DETI STORE

Project 1 - SIO

Index

1. Introduction

The present report serves as documentation for Project 1 of SIO which intends to explore the possible vulnerabiilites, their consequences and their counters in a webapp for a ficitious online store: Deti Store.

2. Overview

To implement and counteract our selected vulnerabilites we used Flask: HTML with Boostrap on the frontend, data renderization with templating using Jinja2 and a SQLite database for data persistency on the backend.

3. Vulnerabilites

CWE - 89 - Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

CVSS

Severity: 4.3

Vector String: CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N

Breakdown

Metric	Value	Justification

Metric	Value	Justification
AV	N	The vulnerability is exploitable from a remote network, such as the internet, without requiring user interaction.
AC	L	The attack requires low complexity, such as the availability of an easily guessable or known SQL injection payload.
PR	N	No privileges are required to exploit the vulnerability.
UI	N	No user interaction is required to exploit the vulnerability.
S	U	The vulnerability affects the security of the entire system, not just individual resources within the system.
C	L	The vulnerability may allow an attacker to access or modify sensitive information, but the data is typically difficult to recover or exploited.
I	L	The vulnerability may allow an attacker to modify data, but it is unlikely to have a significant impact on the integrity of the affected system or data.
A	N	The vulnerability does not affect the availability of the affected system or data.

Abstract

SQL injection occurs when an attacker maliciously inserts SQL code into a web form input field with the intention of gaining unauthorized access to resources or manipulating data. In essence, an SQL query is a command that instructs a database to perform a specific action.

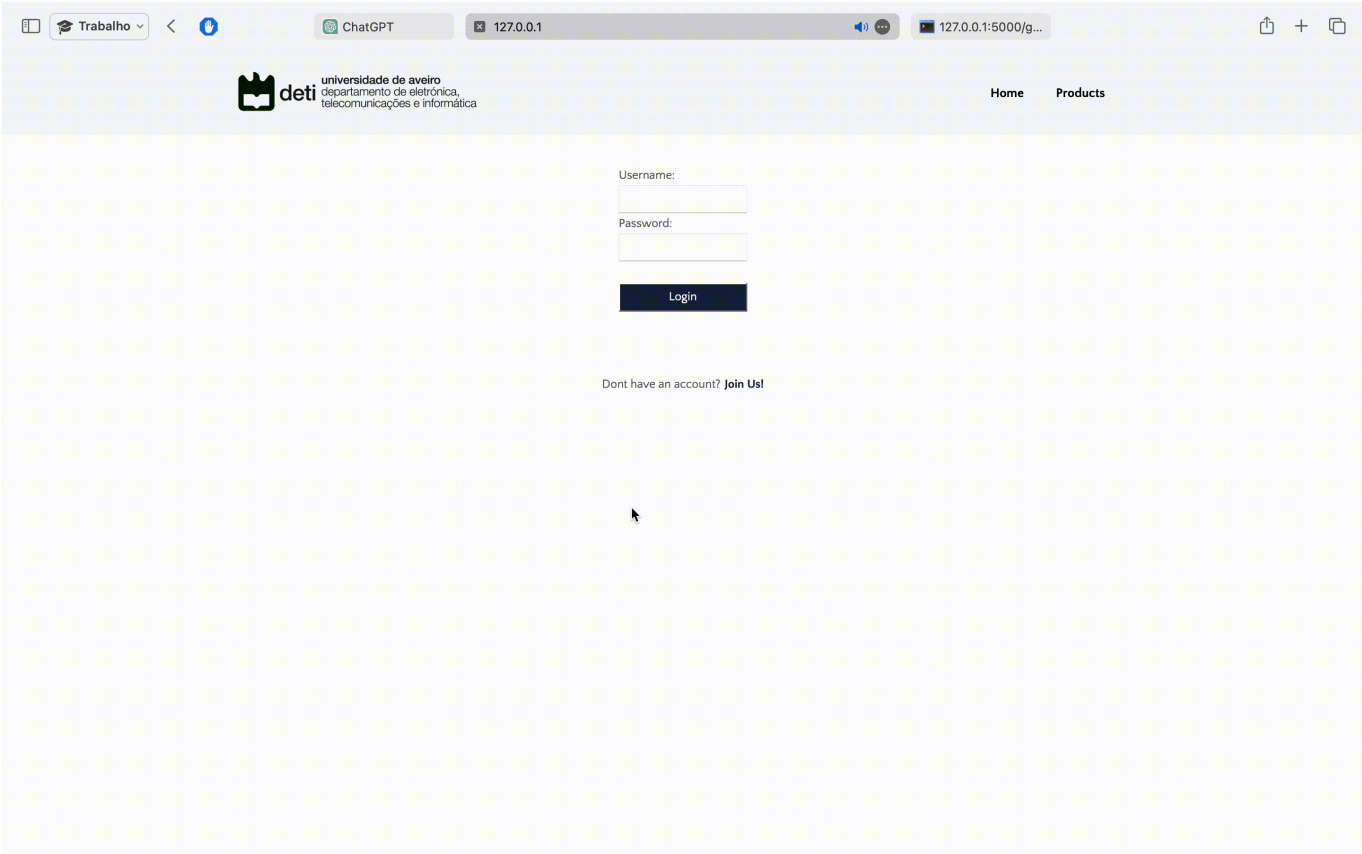
In a typical scenario, when a user enters their username and password into the respective text fields of a web form for authentication, these values are used in a SELECT query. The user is granted access if the provided information matches what's expected. Access is denied if there's no match.

However, many web forms lack adequate security measures to protect against unwanted input manipulation. In the absence of these safeguards, a hacker can exploit the input fields to send custom SQL commands to the database. This can allow them to download the entire database or interact with it in unauthorized ways.

As a result, SQL injection provides the attacker with unrestricted access to sensitive data, such as client details, personal information, proprietary information, trade secrets, and other confidential data. The ability to read, modify, or steal this sensitive information makes it easy for attackers to compromise a system and gain control over it.

Exploitation #1

In this case, SQL injection is possible in the password field of the login page, by entering an input that abuses the SQL quotation notation, for example ' or 1=1 -- as such:



Counteraction

Originally, the password is received and processed directly like so:

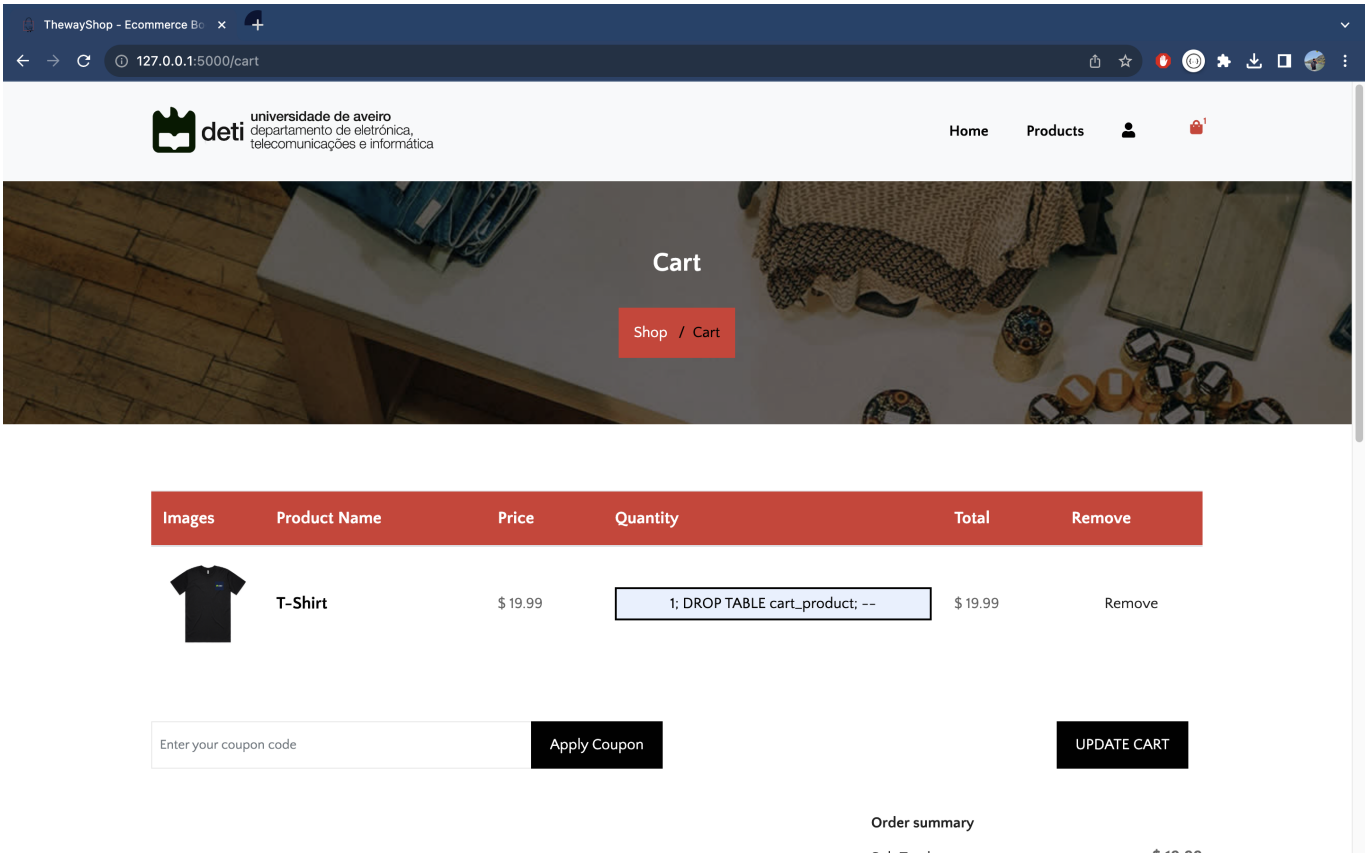
```
query = text(
        "SELECT * FROM user WHERE username = '"
        + user
        + "' AND password = '"
        + key
        + "';"
    )

result = db.session.execute(query).fetchall()

if not result:
    flash("User not found!", "error")
    return redirect(url_for("auth.login"))

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

To correct this, the **werkzeug** library was employed to process the password through hashing. Furthermore, **SQL Alchemy** was used to make sure that the password matches that which is associated with the user. In pratical terms, this translates into a guard clause like the following:

```
user = User.query.filter_by(username=username).first()
if not user or not check_password_hash(user.password, key):
    flash("Please check your login details and try again.")
    return redirect(url_for("auth.login"))

login_user(user)
```

Exploitation #2

In this case, SQL injection is possible in the quantity field of the cart page, by entering an input that abuses the SQL quotation notation:



Counteraction

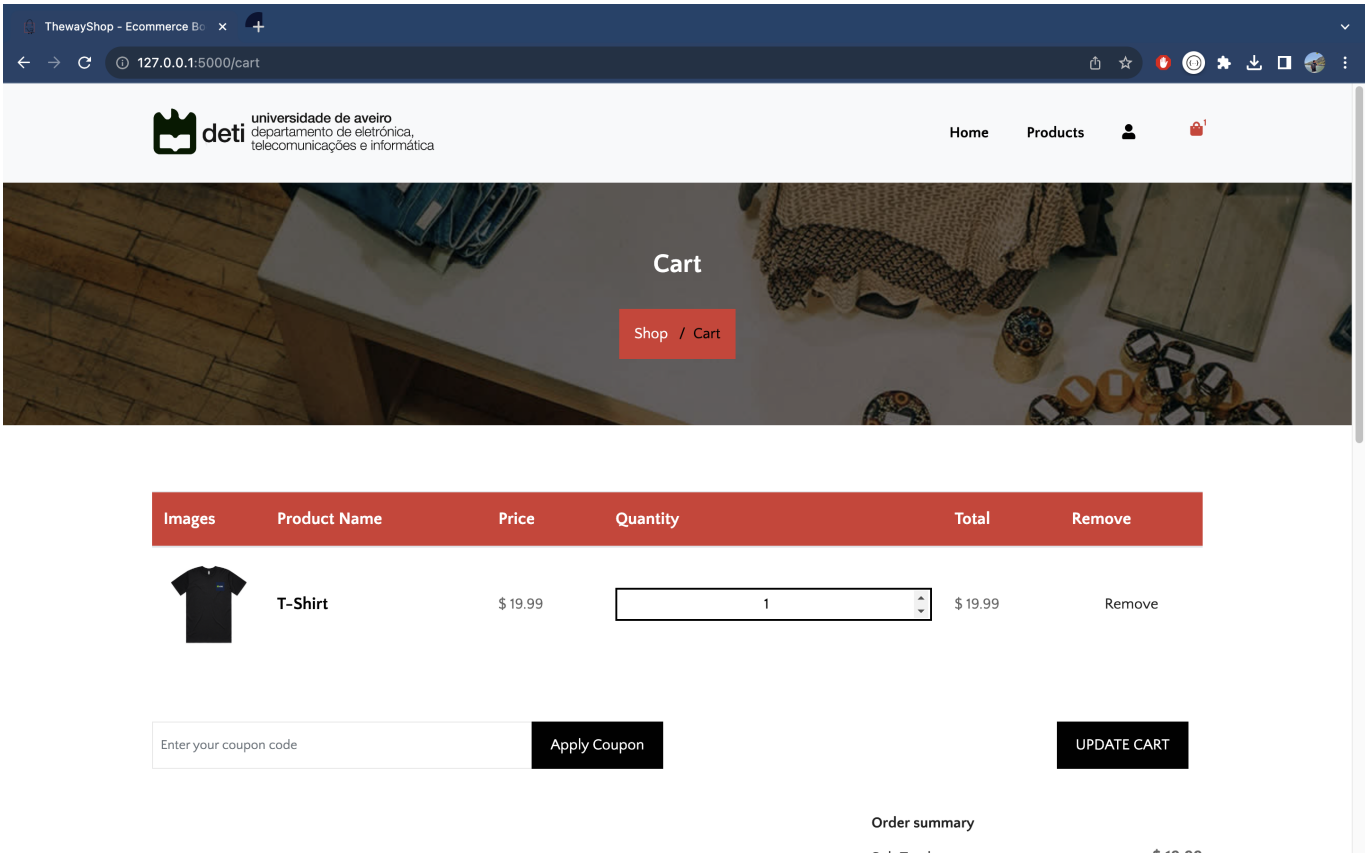
The vulnerability was present in the way we processed the quantity data in both the HTML and Python code. The original code allowed direct user input without proper validation and sanitization. In the HTML, the input field had the type "text," which did not restrict the input to numbers. In the Python code, the quantity was directly incorporated into an SQL query without any validation. Originally, the quantity is received and processed directly like so:

```
<td class="quantity-box">
    <input
    type="text"
    name="product_{{ product.id }}"
    value="{{ product_quantities[product.id] }}"
    min="0"
```

```
    step="1"
    class="c-input-text qty text"
    />
</td>
```

```
for product_id in request.form:
        id = product_id.split("_")[1]
        query = text(
            "UPDATE cart_product SET quantity = "
            + request.form[product_id]
            + " WHERE cart_id = "
            + str(cart.id)
            + " AND product_id = "
            + str(id)
            + ""
        )
        db.session.execute(query)
        db.session.commit()
```

To mitigate this SQL injection vulnerability, we made critical changes to how we handle the quantity data in our application. In the HTML file, we restricted the input to numeric values only by setting the input type to "number." In the Python code, we validate the input to ensure it is a number before updating the database. If the input is not a number, we generate an error message and prevent any database updates.



```
<td class="quantity-box">
    <input
    type="number"
    name="product_{{ product.id }}"
    value="{{ product_quantities[product.id] }}"
    min="0"
    step="1"
    class="c-input-text qty text"
    />
</td>
```

```
for product_id in request.form:
    id = product_id.split("_")[1]

    if request.form[product_id].isnumeric() == False:
        flash("Invalid quantity.", "error")
        return redirect(url_for("cart.cart"))

    query = text(
        "UPDATE cart_product SET quantity = "
        + request.form[product_id]
        + " WHERE cart_id = "
        + str(cart.id)
        + " AND product_id = "
        + str(id)
        + ""
    )
    db.session.execute(query)
    db.session.commit()
```

CWE - 352 - Cross-Site Request Forgery

CVSS

Severity: 6.1

Vector String: CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:N/I:H/A:N

Breakdown

Metric	Value	Justification
AV	N	The vulnerability is exploitable from a remote network, such as the internet, without requiring direct access to the target system. An attacker can leverage the vulnerability by crafting a malicious website that is visited by the target user.
AC	L	The attack requires low complexity, such as the availability of a known CSRF payload or the ability to predict predictable input fields in the application.

Metric	Value	Justification
PR	N	No privileges or special knowledge are required to exploit the vulnerability.
UI	R	User interaction is required to exploit the vulnerability, such as visiting a malicious website or clicking a malicious link.
S	C	The vulnerability only affects the security of individual resources, such as a single user account, rather than the entire system.
C	N	The vulnerability does not allow an attacker to access sensitive information or steal user data.
I	H	The vulnerability allows an attacker to modify data, such as changing the target user's account settings or making unauthorized purchases.
A	N	The vulnerability does not affect the availability of the affected system or data, but it still poses a serious security risk.

Abstract

When a web server is designed to accept requests from clients without verifying their authenticity, it creates an opportunity for attackers to deceive a client into making unintentional requests to the web server, which are treated as legitimate requests. This can be achieved through various methods, including manipulating URLs, loading images, making XMLHttpRequests, or other techniques, ultimately leading to potential data exposure or unintended execution of code.

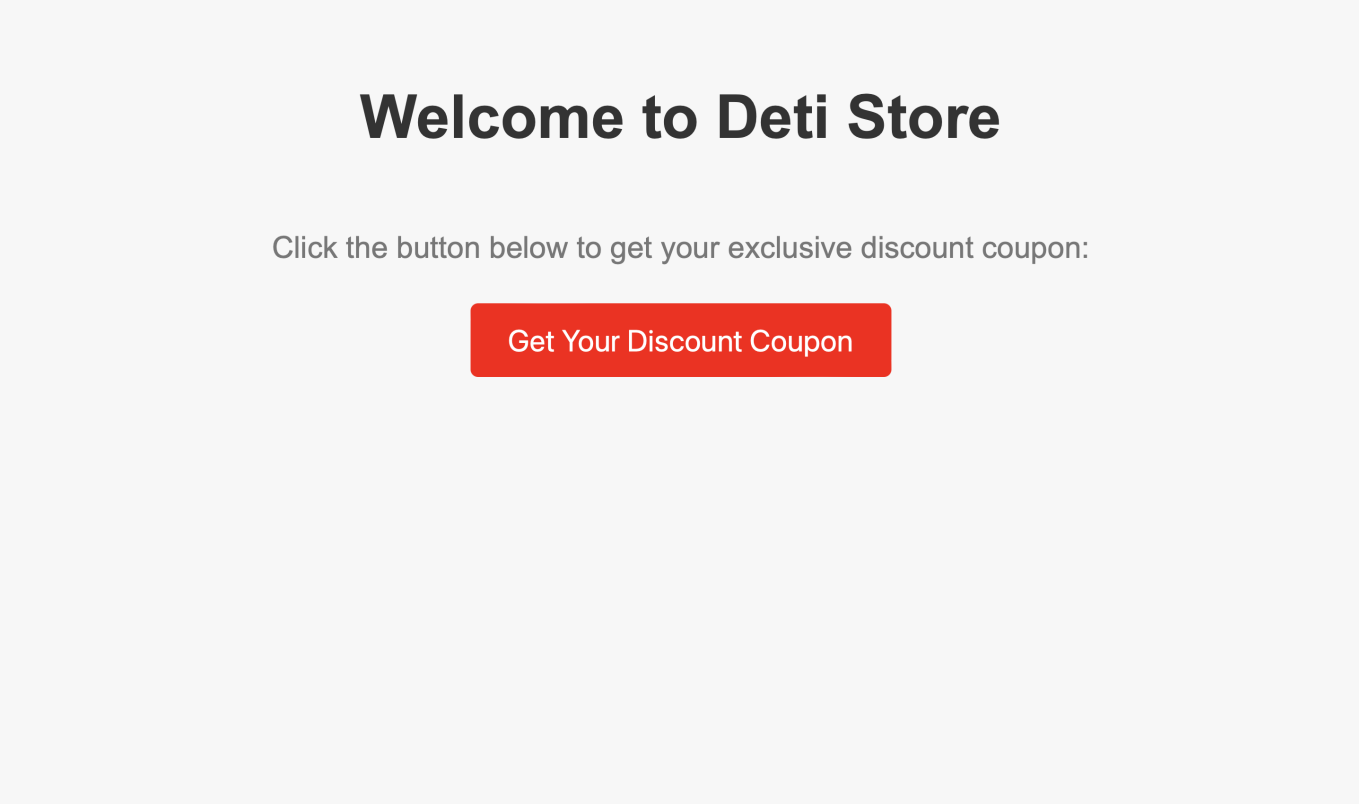
To mitigate and prevent Cross-Site Request Forgery (CSRF) attacks, web developers can implement several protective measures:

1. **Anti-CSRF Tokens**: Web applications can include anti-CSRF tokens in each request. These tokens are unique to each user session and are verified by the server to ensure that the request is legitimate. Without a valid token, the server rejects the request.
2. **Rate-Limiting Requests**: Developers can implement rate limiting to restrict the number of requests a user can make within a certain timeframe. This can help prevent a flood of malicious requests from an attacker.
3. **Same-Site Cookie Policies**: Utilizing same-site cookie attributes can be an effective defense against CSRF attacks. By setting cookies as "SameSite=Lax" or "SameSite=Strict," it restricts the scope of cookies to the same origin, making it harder for attackers to manipulate requests across different sites.

These measures collectively enhance the security of web applications and help safeguard against CSRF attacks, which aim to exploit the trust that a user's browser has in a particular website to perform unauthorized actions on their behalf.

Exploitation

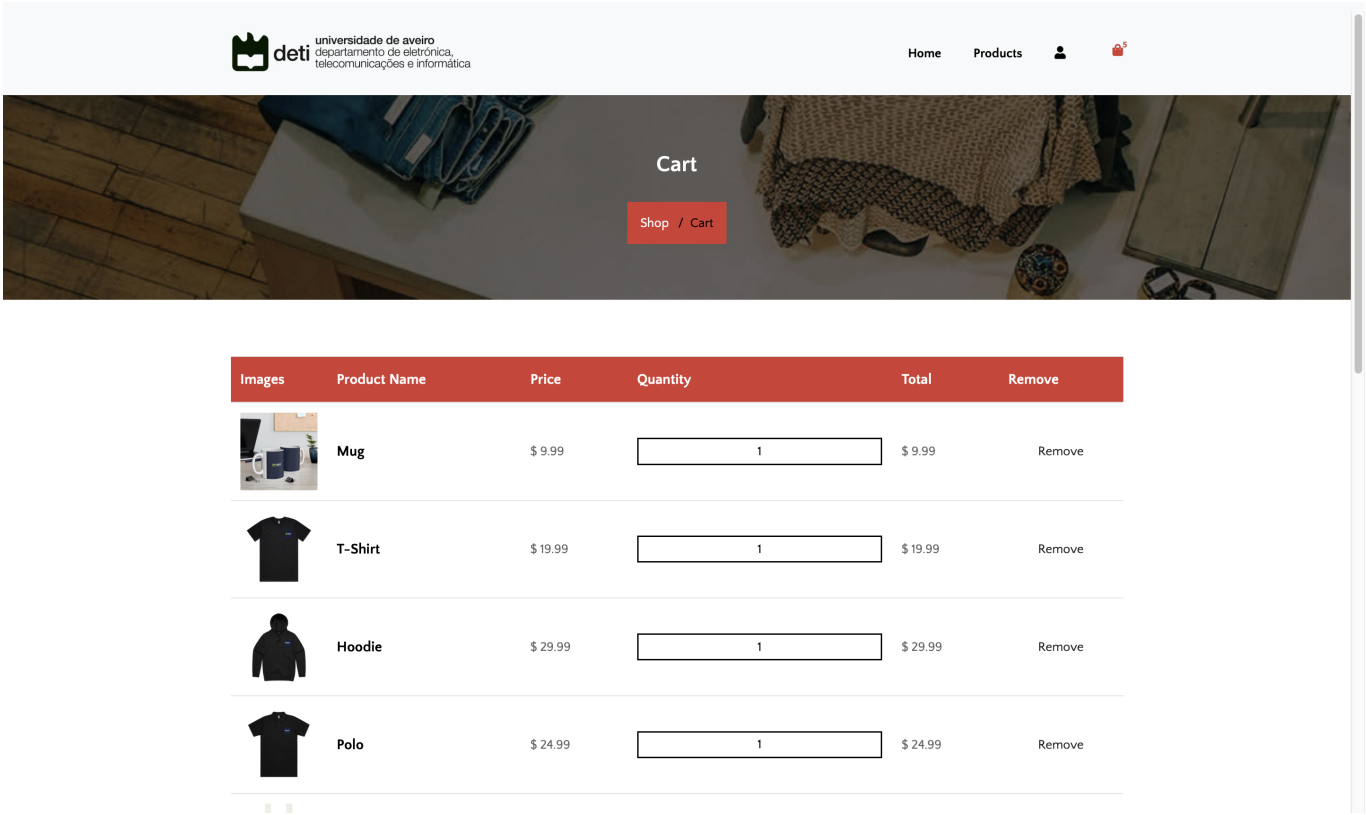
For the purposes of this assignment, we chose to implement a fake, scam site, that would resemble the overall appearance of our real site, enough so that at least some more naïve users would fall for, as seen below:



In reality, it hides a malicious intent, implemented with the following hidden input, that is submitted when the user clicks the button:

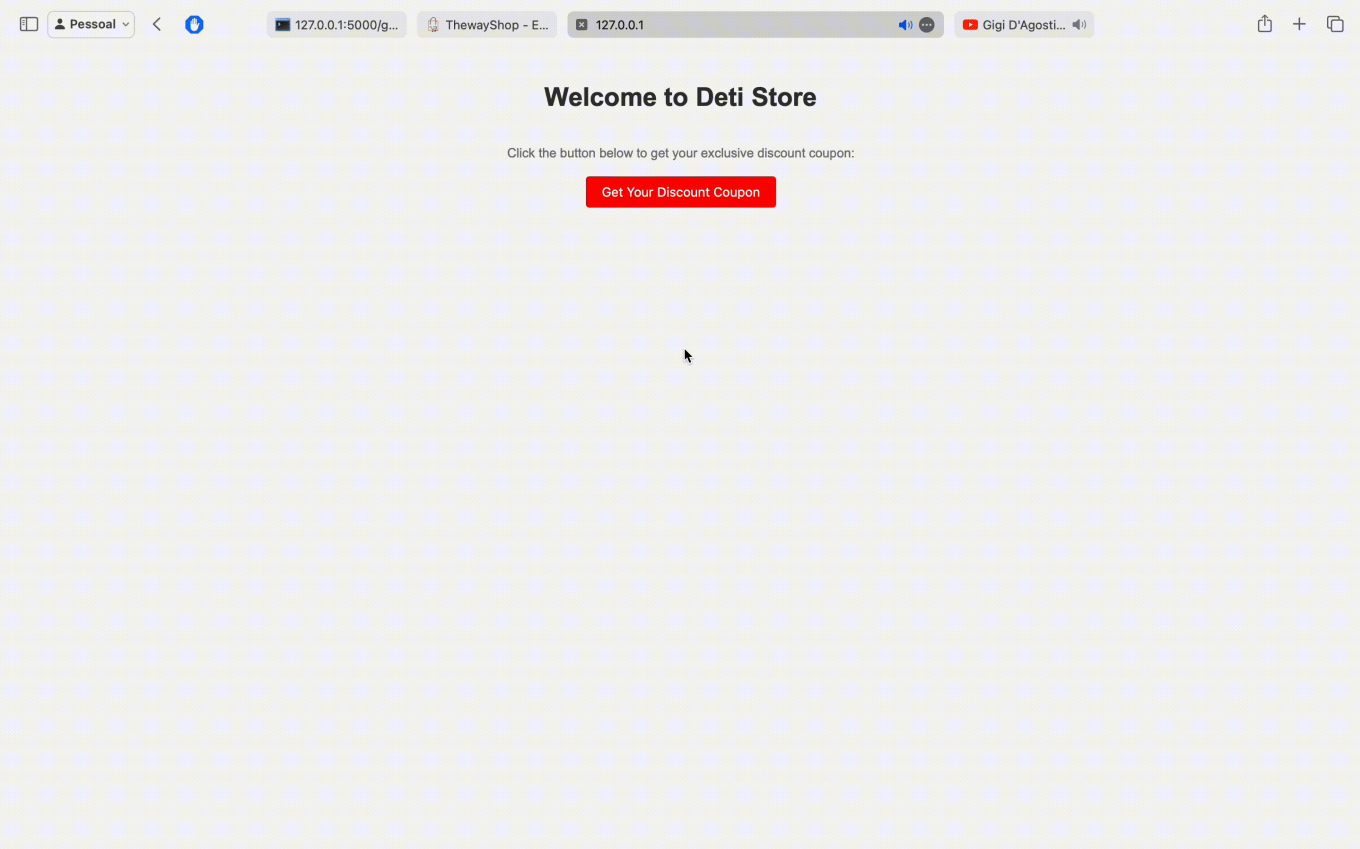
```
<form
  hidden
  id="hack"
  target="csrf-frame"
  action="http://127.0.0.1:5000/shop/add_to_cart/1"
  method="POST"
  autocomplete="off"
></form>

<iframe
  hidden
  name="csrf-frame"
  id="frame"
  width="1000px"
  height="1000px"
></iframe>
```

What the page does is pretend to offer free discount coupons for the store, but instead, it utilizes the user's stored cookies to insert fictitious products into the shopping cart, as can be observed at the bottom of the user's shopping list:



Here's a dramatization of how this vulnerability could be exploited in a real-world scenario:

In a real-world scenario, a user lands on our fraudulent website, lured by the illusion of receiving free discount coupons for their favorite store. Driven by curiosity and the promise of savings, they click on the enticing button to proceed. However, unbeknownst to them, their innocent click triggers a devious scheme. Upon returning to the genuine e-commerce site, the user is suddenly confronted with an alarming message injected by the attacker. To their utter dismay, a new set of imaginary products now populates their shopping cart, thanks to the exploitation of their stored cookies.



Counteraction

Flask-WTF is a widely used library for building web forms in Flask, and it comes with a built-in safeguard against CSRF (Cross-Site Request Forgery) attacks. This protection is implemented by adding a unique security token to every form created with the library.

When Flask-WTF generates a form, it discreetly includes a hidden field in the HTML code, containing a special token. This token is sent back to the server when the user submits the form. On the server side, Flask-WTF checks this token against the stored value and only processes the form if the token matches. This mechanism ensures that, even if an attacker manages to trick a user into submitting a form, the attack is thwarted because the attacker doesn't possess the correct anti-CSRF token.

In simple terms, this means that attackers can't submit any data unless they have access to the specific page where the data should be submitted.

To apply this security measure, all you need to do is import the Flask-WTF library and add a hidden input field as shown below in your form code:

```
<form method="POST" action="/product/add_to_cart/{{ product.id }}">
  <input type="hidden" name="csrf_token" value="{{ csrf_token() }}" />
  <button class="btn btn-success" style="margin-right: 5px;">
    Add To Cart
  </button>
  {% with messages = get_flashed_messages() %} {% if messages %} {% for
message
  in messages %}
  <div class="alert alert-danger" role="alert">{{ message }}</div>
```

```
    {% endfor %} {% endif %} {% endwith %}
</form>
```

This way, your application becomes more resilient to CSRF attacks, providing an additional layer of security.

CWE - 798 - Use of Hard-coded Credentials

CVSS

Severity: 8.8

Vector String: CVSS:3.1/AV:A/AC:H/PR:H/UI:N/S:U/C:H/I:H/A:N

Breakdown

Metric	Value	Justification
AV	A	The vulnerability is exploitable from the local system and does not require access to a network. An attacker can leverage the vulnerability by directly accessing the vulnerable software or device.
AC	H	The attack requires high complexity, such as reverse engineering the software or having physical access to the device.
PR	H	High privileges or special knowledge are required to exploit the vulnerability, such as access to the source code or administrator privileges.
UI	N	No user interaction is required to exploit the vulnerability.
S	U	The vulnerability affects the security of the entire system, potentially exposing all user data or the integrity of the system.
C	H	The vulnerability allows an attacker to access sensitive information or steal user data, such as usernames and passwords, potentially compromising the entire system.
I	H	The vulnerability allows an attacker to modify data, such as changing the target user's account settings or making unauthorized purchases.
A	N	The vulnerability does not affect the availability of the affected system or data, but it still poses a serious security risk.

Abstract

Hard-coded credentials represent a significant security vulnerability that can enable attackers to bypass the authentication mechanisms configured by software administrators. This vulnerability is often elusive for system administrators, and even if detected, rectifying it can be an insurmountable challenge, sometimes necessitating the complete disabling of the affected product.

In this project, we're specifically focusing on the inbound variation of this vulnerability, where a default administrator account is created with a hardcoded, simplistic password embedded in the product. This password remains identical across all installations, and administrators are often unable to modify or

deactivate it without manual intervention, like editing the application's code or updating the software. If this password becomes known or is published (which is unfortunately common on the internet), anyone possessing it can gain unauthorized access to the product. Furthermore, because all instances of the program share the same password, even across different organizations, it opens the door to large-scale attacks like worms.

To mitigate this security risk, it is imperative to avoid using hard-coded credentials in software development. Instead, administrators should be compelled to establish unique and robust passwords for each installation of the product, and the software should be designed to securely store these credentials. This approach helps in averting unauthorized access and limiting the potential fallout of a security breach. Additionally, conducting regular security audits and testing is essential to uncover and rectify any instances of hard-coded credentials within the product. This proactive approach bolsters the overall security posture of the software.

Exploitation

During development, several test users can be created and left, by accident, in the database tables, for example:

```
username: admin

password: admin123
```

Counteraction

Our approach to addressing this issue involves the development and implementation of a script designed to perform a database sanity check during the deployment process. This ensures that the product is deployed in a secure and clean state. Here is how our implementation works:

1. **Script Creation**: We have created a dedicated script for this purpose. This script is designed to be executed during the deployment phase.
2. **Database Inspection**: The script examines the database to identify any default administrator accounts with hard-coded passwords.
3. **Password Update**: If the script detects default administrator accounts, it deletes them. It then prompts the administrator to create a new password for the administrator account.
4. **Deployment Continues**: Once the script completes its database sanity check, the deployment process can proceed with the assurance that the product is in a more secure and compliant state.

By incorporating this script into the deployment workflow, we reduce the risk associated with hard-coded credentials and other security vulnerabilities. This approach helps to ensure that each deployment is carried out with security in mind, reducing the potential for unauthorized access and other security threats. Regular execution of this script as part of the deployment process maintains the ongoing security and integrity of the product.

```
def check_db_security(db):
    query = text("SELECT username FROM user WHERE isAdmin = True;")
    result = db.session.execute(query).fetchall()
    usernames = [username[0] for username in result]

    for username in usernames:
        query = text("SELECT * FROM user WHERE username = '" + username +
"';")
        result = db.session.execute(query).fetchall()

        print("User " + username + " found!")
        query = text("DELETE FROM user WHERE username = '" + username +
"';")
        db.session.execute(query)
        db.session.commit()
        print("Deleted user: " + username)
        print("--------------------")
```

CWE - 620 - Unverified Password Change

CVSS

Severity: 8.1

Vector String: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

Breakdown

Metric	Value	Justification
AV	N	The vulnerability is exploitable from a remote network, such as the internet, without requiring direct access to the target system. An attacker can leverage the vulnerability by exploiting a weakness in the password change mechanism.
AC	L	The attack requires low complexity, such as exploiting a known vulnerability in the password change mechanism or guessing a user's password reset token.
PR	N	No privileges or special knowledge are required to exploit the vulnerability.
UI	N	No user interaction is required to exploit the vulnerability.
S	U	The vulnerability affects the security of the entire system, potentially exposing all user data or the integrity of the system.
C	H	The vulnerability allows an attacker to access sensitive information or steal user data, such as changing the target user's password and compromising their account.
I	H	The vulnerability allows an attacker to modify data, such as changing the target user's account settings or making unauthorized purchases.

Metric	Value	Justification
A	N	The vulnerability does not affect the availability of the affected system or data, but it still poses a serious security risk.

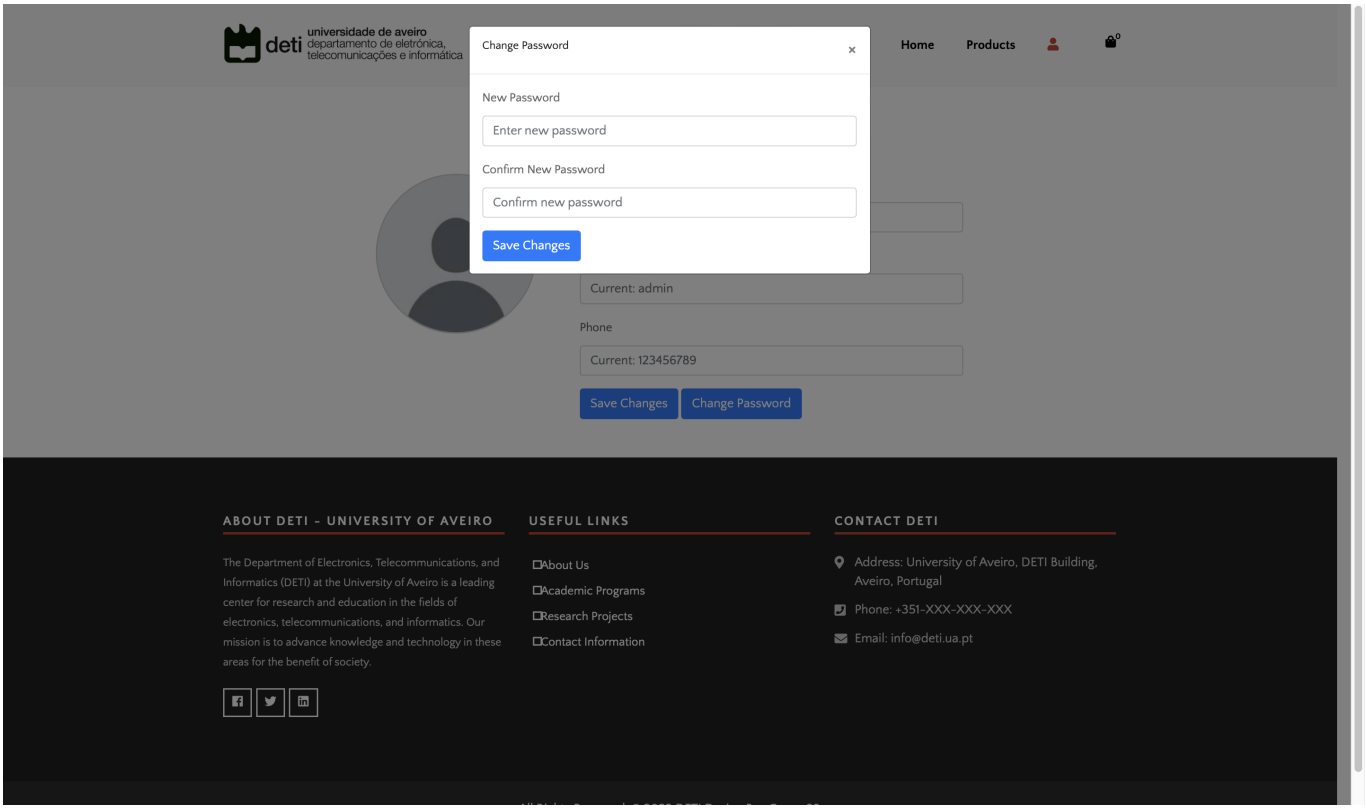
Abstract

The product does not require knowledge of the original password or the utilization of an alternative form of authentication when resetting a user's password. While this approach can provide convenience for users, it introduces a security vulnerability that attackers may exploit.

An attacker could potentially exploit this vulnerability to change passwords for other user accounts, effectively gaining unauthorized access and acquiring the privileges associated with those compromised accounts. This scenario can lead to significant security breaches and unauthorized activities within the system.

Exploitation

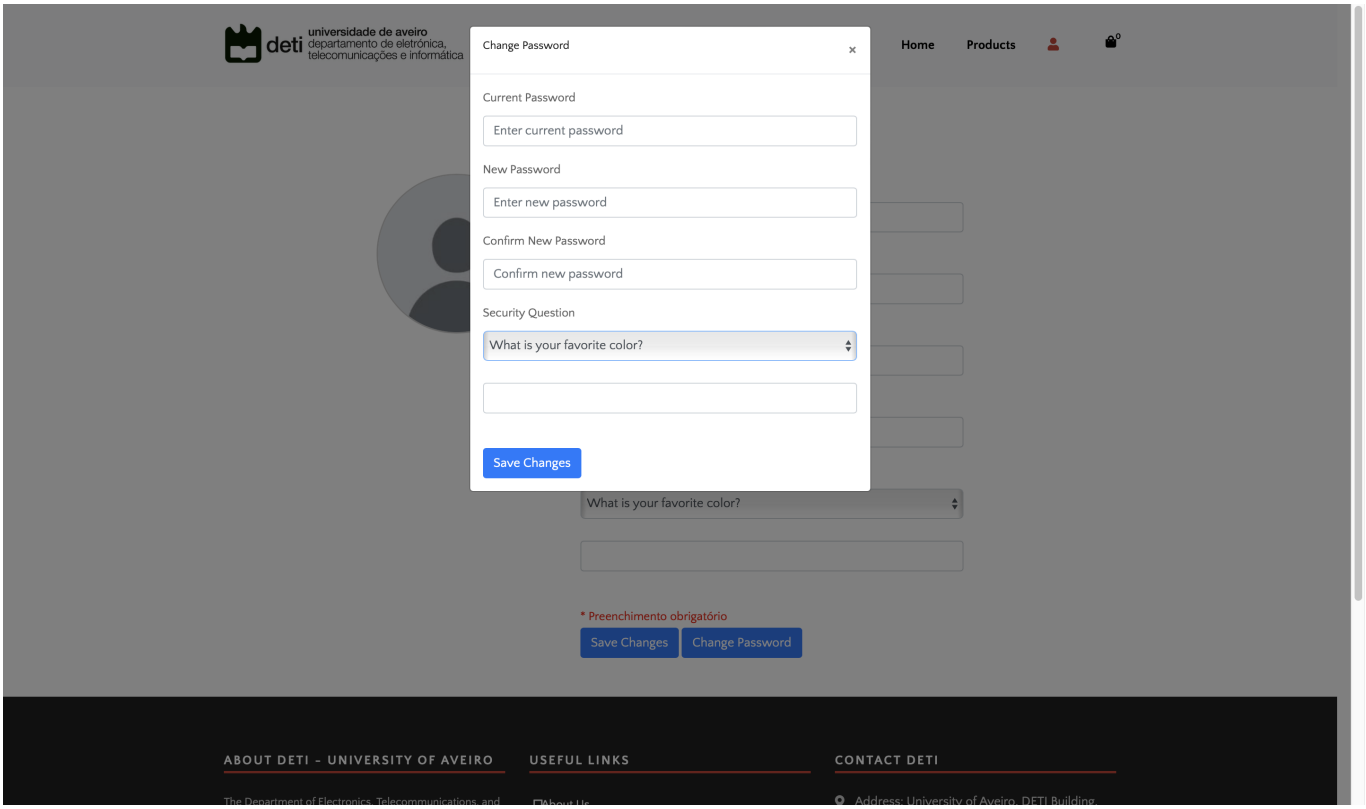
Not requesting the user's current password when editing their profile can create a security vulnerability. In such a scenario, a malicious actor who gains access to the user's current session could potentially exploit this vulnerability to lock the user out of their own account. This could be done by making unauthorized changes to the user's profile settings, such as changing the password or email address.



Counteraction

Introducing a field that necessitates the user to input their current password not only ensures that their account isn't currently compromised but also adds an additional layer of security. Additionally, incorporating a security question as part of the authentication process further enhances the account's protection. This

combination of measures safeguards the user's account from unauthorized access and helps verify the user's identity before making any profile changes.



CWE - 521 - Weak Password Requirements

CVSS

Severity: 7.5

Vector String: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

Breakdown

Metric	Value	Justification
AV	N	The vulnerability is exploitable from a remote network, such as the internet, without requiring direct access to the target system. An attacker can leverage the vulnerability by exploiting the weak password requirements and guessing or cracking weak passwords.
AC	L	The attack requires low complexity, such as exploiting a known vulnerability in the password requirements or guessing a user's password.
PR	N	No privileges or special knowledge are required to exploit the vulnerability.
UI	N	No user interaction is required to exploit the vulnerability.
S	U	The vulnerability affects the security of the entire system, potentially exposing all user data or the integrity of the system.

Metric	Value	Justification
C	H	The vulnerability allows an attacker to access sensitive information or steal user data, such as compromising the target user's account.
I	H	The vulnerability allows an attacker to modify data, such as changing the target user's account settings or making unauthorized purchases.
A	N	The vulnerability does not affect the availability of the affected system or data, but it still poses a serious security risk.

Abstract

Authentication systems often rely on a user's memorized secret, typically referred to as a password, to establish and confirm their identity. Therefore, it is crucial that these passwords are sufficiently strong and intricate to make it challenging for potential attackers to guess or breach. The specific criteria for password complexity can differ based on the type of system being protected. The key is to select appropriate password requirements and ensure they are effectively implemented to enhance the overall security of the authentication mechanism.

Exploitation

The vulnerability exploitation essentially centers around the idea that a straightforward or simple password is easy to crack, making it a risky choice to permit.

Counteraction

To mitigate this weakness, we take a straightforward approach: we disallow users from using weak passwords. We achieve this by rejecting passwords that exhibit characteristics commonly associated with vulnerability, such as being shorter than 8 characters, lacking a digit, lacking a mix of uppercase and lowercase letters, or lacking a special symbol. We enforce these criteria using conditional `if` statements in our password validation process.

```
if len(key) < 8:
    flash("A senha deve ter pelo menos 8 caracteres")
    return redirect(url_for("register.regist"))
elif not any(char.isdigit() for char in key):
    flash("A senha deve ter pelo menos um número")
    return redirect(url_for("register.regist"))
elif not any(char.isupper() for char in key):
    flash("A senha deve ter pelo menos uma letra maiúscula")
    return redirect(url_for("register.regist"))
elif not any(char.islower() for char in key):
    flash("A senha deve ter pelo menos uma letra minúscula")
    return redirect(url_for("register.regist"))
elif not any(char in "~`! @#$%^&*()_-+={[}]|\:;\"'<,>.?/" for char in
key):
    flash("A senha deve ter pelo menos um caractere especial")
    return redirect(url_for("register.regist"))
```

CWE - 522 - Insufficiently Protected Credentials

CVSS

Severity: 7.5

Vector String: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

Breakdown

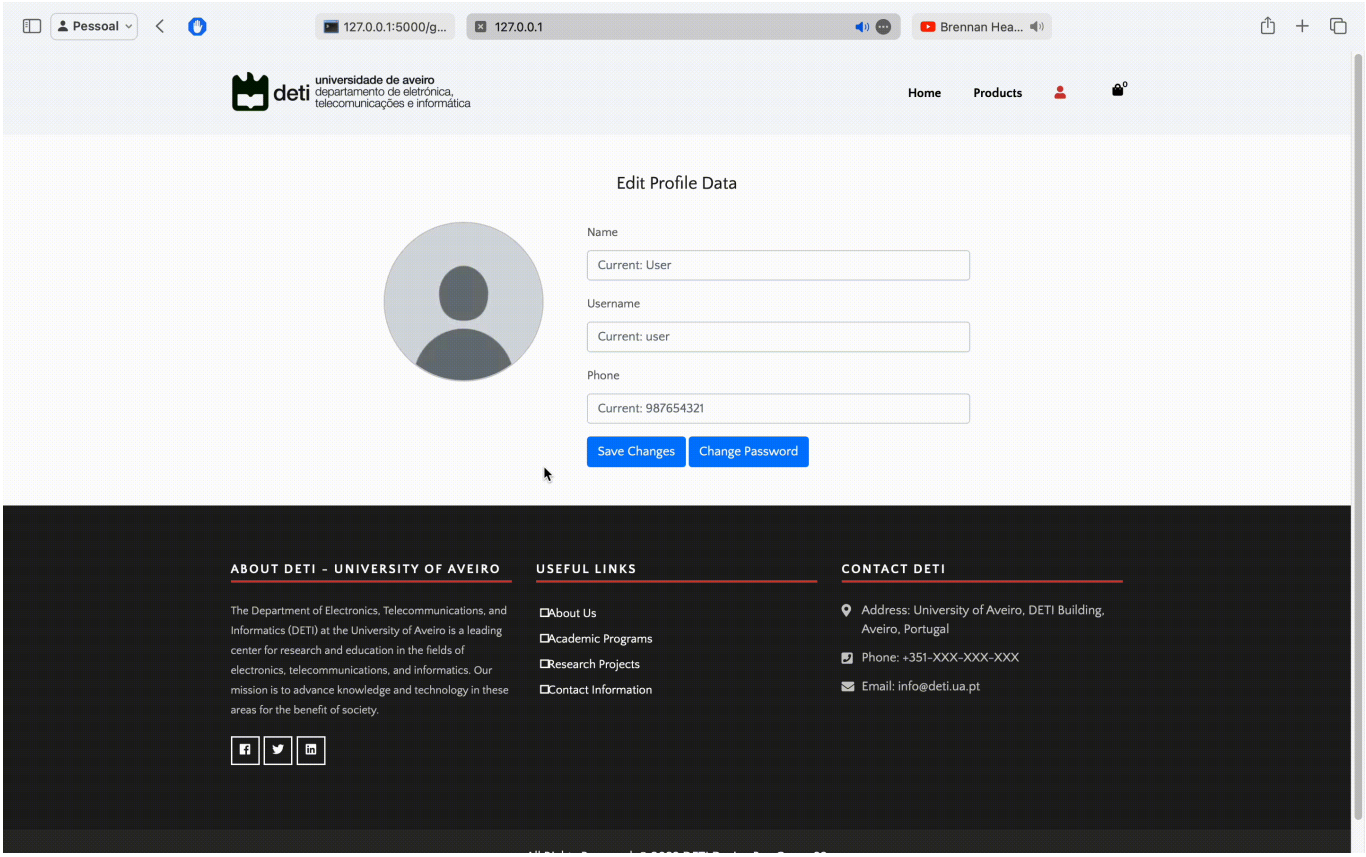
Metric	Value	Justification
AV	N	The vulnerability is exploitable from a remote network, such as the internet, without requiring direct access to the target system. An attacker can leverage the vulnerability by exploiting insufficient protection of the target system's credentials, such as stealing clear-text passwords or unencrypted authentication tokens.
AC	L	The attack requires low complexity, such as exploiting a known vulnerability in the credential storage mechanism or accessing unsecured data storage.
PR	N	No privileges or special knowledge are required to exploit the vulnerability.
UI	N	No user interaction is required to exploit the vulnerability.
S	U	The vulnerability affects the security of the entire system, potentially exposing all user data or the integrity of the system.
C	H	The vulnerability allows an attacker to access sensitive information or steal user data, such as compromising the target user's account.
I	H	The vulnerability allows an attacker to modify data, such as changing the target user's account settings or making unauthorized purchases.
A	N	The vulnerability does not affect the availability of the affected system or data, but it still poses a serious security risk.

Abstract

The website transmits or stores authentication credentials, but it does so in an insecure manner, making it susceptible to unauthorized monitoring or extraction.

Exploitation

While editing the user profile, it's possible to change the field in the URL that corresponds to the user's ID to the ID of another existing user. This action grants access to the editing page of the targeted user's profile.



Counteraction

As mentioned earlier, the URL includes a field related to the user's ID. This is because the current user's ID is passed as an argument within the routing system, as follows:

```
@profile.route("/profile/<int:id>", methods=["GET"])
@login_required
def changeProfile(id):
```

To mitigate this issue, the URL for this page no longer includes the ID field, and this argument is omitted from the routing process. Consequently, it's no longer considered, preventing a user from accessing other editing pages that could potentially alter third-party data, like this:

```
@profile.route("/edit_profile", methods=["GET"])
@login_required
def changeProfile():
```

CWE-434 - Unrestricted Upload of File with Dangerous Type

CVSS

Severity: 7.5

Vector String: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

Breakdown

Metric	Value	Justification
AV	N	The vulnerability is exploitable from a remote network, such as the internet, without requiring direct access to the target system. An attacker can leverage the vulnerability by uploading a file with a dangerous type, such as a script or executable, and executing it on the target system.
AC	L	The attack requires low complexity, such as uploading a file to an unprotected endpoint or exploiting a misconfigured file type validation mechanism.
PR	N	No privileges or special knowledge are required to exploit the vulnerability.
UI	N	No user interaction is required to exploit the vulnerability.
S	U	The vulnerability affects the security of the entire system, potentially exposing all user data or the integrity of the system.
C	H	The vulnerability allows an attacker to execute arbitrary code, such as downloading malware or compromising the target system.
I	H	The vulnerability allows an attacker to modify data, such as modifying system files or inserting malicious code.
A	N	The vulnerability does not affect the availability of the affected system or data, but it still poses a serious security risk.

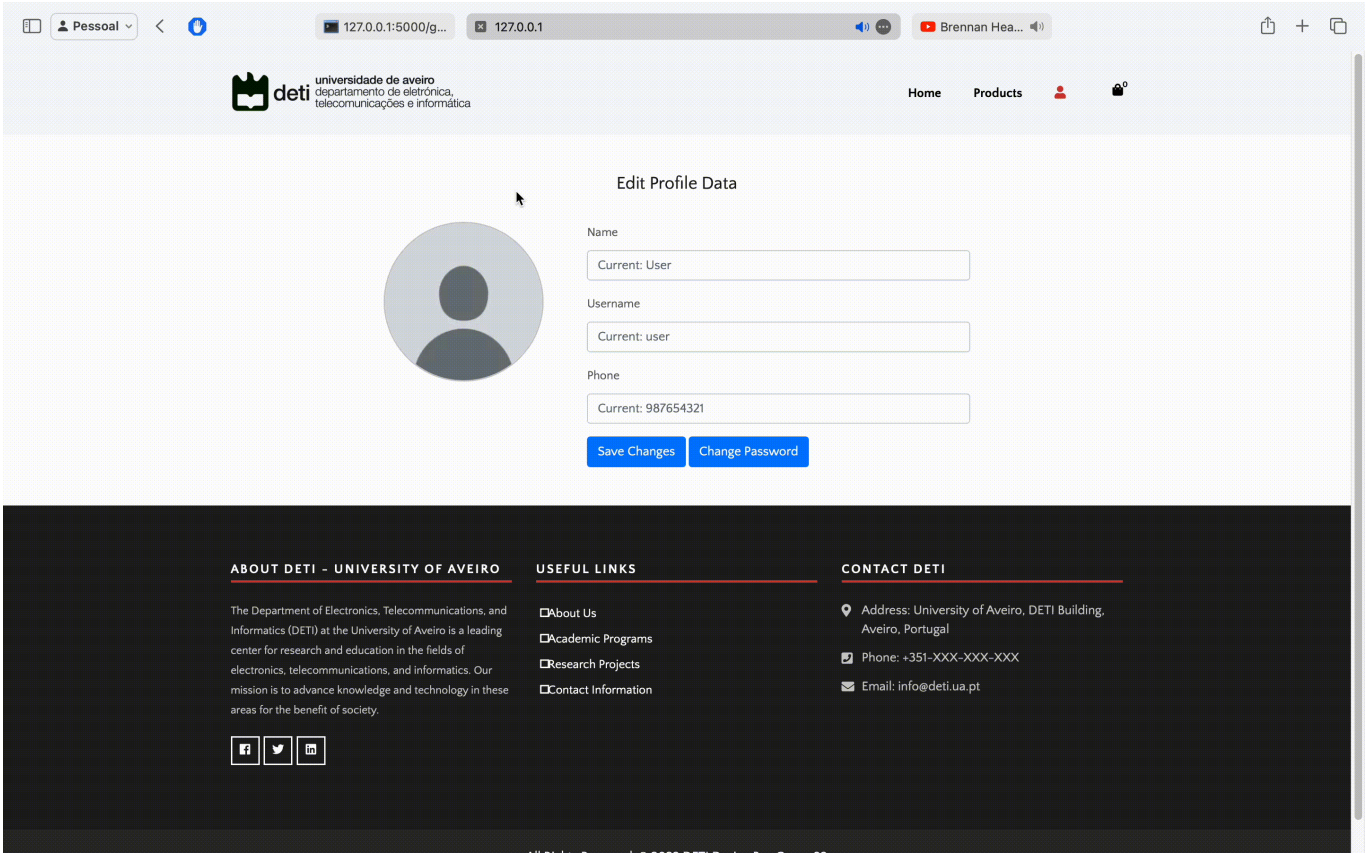
Abstract

The software allows for the upload or transfer of file types that carry inherent security risks, and these files can be automatically processed within the product's environment.

Exploitation

During the process of editing a user's profile, the website permits the upload of files, with the intention that they should be in PNG or JPEG file formats. However, this validation relies on the user's cooperation, leaving a potential vulnerability where a malicious user could upload a file of a dangerous type, capable of disrupting the application's regular operation.

In the following example, the user submits a file in JPG format, and as expected, the system accepts it, potentially creating a security risk.



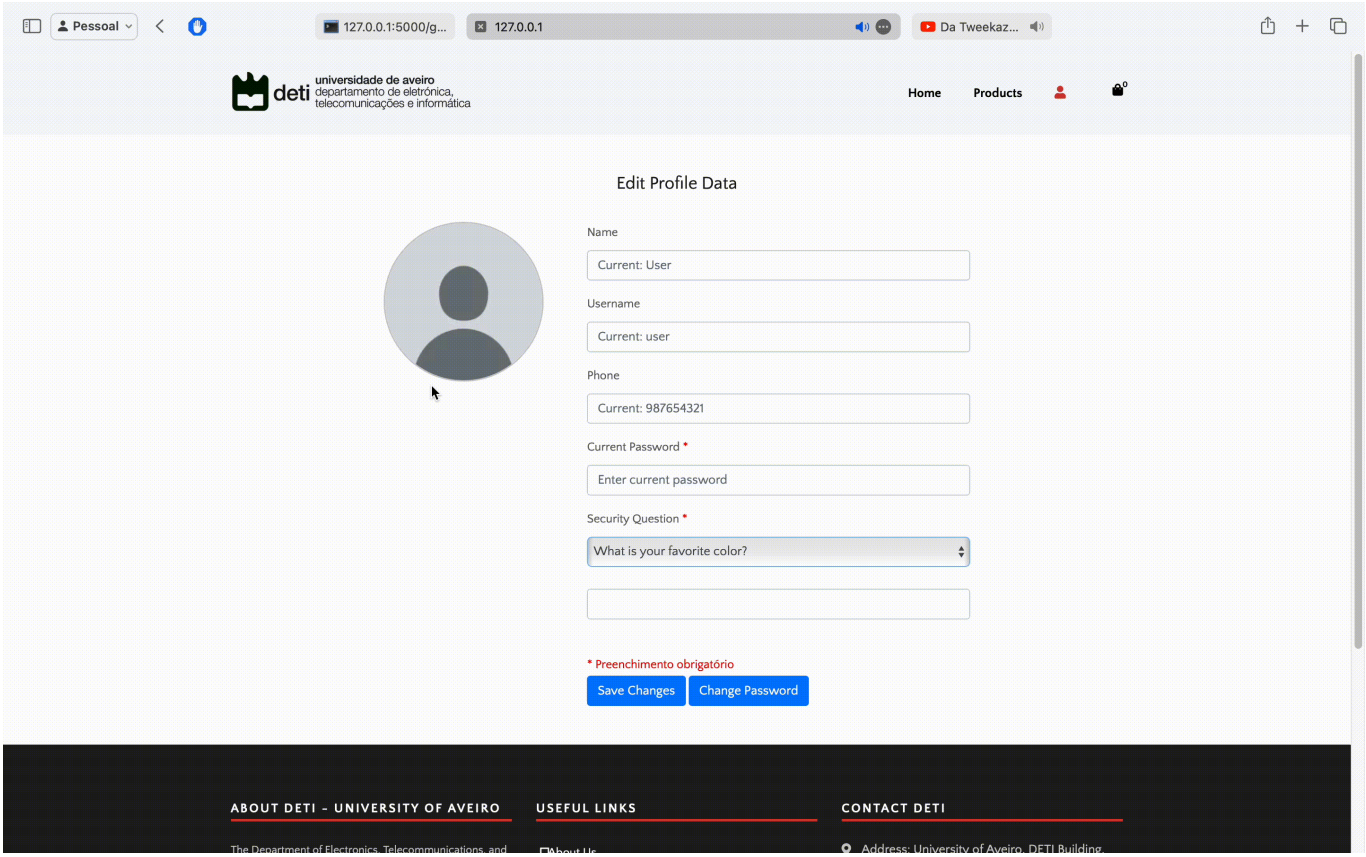
Counteraction

To address this issue, we have implemented a straightforward guard clause to prevent the upload of any file type other than PNG or JPEG. This helps ensure that only safe and permitted file types can be uploaded.

```
if profile_picture.filename.endswith(
    ".png"
) or profile_picture.filename.endswith(".jpeg"):
    try:
        profile_picture.save(
            os.path.join("static/images", profile_picture.filename)
        )
        new_user = User(
            username=user,
            password=generate_password_hash(key),
            name=nome,
            email=email,
            phone=phone,
            image=profile_picture.filename,
            security_question=security_question,
        )
    except:
        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",
        category="danger",
```

```
        )
        return redirect(url_for("register.regist"))
```

After implementing this change, the system's behavior is as follows:



---

4. Final Considerations

In addition to addressing the previously mentioned attack vectors, we made an effort to tackle CWE-1336, which is commonly referred to as 'Template Injection.' However, this vulnerability was already mitigated in a prior version of Jinja2, the template engine used in the project.

This project significantly raised our awareness of the critical importance of developing applications and services with a strong emphasis on security. It underscored the risks associated with neglecting security considerations in software development, emphasizing the need to prioritize security throughout the entire development process.