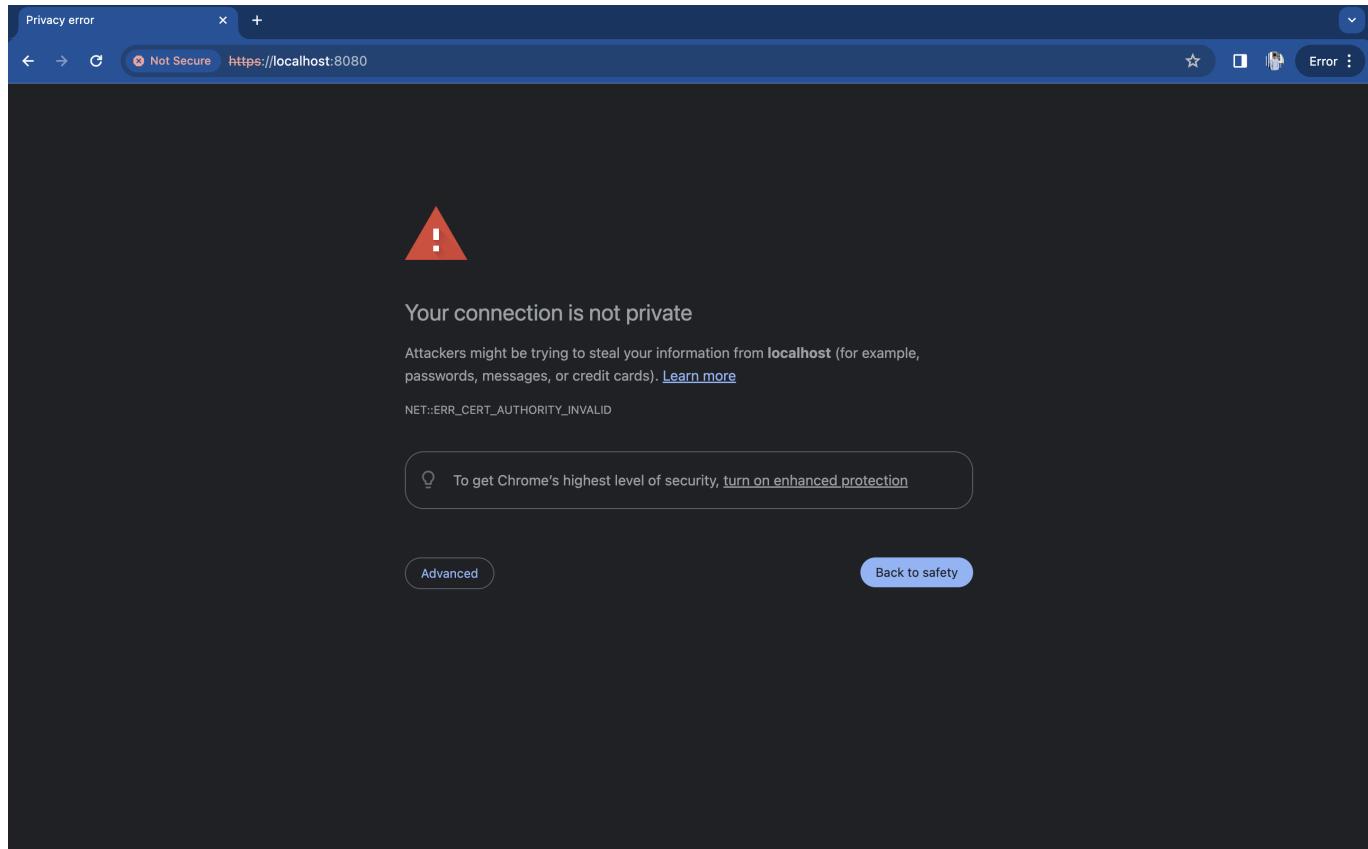
In our Flask application we redirect all HTTP requests to HTTPS we use the following code:

```
@app.before_request
def before_request():
    if (
        not request.is_secure
        and request.headers.get("X-Forwarded-Proto") != "https"
    ):
        url = request.url.replace("http://", "https://", 1)
        code = 301
        return redirect(url, code=code)
```

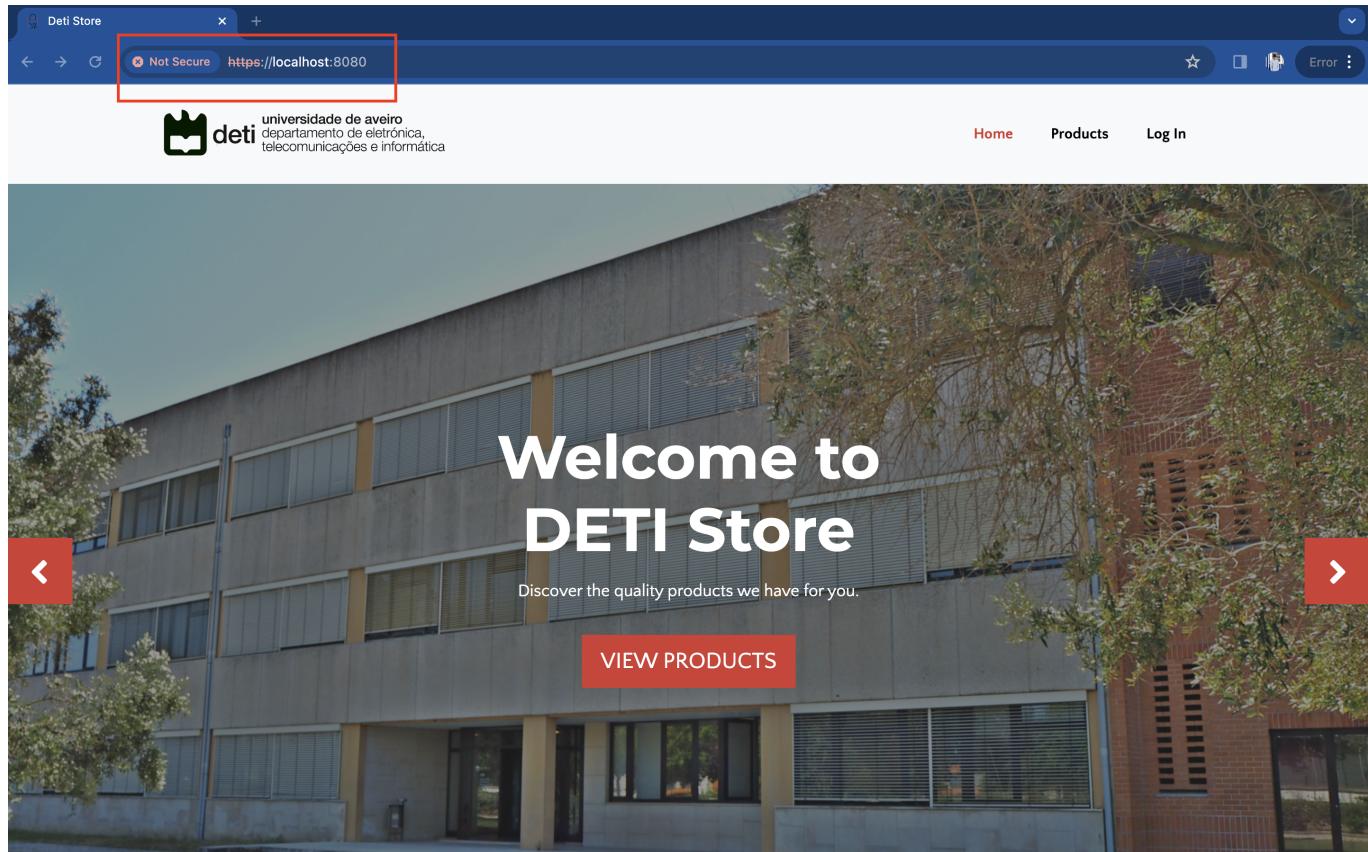
With this code, we ensure that all requests made to our Flask application are transmitted securely via HTTPS, enhancing the overall security of the communication channel and protecting sensitive user data.

Note:

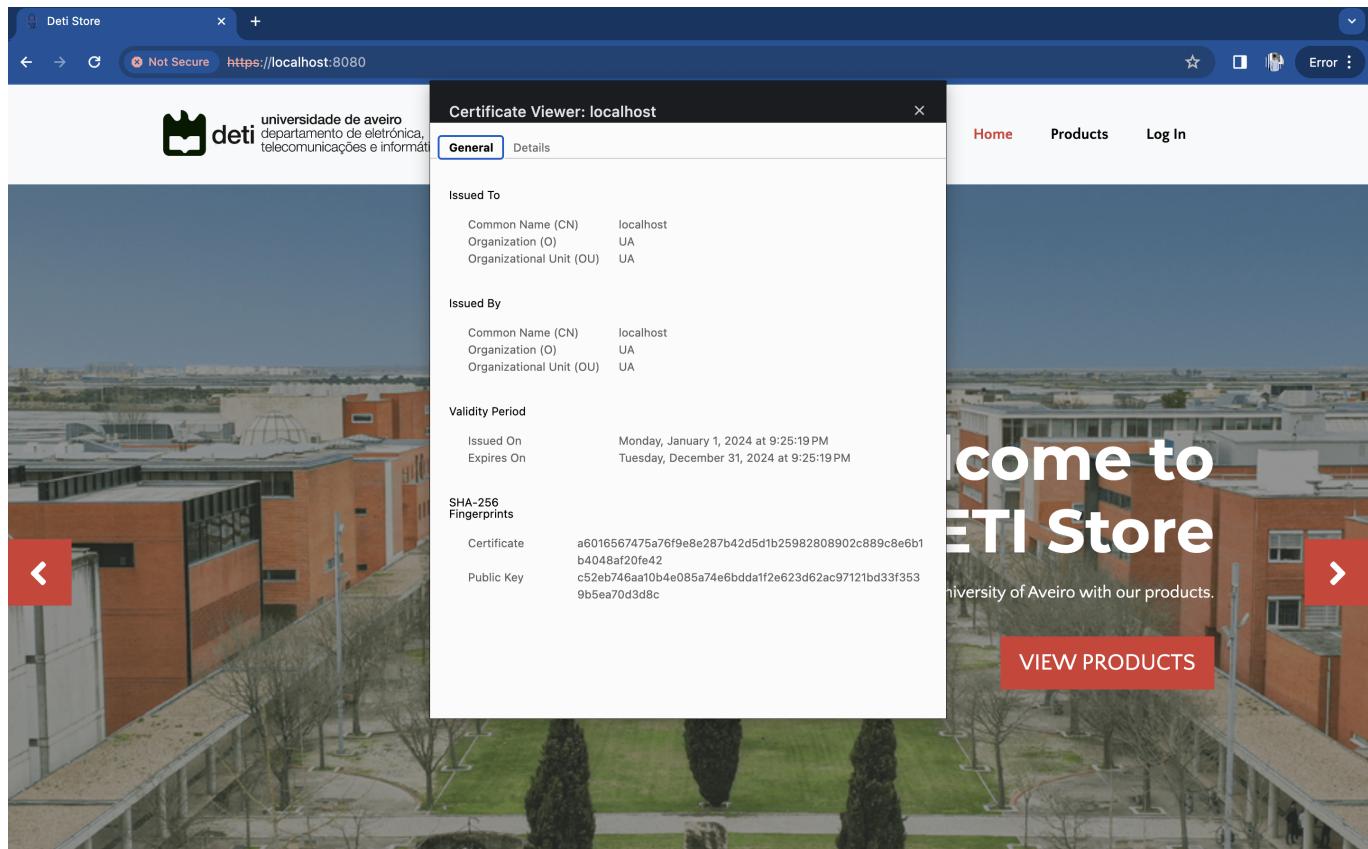
While a self-signed certificate suffices for testing purposes, it is crucial to obtain a valid SSL/TLS certificate from a trusted certificate authority for production environments. This not only ensures the security of user data but also establishes trust among users by displaying the padlock symbol in the browser's address bar.



As we can see in the image above, the browser warns the user that the connection is not secure. This is because the certificate is self-signed and not issued by a trusted certificate authority. This is not a problem for testing purposes, but it is a problem for production environments.



After clicking on "Advanced" and proceeding to the website, we can see that we are connected via HTTPS. With this type of connection, the data is encrypted and transmitted securely.



We can also see that the certificate is self-signed and not issued by a trusted certificate authority. A certificate contains information about the issuer, the validity period, the domain name, etc. In this case, the certificate is issued by the domain name "localhost" and is valid for 365 days.

To obtain this certificate we used the following command:

```
openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout key.pem -  
days 365 -subj "/C=PT/ST=Aveiro/L=Aveiro/O=UA/OU=UA/CN=localhost"
```

openssl is a command-line tool that can be used to generate certificates. In this case, we used the req command to generate a self-signed certificate. The -x509 option specifies that we want to generate a self-signed certificate. The -newkey option specifies that we want to generate a new key. The rsa:4096 option specifies that we want to generate a 4096-bit RSA key. The -nodes option specifies that we do not want to encrypt the private key. The -out option specifies the output file. The -keyout option specifies the private key file. The -days option specifies the number of days the certificate is valid. The -subj option specifies the subject of the certificate. The subject contains information about the issuer, the validity period, the domain name, etc. In this case, the certificate is issued by the domain name "localhost" and is valid for 365 days.

3.2 File Upload Requirements (V12.1.2)

"Verify that the application will not accept large files that could fill up storage or cause a denial of service."

File upload is a common feature in web applications. However, it is also a common source of vulnerabilities. This is because the uploaded files can contain malicious code that can be executed on the server. To prevent this, it is necessary to validate the uploaded files and ensure that they do not contain malicious code.

Besides validating the uploaded files, it is also necessary to ensure that the uploaded files are not too large. This is because large files can cause a denial of service attack by consuming all the server's resources.

In our application, we allow users to upload an image when creating an account. This image is then displayed on the user's profile page. To prevent malicious code from being uploaded, we validate the uploaded file and ensure that it is an image while also ensuring that the image is not too large. This is done by using the following code:

```
if profile_picture:  
    if (  
        profile_picture.filename.endswith(".png")  
        or profile_picture.filename.endswith(".jpeg")  
        or profile_picture.filename.endswith(".jpg")  
    ):  
        try:  
            upload_folder = "static/images/profile_pictures"  
            file_name = email + "_" + profile_picture.filename  
            os.makedirs(upload_folder, exist_ok=True)  
  
            # Save the file to the directory  
            profile_picture.save(os.path.join(upload_folder,  
                file_name))  
  
            # check if picture is bigger than 5MB  
            if os.path.getsize(upload_folder + "/" + file_name) /
```

```
(1024 * 1024) > 5:
    # remove the file
    os.remove(upload_folder + "/" + file_name)
    flash("A imagem não pode ter mais de 5MB!", category="danger")
    return redirect(url_for("register.regist"))

    # encrypt email and phone number
    email_key = E.generate_key()
    phone_key = E.generate_key()
    # store the keys
    E.store_key(email_key, f"{user.upper()}_EMAIL_KEY")
    E.store_key(phone_key, f"{user.upper()}_PHONE_KEY")
    email_enc = E.chacha20_encrypt(email, email_key)
    phone_enc = E.chacha20_encrypt(phone, phone_key)
    if email_enc is None or phone_enc is None:
        flash("Erro ao encriptar email ou número de telefone!")
    return redirect(url_for("register.regist"))

    new_user = User(
        username=user,
        password=generate_password_hash(key),
        name=nome,
        email=email_enc,
        phone=phone_enc,
        image=upload_folder + "/" + file_name,
        security_question=security_question,
        google_account=False,
    )
except Exception as e:
    print(e)
    flash("Erro ao fazer upload da imagem!", category="danger")
    return redirect(url_for("register.regist"))
else:
    flash(
        "Por favor insira uma imagem com extensão .png ou .jpeg ou .jpg",
        category="danger",
    )
return redirect(url_for("register.regist"))
```

First we make sure that the uploaded file is an image by checking the file extension. Then we check if the image is too large. If the image is too large, we remove the image and display an error message. If the image is not too large, we encrypt the user's email and phone number and store the encryption keys. By setting a threshold of 5MB, we ensure that the image is not too large and that it does not consume too much space on the server.

In case the user tries to upload an image bigger than 5MB, the following error message is displayed:

The screenshot shows a web page with a header containing the logo of the University of Aveiro's Department of Electronics, Telecommunications and Informatics (DETI) and navigation links for 'Home' and 'Products'. Below the header is a 'New User Registration' form. The form includes fields for Name, Email, Phone Number, Username, and Password. A note at the top states 'Imagen muito grande! Maximo 5MB!' (Image too large! Maximum 5MB!). On the right side of the form, there is a link labeled 'Privacy - Terms' next to a small circular icon.

3.3 Session Logout and Timeout Requirements (V3.3.2)

"If authenticators permit users to remain logged in, verify that re-authentication occurs periodically both when actively used or after an idle period" ([C6](#)).

Session management is a crucial part of any web application. It is important to ensure that the user's session is terminated when the user logs out or when the session expires. This is done by checking if the user is logged in and if the session has expired. If the user is logged in and the session has not expired, the user will be asked to login again, thus ensuring that the user's session is terminated.

This type of protection is important because it prevents unauthorized users from accessing the user's account.

For this in added a parameter to the User table in the database called "last_activity_time" that stores the last time the user logged in. This parameter is updated every time the user logs in. This is done by using the following code:

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    password = db.Column(db.String(100), nullable=True)
    isAdmin = db.Column(db.Boolean, default=False)
    username = db.Column(db.String(100), unique=True)
    name = db.Column(db.String(100))
    email = db.Column(db.String(100), unique=True)
    phone = db.Column(db.String(100), nullable=True)
    image = db.Column(
        db.String(20), nullable=False,
        default="../static/images/default.jpg"
    )
    security_question = db.Column(db.String(100), nullable=True)
    cart = db.relationship("Cart", backref="user")
```

```
wishlist = db.relationship("Wishlist", backref="user")
google_account = db.Column(db.Boolean, default=False)
last_activity_time = db.Column(db.DateTime, default=datetime.utcnow) # here
verification_code = db.Column(db.String(100), nullable=True,
default=None)
verification_timestamp = db.Column(db.Integer, default=0)
```

Then, when the user tries to access a page, we check if the user is logged in and if the session has expired. This is done by using the following code:

```
@login_required
def recheck_login():
    # Check if re-authentication is needed based on the configured periods
    idle_period_limit = datetime.utcnow() - timedelta(days=30) # L1: 30
    days
    actively_used_limit = datetime.utcnow() - timedelta(hours=12) # L2:
    12 hours

    if current_user.last_activity_time < idle_period_limit:
        # Re-authenticate for idle period
        flash("Please re-enter your password to continue.", "danger")

        logout_user()

        return redirect(url_for("auth.login"))

    if current_user.last_activity_time < actively_used_limit:
        # Re-authenticate for actively used period
        flash("Please re-enter your password to continue.", "danger")

        logout_user()

        return redirect(url_for("auth.login"))
```

If the user is logged in and the session has not expired, the user will be asked to login again and this message is shown:

Login

Please re-enter your password to continue.

Username:

Password:

Dont have an account? [Join Us!](#)

[Login](#)



[Login with Google](#)

3.4 Error Handling (V7.4.1)

"Verify that a generic message is shown when an unexpected or security sensitive error occurs, potentially with a unique ID which support personnel can use to investigate." ([C10](#))

Error handling is an important part of any application. It is important to ensure that the user is informed of any errors that occur. This is done by displaying an error message to the user. This error message should be generic and should not contain any sensitive information. This is because the error message can be used by attackers to gain information about the application.

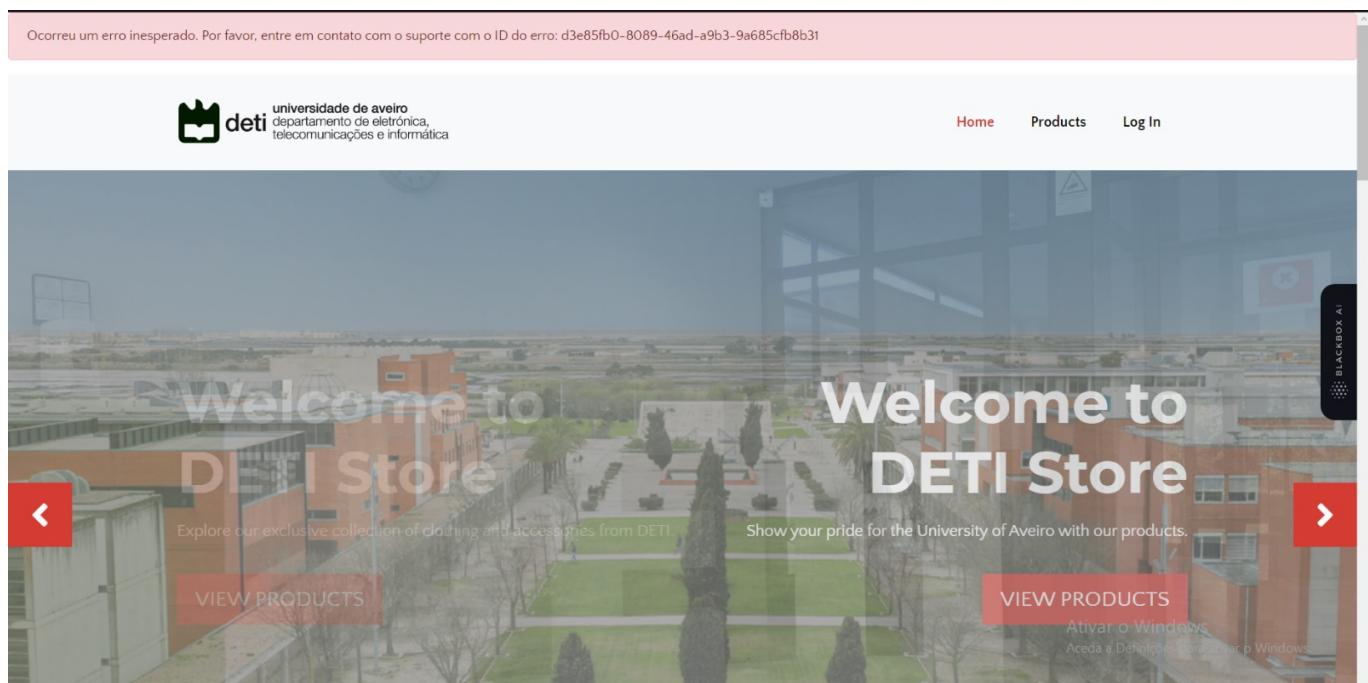
For this we used logging from the python library. This allows us to log any errors that occur in the application. This is done by using the following code:

```
def handle_error(e):
    error_id = generate_unique_error_id()
    # check if datetime has attribute utcnow
    if hasattr(datetime, "utcnow"):
        timestamp = datetime.utcnow().isoformat()
    else:
        timestamp = datetime.datetime.now().isoformat()
    user_info = (
        f"User: {current_user.username}"
        if current_user.is_authenticated
        else "User: Not authenticated"
    )
    logger.error(
        "Error ID: %s\nTimestamp: %s\n%s\n%s", error_id, timestamp,
        user_info, str(e)
    )

    flash(
        "Ocorreu um erro inesperado. Por favor, entre em contato com o
        suporte com o ID do erro: "
        + error_id,
        category="danger",
    )
    return render_template("index.html")

def generate_unique_error_id():
    return str(uuid.uuid4())
```

This allows us to log any errors that occur in the application and display a generic error message to the user. An id is attributed to each error so that the support team can identify the error and fix it.



As we can see in the image above, the error message is generic and does not contain any sensitive information. This is because the error message can be used by attackers to gain information about the application.

3.5 & 3.6 Log Content Requirements (V7.1.1) & (V7.1.2)

"Verify that the application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form." ([C9](#), [C10](#))

"Verify that the application does not log other sensitive data as defined under local privacy laws or relevant security policy." ([C9](#))

Logging is an important part of any application. It is important to ensure that the application logs all the necessary information. However, it is also important to ensure that the application does not log sensitive information. This is because the log files can be accessed by attackers and used to gain information about the application.

To prevent this, we ensure that the application does not log sensitive information by removing all the logs that contain information about the application, user's credentials, payment details, etc.

This way attackers cannot gain information about the application by accessing the log files.

3.7 Dependency (V14.2.1)

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

Dependencies are an important part of any application. However, it is important to ensure that the dependencies are up to date. This is because outdated dependencies can contain vulnerabilities that can be exploited by attackers.

To ensure that we resort to a dependency checker called "safety" and a dependency "pip-check".

This dependencies were added to our requirements.txt which is the used by the dockerfile to install the dependencies.

```
FROM python:3.11
WORKDIR /app
ENV FLASK_APP=__init__.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apt update -y
RUN apt install gcc musl-dev wkhtmltopdf -y
COPY requirements.txt requirements.txt
RUN python -m venv venv
RUN venv/bin/pip install --upgrade pip && venv/bin/pip install -r requirements.txt
RUN pip install --upgrade pip && pip install -r requirements.txt
RUN pip-check
RUN safety check
EXPOSE 5000
COPY . .
```

```
CMD ["bash", "-c", "\n    export FLASK_APP=__init__.py && \n    export FLASK_DEBUG=1 && \n    flask run --cert=cert.pem --key=key.pem"]
```

This dependencies ensure that all the dependencies are up to date and that there are no vulnerabilities in the dependencies.

3.8 Password Security Credentials (V2.1.1) & (V2.1.2) & (V2.1.3) & (V2.1.4) & (V2.1.9)

"Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined). (C6)"

"Verify that passwords 64 characters or longer are permitted but may be no longer than 128 characters. (C6)"

"Verify that password truncation is not performed. However, consecutive multiple spaces may be replaced by a single space. (C6)"

"Verify that any printable Unicode character, including language neutral characters such as spaces and Emojis are permitted in passwords."

"Verify that there are no password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. (C6)"

Passwords are an important part of any application. It is important to ensure that the passwords are secure and that they cannot be easily guessed by attackers. This is done by enforcing a password policy that ensures that the passwords are secure.

In our application, we enforce a password policy that ensures that the passwords are secure. This is done by using the following code:

```
def is_valid_password(password):
    # Check if the password is breached
    if check_breached_password(password):
        flash("A senha foi comprometida, tente outra")
        return False
    # Ensure password length is within the allowed range
    elif len(password) < 12:
        flash("A senha deve ter pelo menos 12 caracteres (espaços não incluídos)")
        return False
    elif len(password) > 128:
        flash("A senha deve ter no máximo 128 caracteres (espaços não incluídos)")
        return False
    # Check if all characters in the password are printable Unicode characters
    elif not all(c.isprintable() for c in password):
```

```
flash("A senha deve conter apenas caracteres Unicode imprimíveis")
return False

return True
```

This code ensures that the passwords are secure and that they cannot be easily guessed by attackers. This is done by enforcing a password policy that ensures that the passwords are secure.

New User Registration

A senha deve ter pelo menos 12 caracteres (espaços não incluídos)

Name:

ZeMendes

Email:

zepedrosmendes@gmail.com

Phone Number:

912345678

Username:

zemendes17

Password:

In the image above we try to register a user with a password that is shorter than 12 characters, showing an error message to the user, as expected.

New User Registration

A senha deve ter no máximo 128 caracteres (espaços não incluídos)

Name:

ZeMendes

Email:

zepedrosmendes@gmail.com

Phone Number:

912345678

Username:

zemendes17

In the image above we try to register a user with a password that is longer than 128 characters, showing an error message to the user, as expected.

In order to ensure that consecutive multiple spaces are replaced by a single space, we use the following code:

```
processed_key = re.sub(" +", " ", key)
if not is_valid_password(processed_key):
    return redirect(url_for("register.register"))
```

Using the re library, we replace all consecutive multiple spaces by a single space with the sub function. This is done using the password that the user inputs, validating it only after the replacement.

To ensure that any printable Unicode character, including language neutral characters such as spaces and Emojis are permitted in passwords, we use the following code:

```
elif not all(c.isprintable() for c in password):
    flash("A senha deve conter apenas caracteres Unicode imprimíveis")
    return False
```

Using the `isprintable` function, we ensure that any printable Unicode character is permitted in passwords. In case the password contains a non-printable Unicode character, an error message is displayed to the user.

Finally, no password composition rules limiting the type of characters permitted are enforced, as seen in the function `is_valid_password` located above.

3.9 Password Security Credentials (V2.1.7)

"Verify that passwords submitted during account registration, login, and password change are checked against a set of breached passwords either locally (such as the top 1,000 or 10,000 most common passwords which match the system's password policy) or using an external API. If using an API a zero knowledge proof or other mechanism should be used to ensure that the plain text password is not sent or used in verifying the breach status of the password. If the password is breached, the application must require the user to set a new non-breached password. (C6)"

```
def check_breached_password(password):
    # Hash the password using SHA-1
    sha1_hash = hashlib.sha1(password.encode("utf-8")).hexdigest().upper()

    # Send the first 5 characters of the hashed password to the HIBP API
    api_url = f"https://api.pwnedpasswords.com/range/{sha1_hash[:5]}"
    response = requests.get(api_url)

    if response.status_code == 200:
        # Check if the remaining part of the hashed password appears in
        # the response
        tail = sha1_hash[5:]
        if tail in response.text:
            return True # Password is breached
    return False # Password is not breached
```

We use the HIBP API to check if the password is breached. This is done by sending the first 5 characters of the hashed password to the HIBP API. If the remaining part of the hashed password appears in the response, the password is breached. If the password is breached, the user is asked to set a new password.

New User Registration

A senha foi comprometida, tente outra

Name:

ZeMendes

Email:

zepedrosmendes@gmail.com

Phone Number:

912345678

Username:

zemendes17

In the image above we try to register a user with a password that is breached (in this example "123456789"), showing an error message to the user, as expected.

3.9 Password Security Credentials (V2.1.8) & (V2.1.12)

"Verify that a password strength meter is provided to help users set a stronger password."

"Verify that the user can choose to either temporarily view the entire masked password, or temporarily view the last typed character of the password on platforms that do not have this as built-in functionality."

We use the zxcvbn library to check the strength of the password. This library returns a score from 0 to 4. The higher the score, the stronger the password. We use this score to display the strength of the password to the user.

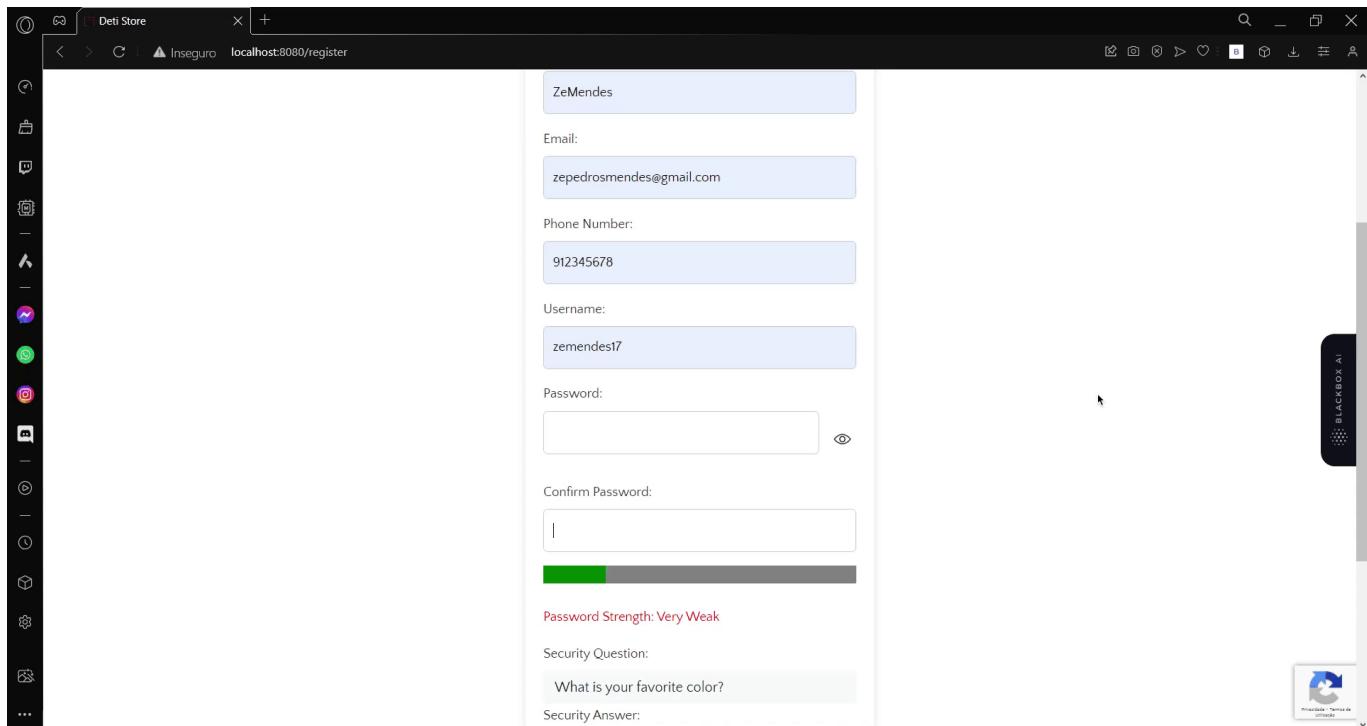
```
<label for="password">Password:</label>
<div id="togglePasswordContainer" style="display: flex; align-items: center;">
```

```
<input  
    type="password"  
    name="password"  
    id="password"  
    oninput="checkPasswordStrength()"  
    required  
/>  
<button  
    type="button"  
    id="togglePasswordButton"  
    onclick="togglePasswordVisibility()"  
    style="margin-left: 10px; background-color: white; border: none;  
display: flex; align-items: center;"  
>  
      
</button>  
</div>  
  
<label for="confirm_password">Confirm Password:</label>  
<input type="password" name="confirm_password" /><br />  
<progress id="password-strength-meter" max="5" value="0"></progress>  
<p id="password-strength-text">Password Strength: Very Weak</p>
```

```
function checkPasswordStrength() {  
    var password = document.getElementById("password").value;  
    var result = zxcvbn(password);  
  
    // Update the password strength meter  
    var strengthMeter = document.getElementById("password-strength-meter");  
    strengthMeter.value = result.score + 1;  
  
    // Update the text based on the password strength  
    var strengthText = document.getElementById("password-strength-text");  
    strengthText.innerHTML =  
        "Password Strength: " + getStrengthText(result.score);  
  
    // Adicione a lógica para alterar a cor do texto com base na força da  
    // senha  
    switch (result.score) {  
        case 0:  
            strengthText.style.color = "red"; // Very Weak  
            strengthMeter.style.background = "red";  
            break;  
        case 1:  
            strengthText.style.color = "orange"; // Weak
```

```
        strengthMeter.style.background = "orange";
        break;
    case 2:
        strengthText.style.color = "yellow"; // Moderate
        strengthMeter.style.background = "yellow";
        break;
    case 3:
        strengthText.style.color = "green"; // Strong
        strengthMeter.style.background = "green";
        break;
    case 4:
        strengthText.style.color = "darkgreen"; // Very Strong
        strengthMeter.style.background = "darkgreen";
        break;
    default:
        strengthText.style.color = "#333"; // Cor padrão
        strengthMeter.style.background = "#eee";
    }
}

function getStrengthText(score) {
    switch (score) {
        case 0:
            return "Very Weak";
        case 1:
            return "Weak";
        case 2:
            return "Moderate";
        case 3:
            return "Strong";
        case 4:
            return "Very Strong";
        default:
            return "";
    }
}
```



As we can see above, the password strength is displayed to the user while we types the password. This allows the user to choose a stronger password.

We also allow the user to view the password by clicking on the eye icon. This is done by using the following code:

```
<button
  type="button"
  id="togglePasswordButton"
  onclick="togglePasswordVisibility()"
  style="margin-left: 10px; background-color: white; border: none;
display: flex; align-items: center;">
  
</button>
```

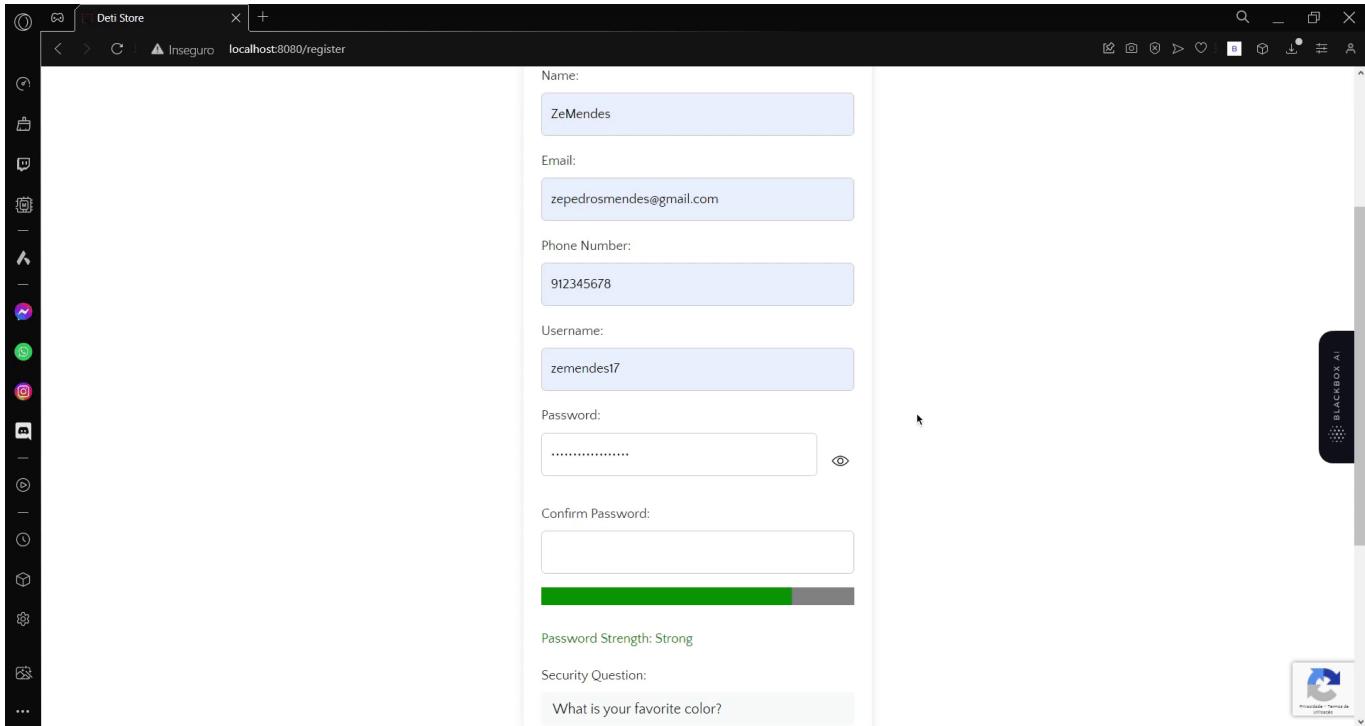
```
function togglePasswordVisibility() {
  var passwordInput = document.getElementById("password");
  var eyeIcon = document.getElementById("eyeIcon");

  if (passwordInput.type === "password") {
    passwordInput.type = "text";
    eyeIcon.src = "./static/images/eye-off-icon.png";
  } else {
```

```

    passwordInput.type = "password";
    eyeIcon.src = "./static/images/eye-on-icon.png";
}
}

```



3.10 General Authenticator Requirements (V2.2.1)

"Verify that anti-automation controls are effective at mitigating breached credential testing, brute force, and account lockout attacks. Such controls include blocking the most common breached passwords, soft lockouts, rate limiting, CAPTCHA, ever increasing delays between attempts, IP address restrictions, or risk-based restrictions such as location, first login on a device, recent attempts to unlock the account, or similar. Verify that no more than 100 failed attempts per hour is possible on a single account."

We used google recaptcha to prevent automated attacks. This is done by using the following code:

```

<button
  class="g-recaptcha"
  data-sitekey="6LeFQDkpAAAAABKdp4pinNykhov9pQeL493lwh1_"
  data-callback="onSubmit"
  data-action="submit"
  style="width: 100%; background-color:#162442; color:white">
  Sign In
</button>

```

```

function onSubmit(token) {
  document.getElementById("form").submit();
}

```

```

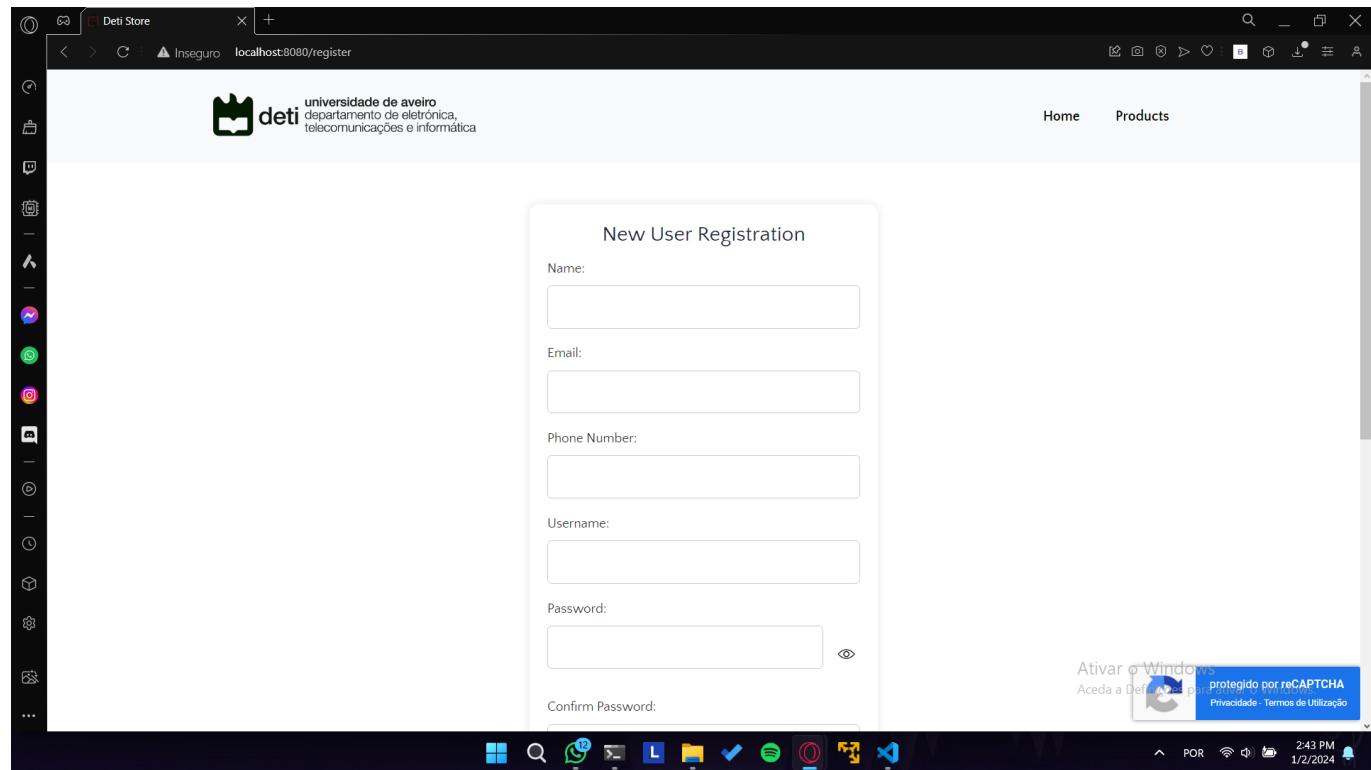
recaptcha_response = request.form["g-recaptcha-response"]

recaptcha_request = requests.post(
    "https://recaptchaenterprise.googleapis.com/v1/projects/deti-store-
1703363018508/assessments?key=AIzaSyDHxOKFmFzw4ijJ-pUmTDRFLvrnJt0xzw",
    json={
        "event": {
            "token": recaptcha_response,
            "expectedAction": "register",
            "siteKey": "6LeFQDkpAAAAABKdp4pinNykhov9pQeL493lwh1_",
        }
    },
    headers={"Content-Type": "application/json"},
).json()

if not recaptcha_request["tokenProperties"]["valid"]:
    flash("Recaptcha inválido!", category="danger")
    return redirect(url_for("auth.login"))

```

This prevents automated attacks by checking if the user is a human or a robot. If the user is a human, the user is allowed to login. If the user is a robot, the user is not allowed to login.



3.11 General Authenticator Requirements (V2.2.2)

"Verify that the use of weak authenticators (such as SMS and email) is limited to secondary verification and transaction approval and not as a replacement for more secure authentication methods. Verify that stronger methods are offered before weak methods, users are aware of the risks, or that proper measures are in place to limit the risks of account compromise."

On our website, we provide a user-friendly registration process utilizing email, where users are required to input a valid email address. During the login phase, an additional layer of security is implemented through the introduction of a verification code, enhancing the overall account protection. Moreover, recognizing the importance of convenience and accessibility, we also offer a seamless alternative by enabling users to log in using their Google credentials. This feature is powered by OAuth 2.0 and OIDC (OpenID Connect) login protocols, ensuring a secure and efficient authentication process for those who prefer the simplicity of connecting through their Google accounts.

Login

Username:

Password:

Dont have an account? [Join Us!](#)

Login

 [Login with Google](#)

When users log in using their Google credentials, they are redirected to the Google login page, where they are asked to login using their Google credentials.

3.12 General Authenticator Requirements (V2.2.3)

"Verify that secure notifications are sent to users after updates to authentication details, such as credential resets, email or address changes, logging in from unknown or risky locations. The use of push notifications - rather than SMS or email - is preferred, but in the absence of push notifications, SMS or email is acceptable as long as no sensitive information is disclosed in the notification."

Users will exclusively receive their login verification codes and confirmations for profile data changes via email, ensuring a secure and reliable communication channel for these critical account activities. This is

done by using the following code:

```
def send_otp_via_email(otp_code, email):
    try:
        email_server = current_app.config["EMAIL_SERVER"]

        subject = "Verification Code"
        body = f"Your verification code is: {otp_code}"

        message = f"From: detiStore@outlook.com\r\nTo: {email}\r\nSubject: {subject}\r\n\r\n{body}"

        # Sending email
        email_server.sendmail("dettiStore@outlook.com", [email], message)
    except Exception as e:
        # Handle unexpected errors
        return handle_error(e)
```

```
def send_email_notification(email):
    try:
        email_server = current_app.config["EMAIL_SERVER"]

        subject = "DetiStore – Dados da sua conta foram alterados"
        body = "Dados da sua conta foram alterados. Se não foi você, por favor entre em contato com o suporte."

        message = f"From: detiStore@outlook.com\r\nTo: {email}\r\nSubject: {subject}\r\n\r\n{body}"

        # Sending email
        email_server.sendmail("dettiStore@outlook.com", [email],
message.encode("utf-8"))
    except Exception as e:
        # Handle unexpected errors
        return handle_error(e)
```

This ensures that the user receives the verification code and the confirmation for profile data changes via email, ensuring a secure and reliable communication channel for these critical account activities.

Enter Verification Code

Code:

Submit

detiStore@outlook.com através de sendinblue.com
para mim ▾

domingo, 31/12/2023, 14:27 (há 2 dias)

☆ ☺ ↵ :

Your verification code is: 676494

Responder

Encaminhar



3.13 Credential Recovery Requirements (V2.5.5)

"Verify that if an authentication factor is changed or replaced, that the user is notified of this event."

In our application, we allow users to change all their profile data. They are notified of this event by email. This is done by using the following code:

```
def send_email_notification(email):
    try:
        email_server = current_app.config["EMAIL_SERVER"]

        subject = "DetiStore – Dados da sua conta foram alterados"
        body = "Dados da sua conta foram alterados. Se não foi você, por favor entre em contato com o suporte."

        message = f"From: detiStore@outlook.com\r\nTo: {email}\r\nSubject: {subject}\r\n\r\n{body}"

        # Sending email
        email_server.sendmail("detiStore@outlook.com", [email],
message.encode("utf-8"))
    except Exception as e:
        # Handle unexpected errors
        return handle_error(e)
```



detiStore@outlook.com

DetiStore - Dados da sua conta foram alterados

Para: André Oliveira

Recebido - a...a.net1@gmail.com

16:16

Dados da sua conta foram alterados. Se não foi você, por favor entre em contato com o suporte.

3.14 Out of Band Verifier Requirements (V2.7.2)

"Verify that the out of band verifier expires out of band authentication requests, codes, or tokens after 10 minutes."

In our application, the verification code expires after 10 minutes. This is done by using the following code:

```
verification_code = request.form["otp"]

user = User.query.filter_by(username=username).first()

if user.verification_code == verification_code:
    if (
        user.verification_timestamp
        and (int(time.time()) - user.verification_timestamp) < 600
    ):
        login_user(user)
        return redirect(url_for("main.index"))
else:
    flash("Invalid Verification Code", category="danger")
    return redirect(url_for("auth.login"))
```

Login

Invalid Verification Code

Username:

Password:

Dont have an account? [Join Us!](#)

[Login](#)



[Login with Google](#)

3.15 Out of Band Verifier Requirements (V2.7.3)

"Verify that the out of band verifier authentication requests, codes, or tokens are only usable once, and only for the original authentication request."

In our application, the verification code can only be used once. This is done by using the following code:

```
verification_code = totp.now()

try:
    # Save the verification code and timestamp in the database
    user.verification_code = verification_code
    user.verification_timestamp = int(time.time())
    db.session.commit()
    # Send the verification code via email
    send_otp_via_email(verification_code, email)
```

```

    return render_template("enter_otp.html", username=username)
except Exception as e:
    flash("Erro ao guardar código de verificação!", category="danger")
    return redirect(url_for("auth.login"))

```

3.16 Data Classification (V6.1.1) & (V6.1.3)

"Verify that regulated private data is stored encrypted while at rest, such as Personally Identifiable Information (PII), sensitive personal information, or data assessed likely to be subject to EU's GDPR."

"Verify that regulated financial data is stored encrypted while at rest, such as financial accounts, defaults or credit history, tax records, pay history, beneficiaries, or de-anonymized market or research records."

In order to perform the encryption and decryption of the user's PII (Personally Identifiable Information), we used the cryptography library. This library allows us to encrypt and decrypt the user's PII using the ChaCha20 algorithm. This algorithm is a symmetric stream cipher that uses a 32-byte key and a 16-byte nonce. This algorithm is secure, fast and does not require padding, making it ideal for encrypting and decrypting the user's PII.

```

def chacha20_encrypt(message, key):
    iv = os.urandom(16)

    try:
        cipher = Cipher(algorithms.ChaCha20(key, iv), mode=None,
backend=default_backend())
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(message.encode()) +
encryptor.finalize()
        return iv + ciphertext
    except Exception as e:
        print("Encryption error:", e)
        return None

def chacha20_decrypt(ciphertext, key):
    iv = ciphertext[:16]

    try:
        cipher = Cipher(algorithms.ChaCha20(key, iv), mode=None,
backend=default_backend())
        decryptor = cipher.decryptor()
        plaintext = decryptor.update(ciphertext[16:]) +
decryptor.finalize()
        return plaintext.decode()
    except Exception as e:
        print("Decryption error:", e)
        return None

```

The keys are being stored in a configuration file (.env), ensuring that they are not hardcoded in the source code, or in the database. Consequently, to store and retrieve the keys, the following functions were

implemented:

```
def store_key(key, name):
    os.environ[name] = key.hex()
    # if there is no .env file, create one
    if not os.path.exists('.env'):
        open('.env', 'w').close()
    # if the key is already in the .env file, remove it
    with open('.env', 'r') as f:
        lines = f.readlines()
    with open('.env', 'w') as f:
        for line in lines:
            if not line.startswith(name):
                f.write(line)

    # make it persistent
    with open('.env', 'a') as f:
        f.write(f'{name}={key.hex()}\n')

def get_key(name):
    if name in os.environ:
        return bytes.fromhex(os.environ[name])
    else:
        # read from .env file
        with open('.env', 'r') as f:
            for line in f:
                if line.startswith(name):
                    return bytes.fromhex(line.split('=')[1])
```

With this, all we need to do to encrypt is to store the key and then call the chacha20_encrypt function. To decrypt, we just need to call the chacha20_decrypt function, using the key that was previously stored.

Example:

```
# encrypt phone number
key = E.generate_key()
# store the key
E.store_key(key, f"{user.username.upper()}_PHONE_KEY")
user.phone = E.chacha20_encrypt(phone, key)
if user.phone is None:
    flash("Erro ao encriptar número de telefone!", category="danger")
    return redirect(url_for("profile.changeProfile", id=user.id))

# decrypt phone number
phone = E.chacha20_decrypt(
    user.phone, E.get_key(f"{user.username.upper()}_PHONE_KEY")
)
```

Finally, to ensure that the user's PII is encrypted (V6.1.1), we are encrypting the user's email and phone number, and to ensure that the user's financial data is encrypted (V6.1.3), we are encrypting the shipping

and billing addresses and the tracking number (so that, if a tracking system is implemented, the attackers cannot access where the package is or where it is going to be delivered).

3.17 Algorithms (V6.2.1)

"Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable Padding Oracle attacks."

In our application, we use the cryptography library to encrypt and decrypt the user's PII. This library uses the ChaCha20 algorithm, a symmetric stream cipher that does not require padding, making it secure against Padding Oracle attacks.

To ensure that the cryptographic modules fail securely, these use a try-except block to handle any errors that may occur:

```
def chacha20_encrypt(message, key):
    iv = os.urandom(16)

    try:
        cipher = Cipher(algorithms.ChaCha20(key, iv), mode=None,
backend=default_backend())
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(message.encode()) +
encryptor.finalize()
        return iv + ciphertext
    except Exception as e:
        print("Encryption error:", e)
        return None

def chacha20_decrypt(ciphertext, key):
    iv = ciphertext[:16]

    try:
        cipher = Cipher(algorithms.ChaCha20(key, iv), mode=None,
backend=default_backend())
        decryptor = cipher.decryptor()
        plaintext = decryptor.update(ciphertext[16:]) +
decryptor.finalize()
        return plaintext.decode()
    except Exception as e:
        print("Decryption error:", e)
        return None
```

When calling this functions, we check if the return value is None. If it is, we display an error message to the user:

```
# encrypt the tracking number
key = E.generate_key()
E.store_key()
```

```
key,
f"{current_user.username.upper()}"
{number_of_orders+1}_TRACKING_NUMBER_KEY",
)

tracking_number_enc = E.chacha20_encrypt(generate_tracking_number(), key)

if tracking_number_enc is None:
    flash("Erro ao encriptar tracking number!", category="danger")
    return redirect(url_for("checkout.check"))

# encrypt the shipping address
key = E.generate_key()
E.store_key(
    key,
    f"{current_user.username.upper()}"
{number_of_orders+1}_SHIPPING_ADDRESS_KEY",
)

shipping_address_enc = E.chacha20_encrypt(address, key)

if shipping_address_enc is None:
    flash("Erro ao encriptar shipping address!", category="danger")
    return redirect(url_for("checkout.check"))

# encrypt the billing address
key = E.generate_key()
E.store_key(
    key,
    f"{current_user.username.upper()}"
{number_of_orders+1}_BILLING_ADDRESS_KEY",
)

billing_address_enc = E.chacha20_encrypt(address2, key)

if billing_address_enc is None:
    flash("Erro ao encriptar billing address!", category="danger")
    return redirect(url_for("checkout.check"))
```