Relatório de SIO
Projecto Nº 1 - Vulnerabilidades

Universidade de Aveiro

João Pedro Nunes Vieira

José Miguel Guardado Silva

Henrique Miguel Escudeiro Cruz

Luís Manuel Trindade Diogo

Departamento de Electrónica Telecomunicações e Informática

Ver.: 1.1

Relatório de SIO
Projecto Nº 1 - Vulnerabilidades

Departamento de Electrónica Telecomunicações e Informática

Universidade de Aveiro

João Pedro Nunes Vieira, Nº Mec.: 50458

joaopvieira@ua.pt

José Miguel Guardado Silva, Nº Mec.: 103248

jm.silva@ua.pt

Henrique Miguel Escudeiro Cruz, Nº Mec.: 103442

henriquecruz@ua.pt

Luís Manuel Trindade Diogo, Nº Mec.: 108668

luismtd@ua.pt

Novembro de 2023

Resumo

O presente relatório aborda o desenvolvimento de um website do estilo loja online, onde foram implementadas duas soluções distintas relativamente à segurança informática, cujo o objectivo é a venda merchandise do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro usando conceitos abordados no decorrer da Unidade Curricular de Segurança Informática e Nas Organizações.

Este trabalho é relevante pois permite a aplicação de conceitos adquiridos no contexto de aula teórica, nomeadamente o foco na previsibilidade de sistemas, processos, ambientes etc... de soluções informáticas seguras, tais como o planeamento, desenvolvimento, execução, análise e auditoria, com especial foco na defesa contra actividades não autorizadas fazendo prevalecer o funcionamento previsto da solução recorrendo a políticas de segurança pré estabelecidas, os "Objectivos da Segurança da Informação"Confidenciality, Integrity and Availability (CIA) e conhecimento sobre vulnerabilidades comuns Common Weakness Enumeration (CWE) a serem evitadas e/ou corrigidas.

O projecto foi realizado por quatro pessoas, tendo como objectivos a consolidação de conceitos adquiridos, desenvolvimento de capacidades de deteção e prevenção de CWE e a realização de análise e auditoria de falhas, erros, comportamentos desviantes, exposições presentes e potenciais impactos da exploração maliciosa destas.

Foi possível implementar com sucesso duas versões do website referido: uma vulnerável com a aplicação de CWE e uma corrigida com funcionalidades de segurança de forma. Todo o presente trabalho realizado encontra-se disponível em `https://github.com/detiuaveiro/1st-project-group_26`.

Índice

Lista de Figuras

Acrónimos

UA Universidade de Aveiro

DETI Departamento de Eletrónica, Telecomunicações e Informática

LECI Licenciatura em Engenharia de Computadores e Informática

UC Unidade Curricular

SIO Segurança Informática e Nas Organizações

CWE Common Weakness Enumeration

CIA Confidenciality, Integrity and Availability

PC Personal Computer

WWW World Wide Web

Capítulo 1

Introdução

O presente projecto é dedicado ao desenvolvimento de um website no estilo de loja online, concebido com o objectivo de promover e vender merchandise do Departamento de Eletrónica, Telecomunicações e Informática (DETI) da Universidade de Aveiro (UA) , incorporando abordagens essenciais da Segurança Informática e aplicando conceitos fundamentais lecionados no presente ano lectivo no decorrer da Unidade Curricular (UC) de Segurança Informática e Nas Organizações (SIO) do curso de Licenciatura em Engenharia de Computadores e Informática (LECI), com especial foco em:

1. Previsibilidade constante do sistema e solução apresentada (estratégia defensiva).
2. Defesa contra atividades não autorizadas (garantindo o funcionamento seguro e ininterrupto da plataforma).
3. Conceitos de Confidenciality, Integrity and Availability (CIA) da Segurança da Informação.
4. Conceitos de Common Weakness Enumeration (CWE)

1.1 Enquadramento

Com a introdução dos conceitos de Personal Computer (PC) e World Wide Web (WWW) à sociedade e público geral, foram introduzidos diversos benefícios e desafios de segurança. A dependência dos sistemas digitais complexos é um paradigma da sociedade contemporânea tornando assim imperativa a aprendizagem de conceitos fundamentais relativos à proteção de informações pessoais e confidenciais, e à garantia do funcionamento previsto (e seguro) dos sistemas e soluções digitais apresentadas ao público comum.

Este enquadramento destaca a relevância do projeto ao abordar questões cruciais, tais como:

- A necessidade de garantir a segurança de sistemas e informações num ambiente digital.
- A aplicação prática dos conceitos de SIO adquiridos.
- O compromisso com a proteção dos dados pessoais e/ou confidenciais dos utilizadores.
- O desenvolvimento de competências de detecção e prevenção de vulnerabilidades comuns CWE.
- A realização de análise e auditoria para identificar potenciais falhas, erros e exposições, assim como identificar e avaliar o seu impacto caso seja realizado um ataque com base em cada um destes.

Capítulo 2

Funcionamento Previsto do Website

2.1 Home Page

Página inicial do site da loja online. Serve como ponto de entrada para os utilizadores. Fornece uma visão geral dos produtos e serviços oferecidos pela loja, além de direcionar os clientes para as secções relevantes do site.

2.2 Pesquisa

Funcionalidade que permite aos utilizadores procurar produtos específicos ou categorias de produtos dentro do inventário da loja.

2.3 Sign up

Secção destinada ao registo de novos utilizadores na plataforma. Requer informações básicas, como nome, e-mail, password, entre outras.

2.4 Login

Secção destinada ao login de utilizadores já existentes na plataforma. Requer e-mail e password, previamente registados pelo utilizador em questão.

2.5 Customer Area

Secção personalizada para cada cliente, na qual é possível aceder a informações pessoais, histórico de compras, configurações de conta e detalhes de perfil.

Shopping Cart

Secção que permite aos clientes adicionar itens que desejam comprar antes de finalizar a compra.

Safe Cart

Secção que permite aos clientes adicionar items que desejam comprar no futuro.

Wishlist

Secção que permite aos clientes adicionar items que desejam, mas que não têm intenções de comprar.

Pagamentos

Secção dedicada aos métodos de pagamento aceites pela loja online. Inclui opções como MB WAY, referência multibanco e PayPal.

Capítulo 3

Vulnerabilidades e Correcções

Uma aplicação web permite permite que a interação com o utilizador comum seja efectuada através do *input* informação. Assim, é necessário que o desenvolvimento da mesma seja efectuado de forma cuidada com a aplicação de conceitos, paradigmas e mecanismos de segurança conhecidos que, atuando como uma camada de protecção, evitam a execução de potenciais ações maliciosas (ou divergentes do expectado) por parte do utilizador.

O sistema de categorização Common Weakness Enumeration (CWE) é uma coleção de mais de 600 categorias estruturadas de vulnerabilidades de hardware e software, sustentado por um projeto comunitário com o objetivo de compreender falhas em software e hardware e criar ferramentas automatizadas que possam ser usadas para identificar, corrigir e prevenir essas falhas.

Este trabalho pretende a aplicação, exploração e correcção de algumas CWE que foram consideradas mais relevantes, numa aplicação web do estilo *Online Store* com duas versões:

1. **app**: Aplicação insegura onde são aplicadas vulnerabilidades CWE de forma propositada para efeitos de estudo académico, sendo posteriormente explorada e manipulada de forma maliciosa de forma meramente demonstrativa.
2. **app_sec**: Aplicação segura, onde são feitas alterações nas vulnerabilidades aplicadas anteriormente, servindo o presente relatório como documento elucidativo das alterações efetuadas e da eficácia das mesmas para reforçar a segurança.

Foram aplicadas as seguintes vulnerabilidades à aplicação insegura *app* :

- 3.1 CWE-20: Improper Input Validation
- 3.2 CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- 3.3 CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- 3.4 CWE-200: Exposure of Sensitive Information to an Unauthorized Actor
- 3.5 CWE-256: Plaintext Storage of a Password
- 3.6 CWE-284: Improper Access Control

- 3.7 CWE-307: Improper Restriction of Excessive Authentication Attempts
- 3.8 CWE-311: Missing Encryption of Sensitive Data
- 3.9 CWE-319: Cleartext Transmission of Sensitive Information
- 3.10 CWE-472: External Control of Assumed-Immutable Web Parameter
- 3.11 CWE-521: Weak Password Requirements
- 3.12 CWE-541: Inclusion of Sensitive Information in an Include File
- 3.13 CWE-549: Missing Password Field Masking
- 3.14 CWE-620: Unverified Password Change
- 3.15 CWE-710: Improper Adherence to Coding Standards e 3.15 CWE-1006: Bad Coding Practices
- 3.16 Vulnerabilidade subtil

3.1 CWE-20: Improper Input Validation

A validação de input é uma técnica frequentemente usada para a verificação de inputs potencialmente perigosos, assegurando que o seu comportamento é o expectado e não prejudicial para processamento no interior de código ou durante a comunicação com outros componentes da aplicação.
Quando a validação não é realizada de forma cuidada, um atacante pode criar um input não expectado pela aplicação, fazendo com que partes do sistema aplicacional recebam entradas não intencionais, permitindo a realização de ações não autorizadas que comprometem a integridade dos dados ou até a exploração de outras vulnerabilidades da aplicação.

Demonstração e Impacto (app)

Aplicada ao projecto em questão, podemos verificar que esta vulnerabilidade encontra-se presente no código em diversas funções e rotas da base de dados da app.py (vulnerável). Para fins demonstrativos, existem os códigos da rota **search_product()** demonstrado na Figura 1 a baixo, onde podemos verificar a presença de uma *query*, que posteriormente é executada sem ser tratada (validação ou parametrização) e é inserida directamente na *search_query*, o que constitui uma vulnerabilidade, podendo resultar em outras vulnerabilidades (ou subtipos desta) como:

- **CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')**
- **CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')**

```python
@app.route('/search_product', methods=['POST'])
def search_product():
    if request.method == 'POST':
        search_query = request.form.get('query')
        con, cur = connect_db()

        query = f"SELECT * FROM products WHERE descrip LIKE '%{search_query}%';"

        cur.execute(query)
        products = cur.fetchall()
        con.close()

    return render_template('search_results.html', products=products)
```

Figura 1: CWE-20: Versão Insegura: @app.route search_product() - Base de Dados(app.py)

Correcção (app_sec):

Para corrigir esta vulnerabilidade e evitar futuras ocorrências da mesma, migrámos do uso de SQLite, que não oferece proteções robustas contra injeção de SQL, para SQLAlchemy.

Esta abordagem é mais segura, SQLAlchemy lida garante a separação segura dos dados do utilizador e do código SQL em queries, protegendo a aplicação contra a injeção de SQL e outras vulnerabilidades relacionadas à entrada de dados.

Foi usada a bibioteca *"bleach"*, que sanitiza os inputs dos utilizadores e foi efectuada a parametrização das *queries* recorrendo à função *"text"* de SQLAlchemy, prevenindo com maior eficácia assim ataques **XSS** ou **SQL Injection**, tal como demonstrado na Figura 2.

```python
@app.route('/search_product', methods=['POST'])
def search_product():
    if request.method == 'POST':
        search_query = bleach.clean(request.form.get('query'), strip=True)
        query = text("SELECT * FROM products WHERE descrip LIKE :search_query")

        products = db_session.execute(query, {"search_query": f"%{search_query}%"})
        return render_template('search_results.html', products=products)
```

Figura 2: CWE-20: Versão Segura: @app.route search_product() - Base de Dados(app_sec.py)

3.2 CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Conhecida por *"Cross-site Scripting"* ou *XSS*), esta vulnerabilidade ocorre quando uma aplicação não consegue neutralizar adequadamente os inputs efectuados deliberadamente pelo utilizador antes de os usar como output, sendo efectuado o display deste output a outros utilizadores, compromentendo

assim a integridade e segurança da solução web, permitindo aos atacantes injetar código malicioso, como JavaScript, nos navegadores de outros utilizadores.
Como tal é importante seguir boas práticas de segurança e programação no desenvolvimento de aplicações web, minimizando o risco de uma exploração *"Cross-site Scripting"*, por forma a proteger os utilizadores finais contra potenciais ameaças.
Existem três tipos principais de XSS:

- **Reflected *XSS* (Não Persistente):** O servidor lê diretamente os dados a partir do pedido HTTP e reflete-os na sua resposta HTTP. Ocorre quando um atacante "força"(indirectamente) uma vítima a fornecer conteúdo perigoso a uma aplicação web vulnerável, que é posteriormente refletido de volta para a vítima e executado pelo navegador web. O mecanismo mais comum para a entrega de conteúdo malicioso é incluí-lo como um parâmetro numa URL que é publicada ou enviada diretamente à vítima, sendo que não é armazenada em Base de Dados (não persistente).
- **Stored *XSS* (Persistente):** Um atacante armazena dados perigosos na base de dados usando mensagens em forums, comentários, ou outras áreas exibida a muitos utilizadores. Posteriormente, os dados perigosos são "lidos"pela aplicação e incluídos em conteúdo dinâmico que, ao ser executado por um utilizador, permite ao atacante realizar operações privilegiadas em nome do utilizador (ou aceder a dados sensíveis deste).
- **DOM-Based *XSS*:** Neste caso é o utilizador que realiza a injeção de XSS na página web *(client sided)*. Geralmente envolve script confiável controlado pelo servidor que é enviado para o cliente. Se o script fornecido pelo servidor processar dados fornecidos pelo utilizador e depois os injetar de volta na página web, então o DOM-Based XSS é possível.

Demonstração e Impacto (app)

Esta vulnerabilidade está presente nas funções, **login()**, **rate_comment_product_route()**, **add_product()**, **search_product()**, entre outras. Todas estas funções estão suscetíveis a Cross-Site-Scripting, devido à falta de validação de dados inseridos pelo utilizador, levando à possibilidade de inserção de scripts maliciosos. A função **rate_comment_product_route()**, como podemos observar pelas Figuras 3, 6, 5 e **??** são um excelente exemplo de como esta vulnerabilidade poderia afetar o website, visto que permite exibição de conteúdos indesejados para todos os utilizadores.

```python
def rate_comment_product(product_id, user_id, rating, comment):

    con, cur = connect_db()
    try:
        timestamp = datetime.datetime.now().strftime("%c") # Ex: Wed Nov 01 16:03:43 2023
        cur.execute(
            f"""INSERT INTO product_ratings (
                product_id, user_id, rating, comment, timestamp) VALUES ({product_id}, {user_id}, {rating}, '{comment}', '{timestamp}')"""
        )

        cur.execute(
            f"""INSERT INTO product_comments (
                product_id, user_id, comment, timestamp) VALUES ({product_id}, {user_id}, '{comment}', '{timestamp}')"""
        )

        con.commit()
        con.close()
        msg = "Thank your for rating and commenting our product!"
        return True, msg

    except Exception as e:
        error = f"Error: {str(e)}"
        return False, error
```

Figura 3: CWE-79: Versão Insegura: @app.route rate_comment_product_route() - Base de Dados(app.py)

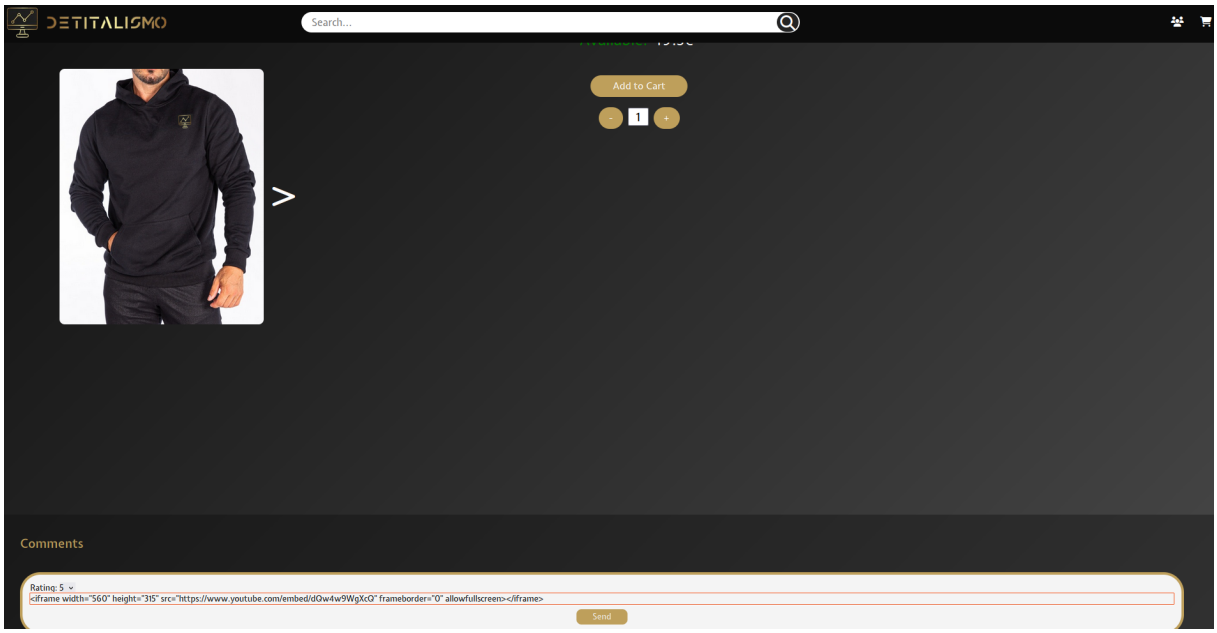


Figura 4: CWE-79: Versão Insegura: Comentário JS em Front-end (Product: Hoodie)

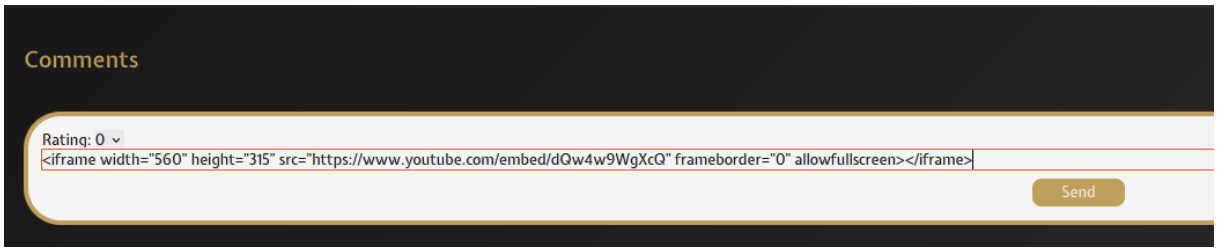


Figura 5: CWE-79: Versão Insegura: Comentário JS em Front-end (Product: Hoodie)

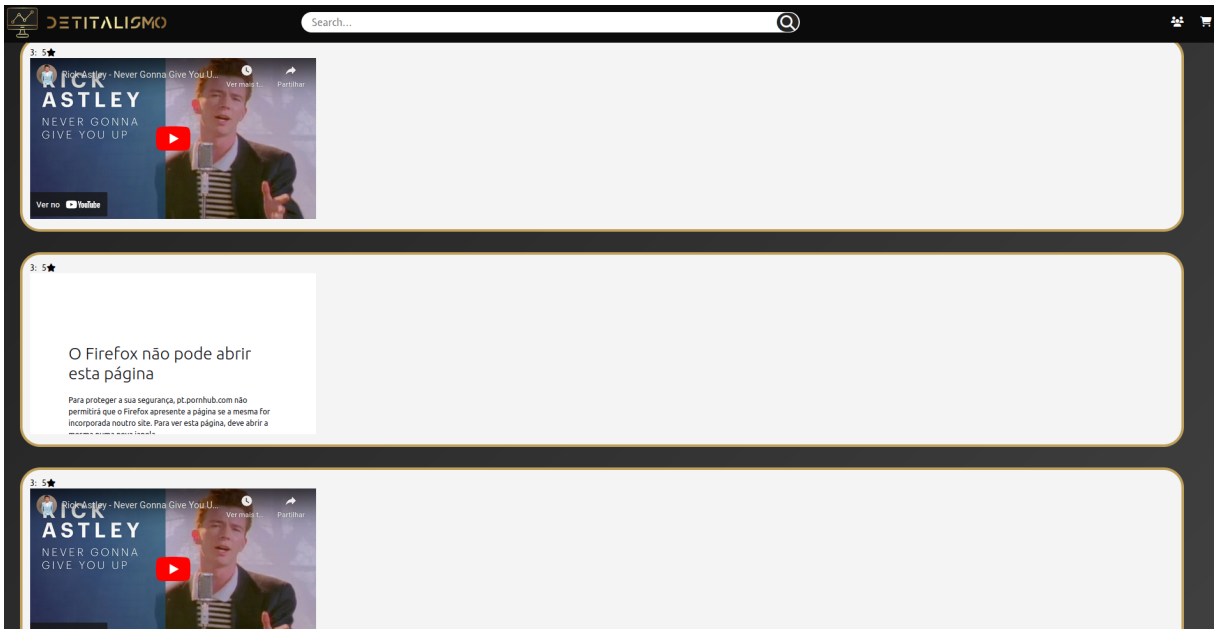


Figura 6: CWE-79: Versão Inesgura: Resultado XSS Attack Front-end (Product: Hoodie)

Correcção (app_sec):

Para corrigir esta vulnerabilidade, foi usada a bibioteca de **Python** *"bleach"*, que sanitiza os inputs dos utilizadores e foi efectuada a parametrização das *queries* recorrendo à função *"text"* de SQLAlchemy, prevenindo com maior eficácia ataques de **XSS**, tal como demonstrado na Figura 7.

```python
@app.route('/rate_comment_product', methods=['POST'])
def rate_comment_product_route():
    if 'user_id' in session:
        user_id = session['user_id']
        if request.method == 'POST':
            data = request.json
            product_id = bleach.clean(data.get('product_id'), strip=True)
            rating = bleach.clean(data.get('rating'), strip=True)
            comment = bleach.clean(data.get('comment'), strip=True)
            if not (product_id and rating and comment):
                return jsonify({'error': 'Insert product_id, rating and comment'})
            success, message = rate_comment_product(product_id, user_id, rating, comment)
            if success:
                return jsonify({'message': message})
            else:
                return jsonify({'error': message})
    else:
        return jsonify({'error': 'Please log in first!'})
```

Figura 7: CWE-79: Versão Segura: @app.route rate_comment_product_route() - Base de Dados(app_sec.py)

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

Também conhecida por *"SQL Injection"*, esta vulnerabilidade ocorre quando se pede a um usuário que forneça um input, como o username, e em vez do username é inserida uma declaração SQL que irá atacar a base de dados. Esta vulnerabilidade permite que o utilizador leia, modifique e elimine dados sensíveis, comprometendo assim a integridade dos mesmos. No contexto do nosso projeto, a aplicação insegura é vulnerável a injeções de SQL, pois não foi realizada a devida prévia validação e saneamento dos dados introduzidos pelos utilizadores.

Demonstração e Impacto (app)

Aplicada ao projeto em questão, podemos verificar que esta vulnerabilidade encontra-se presente no código, na maioria das funções e rotas da base de dados da app vulnerável. Esta occorre quando o input do utilizador é utilizado diretamente numa declaração SQL. As rotas nas quais esta vulnerabilidade pode ser encontrada são:

- **register()**
- **login()**
- **change_password()**
- **mecart()**
- **upCart()**
- **rate_comment_product_route()**
- **search_product()**
- **get_items()**
- **add_product_route()**

```python
@app.route('/register', methods=['POST'])
def register():

    con, cur = connect_db()
    data = request.json  # Extrair os dados JSON da solicitação

    # Extrair os valores do dicionário de dados
    fname = data.get('FName')
    lname = data.get('LName')
    email = data.get("Email")
    passwd = data.get("Password")
    phone = data.get("Phone")
    nif = data.get("Nif")

    # Verificar Campos
    cur.execute("SELECT EMAIL FROM USERS")
    emailList = cur.fetchall()
    for i in emailList:
        if email == i[0]:
            return jsonify(False)

    # Adicionar cliente ao banco de dados
    cur.execute("INSERT INTO USERS (FNAME, LNAME, EMAIL, PASSWORD, PHONE, NIF, ROLE) VALUES ('{}', '{}', '{}', '{}', '{}', '{}', 'Customer')".format(fname, lname, email, passwd, phone, nif))
    cur.execute("SELECT ID FROM USERS WHERE EMAIL = '{}'".format(email))
    user_id = cur.fetchone()[0]
    cur.execute("INSERT INTO CART (USERID, ITEM0, ITEM1, ITEM2, ITEM3, ITEM4, ITEM5, ITEM6, ITEM7, ITEM8, ITEM9, ITEM10, ITEM11, TOTAL) VALUES ('{}', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0')
    cur.execute("INSERT INTO WISHLIST (USERID, ITEM0, ITEM1, ITEM2, ITEM3, ITEM4, ITEM5, ITEM6, ITEM7, ITEM8, ITEM9, ITEM10, ITEM11) VALUES ('{}', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0')".format
    con.commit()
    cur.close()
    con.close()

    return jsonify(True)
```

Figura 8: CWE-89: Versão Insegura: @app.route register() - Base de Dados(app.py)

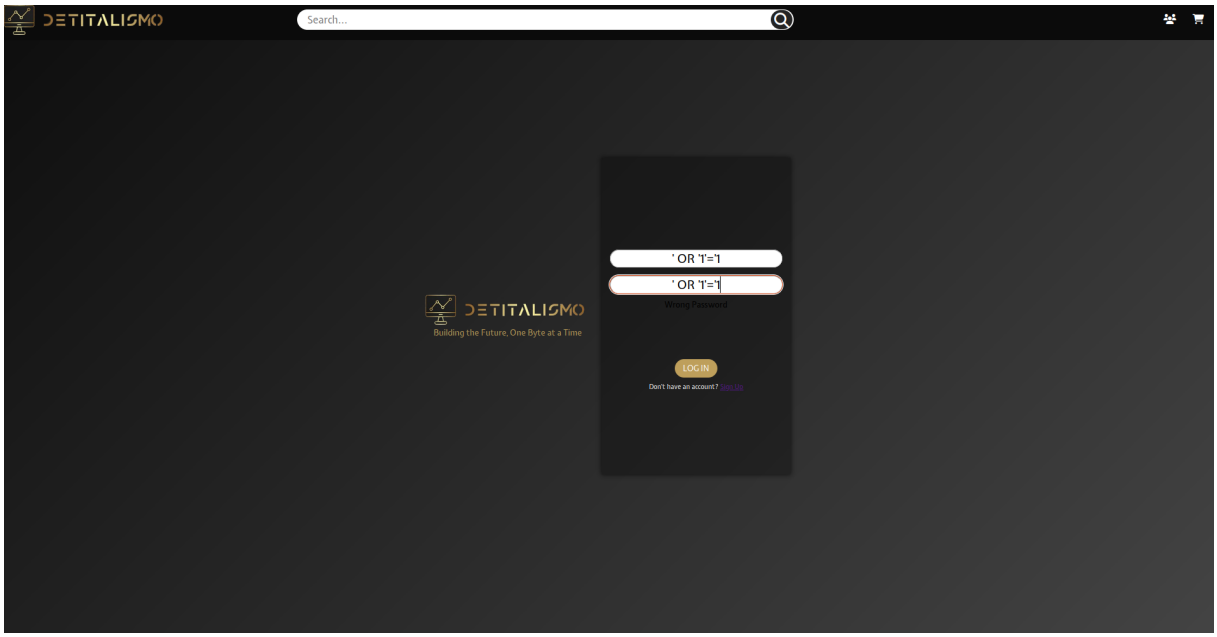


Figura 9: CWE-89: Versão Insegura: User Interface Front-End: Tentativa SQL Injection

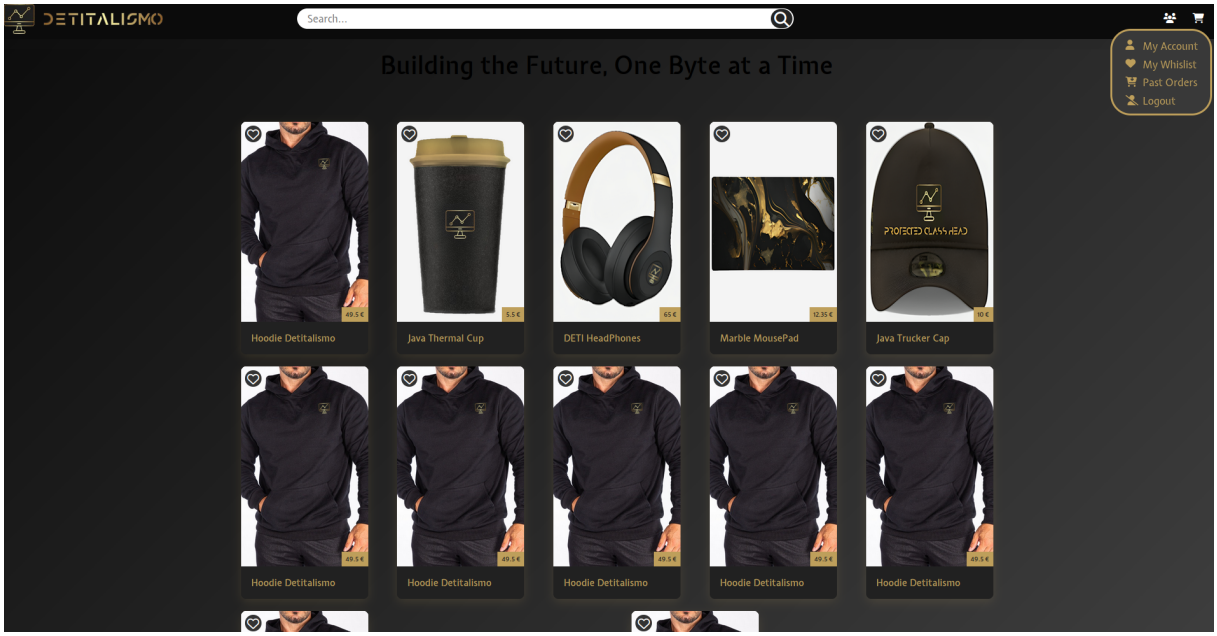


Figura 10: CWE-89: Versão Insegura: User Interface Front-End: Sucesso de SQL Injection

Como podemos observar pelas Figuras 8 e 9, não existe qualquer tipo de validação e saneamento dos dados introduzidos pelo utilizador, sendo estes utilizados numa declaração SQL.

Correcção (app_sec):

Para corrigir esta vulnerabilidade, foi usada a bibioteca de **Python** *"bleach"*, que sanitiza os inputs dos utilizadores e foi efectuada a parametrização das *queries* recorrendo à função *"text"* de SQLAlchemy, prevenindo com maior eficácia assim ataques ou **SQL Injection**, tal como demonstrado na Figura 11.

```python
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        fname = bleach.clean(request.form['fname'], strip=True)
        lname = bleach.clean(request.form['lname'], strip=True)
        email = bleach.clean(request.form['email'], strip=True)
        nick = bleach.clean(request.form['nick'], strip=True)
        password = bleach.clean(request.form['password'], strip=True)
        confirm = bleach.clean(request.form['confirm'], strip=True)

        if password != confirm:
            flash("Passwords do not match.", 'error')
        else:
            existing_user = db_session.query(Customer).filter_by(email=email).first()
            if existing_user:
                flash("Email or nickname is already in use.", 'error')
            else:
                # Hash de passwords!!! :)
                hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
                user = Customer(fname=fname, lname=lname, email=email, nick=nick, passwd=hashed_password)
                db_session.add(user)
                db_session.commit()
                flash("Registration complete!", 'SUCESS')
                return redirect('/login')

    return render_template('signup.html')
```

Figura 11: CWE-89: Versão Segura: @app.route register() - Base de Dados(app_sec.py)

3.4 CWE-200: Exposure of Sensitive Information to an Unauthorized Actor

A exposição de informações sensíveis a atores não autorizados é uma vulnerabilidade grave que pode ocorrer quando dados confidenciais são acessíveis por indivíduos não autorizados. Pode resultar em:

- Violações de privacidade
- Roubo de identidade
- Perda de propriedade intelectual
- Outros impactos prejudiciais

Neste projecto, torna-se especialmente relevante ao no front-end da aplicação, onde dá *feedback* aos utilizadores sobre que campo (email ou password) se encontra incorreto durante uma tentativa de login incorrecta, revelando mais informações do que seria necessário, permitindo que um possível atacante explore com mais facilidade uma entrada não autorizada por tentativa/erro.

Demonstração e Impacto (app)

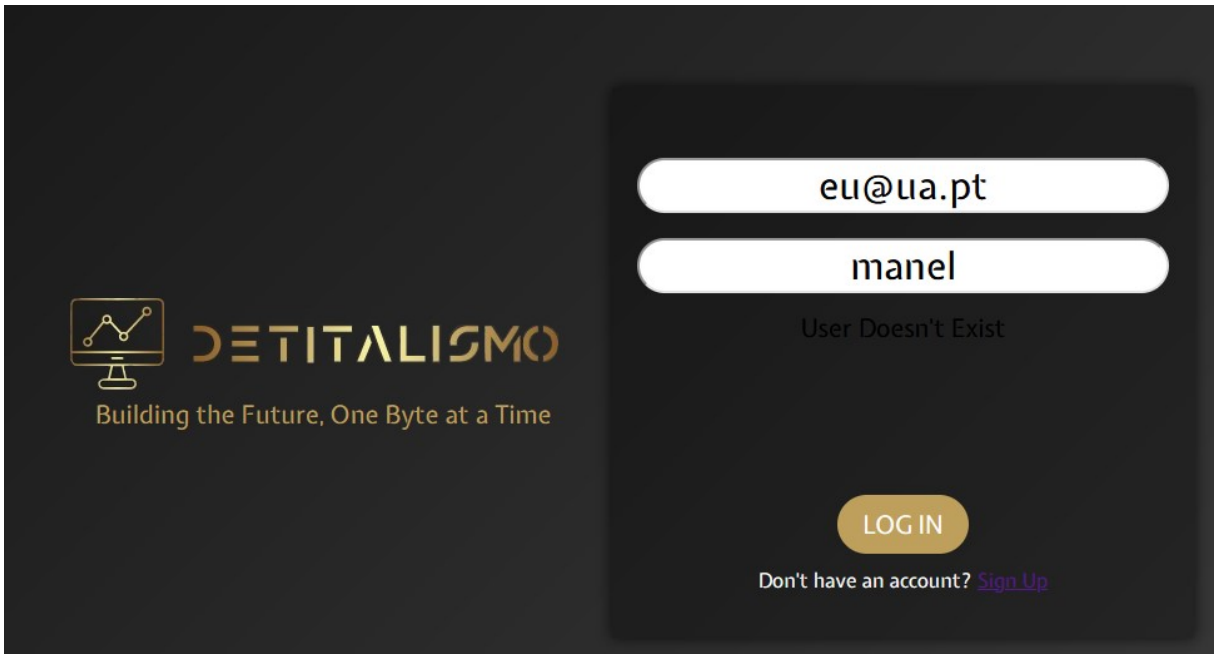


Figura 12: CWE-200: Versão Insegura: "User doesn't exist": User Interface Front-end Login

Figura 13: CWE-200: Versão Insegura: "Wrong Password": User Interface Front-end Login

```
<form className='login-form' onSubmit = {handleSubmit}>
    <fieldset className="form-group">
        <input className="form-input" value={email} onChange={(e) => setEmail(e.target.value)} type="email" name="email" id="email" placeholder='E-mail' />
    </fieldset>
    <fieldset className="form-group">
        <input className="form-input1" value={password} onChange={(e) => setPassword(e.target.value)} type="text" name="password" id="password" placeholder='Passwor
    </fieldset>
</form>
{invLogin2 && <p className='errorp'> Wrong Password</p>}
{invLogin1 && <p className='errorp'> User Doesn't Exist</p>}
<button className='login-btn' onClick={handleSubmit}>LOG IN</button>
```

Figura 14: CWE-200: Versão Insegura: Código inseguro Front-end Login

Correcção (app_sec):

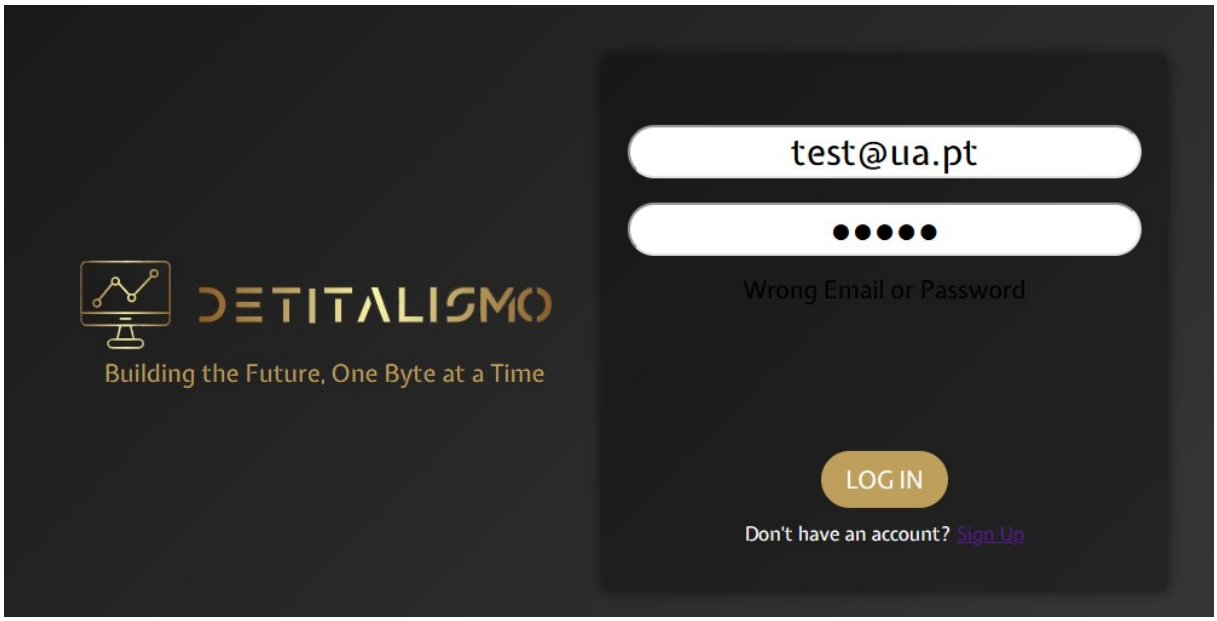


Figura 15: CWE-200: Versão Segura corrigida: Front-end Login

```
<section className='login-container'>
    <section className='login-logo'>
        <Link to="/">
            <img src="/Images/DETItalismo.png" alt="logo" height={60}/>
    (property) JSX.IntrinsicElements.section: React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLElement>
    </section>
    <section className='login-form-container'>

        <form className='login-form' onSubmit = {handleSubmit}>
            <fieldset className="form-group">
                <input className="form-input" value={email} onChange={(e) => setEmail(e.target.value)} type="email" name="email" id="email" placeholder='E-mail' />
            </fieldset>
            <fieldset className="form-group">
                <input className="form-input1" value={password} onChange={(e) => setPassword(e.target.value)} type="password" name="password" id="password" placeholder='Pas
            </fieldset>
        </form>
        {invLogin2 && <p className='errorp'> Wrong Email or Password</p>}
        {invLogin1 && <p className='errorp'> User Doesn't Exist</p>}
        <button className='login-btn' onClick={handleSubmit}>LOG IN</button>
        <p className='signUp-Q'>Don't have an account? <Link to="/Register">Sign Up</Link></p>
    </section>
```

Figura 16: CWE-200: Versão Segura: Código Corrigido Front-end Login

3.5 CWE-256: Plaintext Storage of a Password

A vulnerabilidade CWE-256, é uma vulnerabilidade comum que se refere aos problemas que ocorrem quando password são armazenadas em texto simples nas propriedades da aplicação, nos ficheiros de configuração ou na memória. Esta torna as password acessíveis a qualquer entidade com acesso aos ficheiros onde estas estão guardadas. A mitigação desta vulnerabilidade requer a utilização de algoritmos de hash criptográfico sobre as password e armazenamento do hash resultante.

Demonstração e Impacto (app)

No projecto, esta vulnerabilidade é evidente através da análise do código responsável pela construção das tabelas de utilizadores e sua inserção (Nota: utilizadores são chamados de *customers*) e da visualização da base de dados em questão, tal como demonstrado nas Figuras 17 e 18 seguintes:

```
#
# ------------------------------------------------------------------------------ #
# ---------------- TABLE: CUSTOMERS ------------------------------------------- #
# -----                                                                    ----- #
    cur.execute(
        """CREATE TABLE IF NOT EXISTS USERS (
        ID          INTEGER                         PRIMARY KEY     ,
        FNAME       VARCHAR(32)     NOT NULL                        ,
        LNAME       VARCHAR(32)     NOT NULL                        ,
        EMAIL       VARCHAR(64)     NOT NULL        UNIQUE          ,
        PASSWORD    VARCHAR(32)     NOT NULL                        ,
        PHONE       INTEGER         NOT NULL                        ,
        NIF         DECIMAL                                         ,
        ROLE        VARCHAR(16)     NOT NULL                        );
        """
    )
```

Figura 17: CWE-256: Versão Insegura: Tabela e Inserts de "Customers- Base de Dados(app.py)

	user_id	fname	lname	email	nick	passwd	nib
1	1	João	Vieira	joaopvieira@ua.pt	JPNV	password50458	
2	2	Miguel	Silva	miguel.silva@gmail....	MiguelSilva	Querty1234	
3	3	Ana	Ferreira	anaferreira96@gma...	Ferrero	AsDf1996	
4	4	Raul	Riberio	raul_rib15@gmail.com	Guillette	TheBestAM@nCanG...	
5	5	André	Pereira	pereirandre84@ua.pt	Pereirinha	FuiA0J#rd1md@C3l...	

Figura 18: CWE-256: Versão Insegura: Interior da Base de Dados criada (DOSG26.db)

Como se pode verificar, as *passwords* dos utilizadores inseridos são guardadas (de forma incorrecta/vulnerável) em texto (plaintext), tal como dita a vulnerabilidade.

Correcção (app_sec):

Para corrigir este problema, foram executadas as seguintes alterações demonstradas nas Figuras 19 e 20:

```
for data in customers_data:

    hashed_password = bcrypt.hashpw(data["passwd"].encode('utf-8'), bcrypt.gensalt(rounds=2**5))

    data["passwd"] = hashed_password
    new_customers = Customer(**data)
    session.add(new_customers)
```

Figura 19: CWE-256: Versão Segura: Hashing de Passwords antes de "insert- Base de Dados(db_inserts.py)

	user_id	fname	lname	email	nick	passwd
	Search column...	Search column...	Search column...	Search column...	Search column...	Search column...
1	1	João	Vieira	joaopvieira@ua.pt	JPNV	60 Bytes
2	2	Miguel	Silva	miguel.silva@gmail....	MiguelSilva	60 Bytes
3	3	Ana	Ferreira	anaferreira96@gma...	Ferrero	60 Bytes
4	4	Raul	Ribeiro	raul_rib15@gmail.com	Guillette	60 Bytes
5	5	André	Pereira	pereirandre84@ua.pt	Pereirinha	60 Bytes

Figura 20: CWE-256: Versão Segura: Interior da Base de Dados criada (DOSG26_SEC.db)

Como se pode verificar, recorrendo à função hash criptográfica *"bcrypt"* e metodos de *"salting"* (com 2^5 iterações/rounds, para maior segurança), para corrigir o problema.

3.6 CWE-284: Improper Access Control

A vulnerabilidade CWE-284, refere-se ao acesso inadequado. Ocorre quando não é feito o devido controlo ao acesso de recursos sensíveis. Pode levar a acessos não autorizados, representando uma ameaça significativa para a integridade dos sistemas e dos seus dados. Para mitigar esta vulnerabilidade, é fundamental implementar uma política de controle de acesso bem definida, que inclua uma forte autenticação.

Demonstração e Impacto (app)

No contexto do nosso projeto, podemos verificar que esta vulnerabilidade encontra-se presente no código. Apesar de na base de dados se poder armazenar Admins e Customers, existem funções que deviam ser restritas apenas a admins, no entanto não existe qualquer verificação para o tipo de utilizador (Customer ou Admin) que está a fazer o request.

```python
def add_product(title, description, category, price, stock):
    try:
        con, cur = connect_db()
        query = f"""
        INSERT INTO products (title, descrip, category, price, stock)
        VALUES ('{title}', '{description}', '{category}', '{price}', '{stock}')
        """
        cur.execute(query)

        con.commit()
        con.close()

        return "New product added!"
    except Exception as e:
        return f"Error: {str(e)}"
```

Figura 21: CWE-284: Versão Insegura: @app.route add_product_route() - Base de Dados(app.py)

```python
def change_product(product_id, new_title, new_description, new_category, new_price, new_stock):
    try:
        con, cur = connect_db()

        query = f"""
        UPDATE products
        SET title = '{new_title}', descrip = '{new_description}', category = '{new_category}', price = '{new_price}', stock = '{new_stock}'
        WHERE product_id = '{product_id}'
        """
        cur.execute(query)

        con.commit()
        con.close()

        return "Product updated!"
    except Exception as e:
        return f"Error: {str(e)}"
```

Figura 22: CWE-284: Versão Insegura: @app.route change_product_route() - Base de Dados(app.py)

Como podemos observar pela Figura 21 e 22, qualquer utilizador, Customer ou Admin consegue adicionar e alterar os produtos do nosso website.

Correcção (app_sec):

Esta vulnerabilidade foi corrigida através da verificação do campo "Role", presente em todos os utilizadores. Assim existem funções restritas apenas aos Admins.

```python
# ROTA: ADD PRODUCT (ADMIN ONLY)
@app.route('/add_product', methods=['POST'])
def add_product_route():
    if 'user_id' in session:
        user_id = session['user_id']
        user = db_session.query(User).filter_by(user_id=user_id).first()
        if user.role == "Admin":
            if request.method == 'POST':
                title = bleach.clean(request.form.get('new_title'), strip=True)
                description = bleach.clean(request.form.get('new_description'), strip=True)
                category = bleach.clean(request.form.get('new_category'), strip=True)
                price = bleach.clean(request.form.get('new_price'), strip=True)
                stock = bleach.clean(request.form.get('new_stock'), strip=True)

                success, message = add_product(db_session, title, description, category, price, stock)
                if success:
                    return jsonify({'message': message})
                else:
                    return jsonify({'error': message})
            else:
                return jsonify({'error': 'Invalid request method.'})
        else:
            return jsonify({'error': 'Only Admins can access this method.'})
    else:
        return jsonify({'error': 'Please log in first!'})
```

Figura 23: CWE-284: Versão Segura: @app.route add_product_route() - Base de Dados(app_sec.py)

```python
# ROTA: CHANGE PRODUCT (ADMIN ONLY)
@app.route('/change_product/<int:product_id>', methods=['POST'])
def change_product_route(product_id):
    if 'user_id' in session:
        user_id = session['user_id']
        user = db_session.query(User).filter_by(user_id=user_id).first()
        if user.role == "Admin":
            if request.method == 'POST':
                new_title = bleach.clean(request.form.get('new_title'), strip=True)
                new_description = bleach.clean(request.form.get('new_description'), strip=True)
                new_category = bleach.clean(request.form.get('new_category'), strip=True)
                new_price = bleach.clean(request.form.get('new_price'), strip=True)
                new_stock = bleach.clean(request.form.get('new_stock'), strip=True)

                success, message = change_product(db_session, product_id, new_title, new_description, new_category, new_price, new_stock)
                if success:
                    return jsonify({'message': message})
                else:
                    return jsonify({'error': message})
            else:
                return jsonify({'error': 'Invalid request method.'})
        else:
            return jsonify({'error': 'Only Admins can access this method.'})
    else:
        return jsonify({'error': 'Please log in first!'})
```

Figura 24: CWE-284: Versão Segura: @app.route change_product_route() - Base de Dados(app_sec.py)

3.7 CWE-307: Improper Restriction of Excessive Authentication Attempts

A aplicação web não implementa medidas suficientes para evitar múltiplas tentativas de autenticação por um atacante num curto espaço de tempo, tornanda assim mais suscetível a ataques de força bruta, onde um atacante tenta incessantemente adivinhar passwords ou outras formas de credenciais de acesso.

Demonstração e Impacto (app)

```python
@app.route('/login', methods=['POST'])
def login():
    con, cur = connect_db()
    data = request.json  # Extract the JSON data from the request

    email = data.get('Email')
    passwd = data.get('Password')

    query = f"SELECT PASSWORD FROM USERS WHERE EMAIL = '{email}' AND PASSWORD = '{passwd}'"

    cur.execute(query)

    user_data = cur.fetchone()
    cur.close()
    con.close()

    if user_data:
        return jsonify(1)
    else:
        return jsonify(0)
```

Figura 25: CWE-307: Versão Insegura: @app.route login() - Base de Dados(app_sec)

Correcção (app_sec):

Primeiramente foi decidido implementar um contador de tentativas de login. Fazendo com que quando atingisse 5 tentativas falhadas seguidas, a conta fosse bloqueada e para recuperar seria necessário o contactar o suporte. Para tal foi adicionada à tabela da base de dados a failed_login_attemps ,figura 27, para o seu valor poder ser incrementado, figura 26.

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = bleach.clean(request.form['email'], strip=True)
        password = bleach.clean(request.form['password'], strip=True)

        user = db_session.query(User).filter_by(email=email).first()


        if user:
            if (user.failed_login_attempts >= 5):
                flash("You have exceeded the number of login attempts. Account locked, contact support", 'ERROR')
                return render_template('login.html')
            elif check_password_hash(user.passwd, password):
                session['user_id'] = user.user_id
                user.failed_login_attempts = 0
                flash("You have logged in successfully!", 'SUCESS')
                return redirect('/dashboard')
            else:
                user.failed_login_attempts += 1
                db_session.commit()
                flash("Login failed. Please check your credentials again!", 'ERROR')


    return render_template('login.html')
```

Figura 26: CWE-307: Versão Insegura: @app.route login() - Base de Dados(app_sec.py)

```
Henrique Cruz, há 6 horas | 2 authors (Luis Diogo and others) | CodiumAI: Options | Test this class
class User(Base):
    __tablename__ = "users"

    user_id = Column(Integer, primary_key=True)
    fname   = Column(String(32), nullable=False)
    lname   = Column(String(32), nullable=False)
    email   = Column(String(64), nullable=False, unique=True)
    nick    = Column(String(32), nullable=False, unique=True)
    passwd  = Column(String(64), nullable=False)
    nib     = Column(DECIMAL)
    role    = Column(String(16), nullable=False)
    failed_login_attempts = Column(Integer, default=0)

    # User Constructor:
    CodiumAI: Options | Test this method
    def __init__(self, fname, lname, email, nick, passwd, nib, role):
        self.fname   = fname
        self.lname   = lname
        self.email   = email
        self.nick    = nick
        self.passwd  = passwd
        self.nib     = nib
        self.role    = role
        self.failed_login_attempts = 0
```

Figura 27: CWE-307: Versão Insegura: - Base de Dados(db_tables.py)

No fim foi concluído que esta implementação criaria uma vulnerabilidade onde qualquer conta seria facilmente bloqueada. Então decidimos optar por implementar uma taxa de tentativas de login limitadas ,como é possivel ver na figura 28 nas linhas 2 e 3, solucionando assim a vulnerabilidade.

```
CodiumAI: Options | Test this function
@app.route('/login', methods=['GET', 'POST'])
@limiter.limit("5/minute")
@limiter.limit("20/day")
def login():
    if request.method == 'POST':
        email = bleach.clean(request.form['email'], strip=True)
        password = bleach.clean(request.form['password'], strip=True)

        user = db_session.query(User).filter_by(email=email).first()

        if user and check_password_hash(user.passwd, password):
            session['user_id'] = user.user_id
            user.failed_login_attempts = 0
            flash("You have logged in successfully!", 'SUCESS')
            return redirect('/dashboard')
        else:
            db_session.commit()
            flash("Login failed. Please check your credentials again!", 'ERROR')


    return render_template('login.html')
```

Figura 28: CWE-307: Versão Segura: @app.route login() - Base de Dados(app_sec.py)

3.8 CWE-311: Missing Encryption of Sensitive Data

Esta vulnerabilidade ocorre quando informações confidenciais, passwords, são armazenadas, transmitidas ou processadas sem a devida proteção criptográfica.
A falta de criptografia apropriada coloca em risco a confidencialidade e a integridade de dados, tornando-os vulneráveis a potenciais ameaças e violações de segurança.
Atacantes podem explorar essa vulnerabilidade para interceptar, aceder e/ou comprometer informações sensíveis, podendo resultar em roubo de identidade, fraude financeira, divulgação não autorizada de dados pessoais, etc...

Demonstração e Impacto (app)

No contexto do nosso projeto, podemos verificar que esta vulnerabilidade encontra-se presente no código. O armazenamento das credencias dos utilizadores, bem como outros dados sensíveis são armazenados sem qualquer cifragem, diretamente na base de dados.

```python
@app.route('/register', methods=['POST'])
def register():

    con, cur = connect_db()
    data = request.json  # Extrair os dados JSON da solicitação

    # Extrair os valores do dicionário de dados
    fname = data.get('FName')
    lname = data.get('LName')
    email = data.get("Email")
    passwd = data.get("Password")
    phone = data.get("Phone")
    nif = data.get("Nif")

    # Verificar Campos
    cur.execute("SELECT EMAIL FROM USERS")
    emailList = cur.fetchall()
    for i in emailList:
        if email == i[0]:
            return jsonify(False)

    # Adicionar cliente ao banco de dados
    cur.execute("INSERT INTO USERS (FNAME, LNAME, EMAIL, PASSWORD, PHONE, NIF, ROLE) VALUES ('{}', '{}', '{}', '{}', '{}', '{}', 'Customer')".format(fname, lname, email, passwd, phone, nif))
    cur.execute("SELECT ID FROM USERS WHERE EMAIL = '{}'".format(email))
    user_id = cur.fetchone()[0]
    cur.execute("INSERT INTO CART (USERID, ITEM0, ITEM1, ITEM2, ITEM3, ITEM4, ITEM5, ITEM6, ITEM7, ITEM8, ITEM9, ITEM10, ITEM11, TOTAL) VALUES ('{}', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0')
    cur.execute("INSERT INTO WISHLIST (USERID, ITEM0, ITEM1, ITEM2, ITEM3, ITEM4, ITEM5, ITEM6, ITEM7, ITEM8, ITEM9, ITEM10, ITEM11) VALUES ('{}', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0')".format
    con.commit()
    cur.close()
    con.close()

    return jsonify(True)
```

Figura 29: CWE-311: Versão Insegura: @app.route register() - Base de Dados(app.py)

No caso das credenciais dos admins, optámos por utilizar uma função hash com o algoritmo SHA-1, Figura para realizar a cifragem, como podemos obervar pela Figura 30. Contudo esta é insegura porque não incorpora "salting"ou qualquer outro mecanismo de proteção adicional. O processo de "salting"é uma técnica fundamental que acrescenta uma *string* aleatória e única à palavra-passe antes de aplicar a função de hash. Esta prática torna os hashes mais robustos, dificultando significativamente ataques de força bruta, por exemplo.

```python
def hash_password(password):
    sha1_hash = hashlib.sha1(password.encode('utf-8')).hexdigest()
    return sha1_hash
```

Figura 30: CWE-311: Versão Insegura: hash_password() - Base de Dados(app.py)

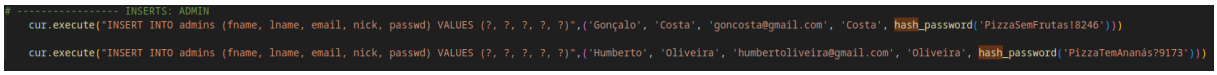


Figura 31: CWE-311: Versão Insegura: Inserir Credencias de Admins na Base de Dados - Base de Dados(app.py)

Correcção (app_sec):

Para corrigir esta vulnerabilidade, foi usada a bibioteca de **Python** *"bcrypt"*. Ao usar o *"bcrypt"*, geramos um "salt"aleatório para cada palavra-passe antes de aplicar a função de hash. Este "salt"é então armazenado juntamente com o hash resultante. A inclusão do "salt"aumenta a complexidade do processo de hash, tornando-o mais difícil de ser descoberta.

```python
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        fname = bleach.clean(request.form['fname'], strip=True)
        lname = bleach.clean(request.form['lname'], strip=True)
        email = bleach.clean(request.form['email'], strip=True)
        nick = bleach.clean(request.form['nick'], strip=True)
        password = bleach.clean(request.form['password'], strip=True)
        confirm = bleach.clean(request.form['confirm'], strip=True)

        if password != confirm:
            flash("Passwords do not match.", 'error')
        else:
            existing_user = db_session.query(Customer).filter_by(email=email).first()
            if existing_user:
                flash("Email or nickname is already in use.", 'error')
            else:
                # Hash de passwords!!! :)
                hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
                user = Customer(fname=fname, lname=lname, email=email, nick=nick, passwd=hashed_password)
                db_session.add(user)
                db_session.commit()
                flash("Registration complete!", 'SUCESS')
                return redirect('/login')

    return render_template('signup.html')
```

Figura 32: CWE-311: Versão Segura: @app.route signup() - Base de Dados(app_sec.py)

3.9 CWE-319: Cleartext Transmission of Sensitive Information

Aplicações não devem transferir informações confidenciais ou sensíveis, sem a devida cifragem ou proteção.
No contexto da segurança de software, a CWE-319 realça os riscos associados ao envio de dados sensíveis em texto simples, tornando-os suscetíveis a escutas e interseções por parte de atacantes.

Demonstração e Impacto (app)

Para demonstração desta vulnerabilidade, pode-se analisar a rota de *change_password* apresentada na Figura seguinte:

```python
@app.route('/change_password', methods=['POST'])
def change_password():
    con, cur = connect_db()
    data = request.json

    email = data[0]
    new_password = data[1]

    if '%40' in email:
        email = email.replace('%40', '@')


    cur.execute("SELECT EMAIL FROM USERS")
    emailList = cur.fetchall()
    for i in emailList:
        if email == i[0]:
            cur.execute(f"SELECT PASSWORD FROM USERS WHERE EMAIL = '{email}'")
            cur.execute(f"UPDATE USERS SET PASSWORD = '{new_password}' WHERE EMAIL = '{email}'")
            con.commit()
            cur.close()
            con.close()
            return jsonify(1)
        else:
            return jsonify(2)
    cur.close()
    con.close()
    return jsonify(0)
```

Figura 33: CWE-319: Versão Insegura: @app.route change_password() - Base de Dados(app.py)

Nesta rota, é visível a clara falta de cifragem de informação sensível que passa por URLs e um atacante com visibilidade de rede (por exemplo, Wireshark) pode interceptar esta informação.

Correcção (app_sec):

Para corrigir foram aplicadas medidas de segurança como a biblioteca *"bleach"* para "limpar"inputs de passwords, verificação de passwords e a hash function *bcrypt* para proteger a password.

```python
@app.route('/change_password', methods=['POST'])
def change_password():
    if 'user_id' in session:
        user_id = session['user_id']
        if request.method == 'POST':
            current_password = bleach.clean(request.form['current_password'], strip=True)
            new_password = bleach.clean(request.form['new_password'], strip=True)

            user = db_session.query(Customer).filter_by(user_id=user_id).first()

            if not (user and check_password_hash(user.passwd, current_password)):
                flash("Current password is incorrect!", 'error')
                return redirect('/dashboard')

            hashed_password = bcrypt.hashpw(new_password.encode('utf-8'), bcrypt.gensalt())
            user.passwd = hashed_password
            db_session.commit()

            flash("Password changed successfully!", 'success')
            return redirect('/dashboard')
    else:
        flash("You need to login first! ", 'error')
        return redirect('/login')
```

Figura 34: CWE-319: Versão Segura: @app.route change_password() - Base de Dados(app_sec.py)

3.10 CWE-472: External Control of Assumed-Immutable Web Parameter

Esta vulnerabilidade ocorre quando se fazem suposições incorretas sobre a imutabilidade e confiabilidade de determinados parâmetros enviados para a aplicação, quando na realidade podem ser manipulados por atacantes.
Pode ocorrer de várias maneiras, como na falta de validação adequada de inputs ou a falta de controlo de acesso, resultando em consequências graves, como fuga de informações, elevação indevida de privilégios ou violação de integridade da aplicação em si.

Demonstração e Impacto (app)

Como se pode verificar pelas figuras 35 e 36, tanto o login como o registo de utilizadores não possui qualquer tipo de validação dos valores de entrada.

```
useEffect(() => {
    if(logentry.Email !== "" || logentry.Password !== ''){
        if(bResult === 1){
            cookies.set("userDetitalism", email, {expires: new Date(Date.now() + 86400000), sameSite: 'none', secure: true})
            getCartPerson(email).then(function(result){window.localStorage.setItem("detitalismoh", JSON.stringify(result))})
            navigatetoHome()
        } else if(bResult === 0){
            setInvLogin1(true)
            setInvLogin2(false)
        }else{
            setInvLogin2(true)
            setInvLogin1(false)
        }
    }
}, [bResult])
```

Figura 35: CWE-472: Versão Insegura: LoginPage (app/src/Components/Pages/LoginPage.js)

```
const handleSubmit = () => {
    regentry.FName = Fname
    regentry.LName = Lname
    regentry.Email = email
    if(firsterror === false){
        regPerson(regentry).then(function(result){setRes(result)})
    }
}
```

Figura 36: CWE-472: Versão Insegura: RegisterPage (app/src/Components/Pages/RegisterPage.js)

Correcção (app_sec):

Aplicando a função representada na figura 39 na LoginPage (figura37) e na RegisterPage(figura38) exerce-se a validação de valores de entrada.

```
useEffect(() => {
  if (logentry.Email !== "" || logentry.Password !== "") {
    if (!isValidInput(logentry.Email, logentry.Password)) {
      return;
    }
    if (bResult === 1) {
      cookies.set("userDetitalism", email, {
        expires: new Date(Date.now() + 86400000),
        sameSite: "none",
        secure: true,
      });
      getCartPerson(email).then(function (result) {
        window.localStorage.setItem("detitalismoh", JSON.stringify(result));
      });
      navigatetoHome();
    } else if (bResult === 0) {
      setInvLogin1(true);
      setInvLogin2(false);
    } else {
      setInvLogin2(true);
      setInvLogin1(false);
    }
  }
}, [logentry.Email, logentry.Password, bResult]);
```

Figura 37: CWE-472: Versão Segura: LoginPage (app_sec/src/Components/Pages/LoginPage.js)

```
const handleSubmit = () => {
  if (!passwordRequirements(password)) {
    setPassError(true);
    return;
  }
  if (!isValidInput(Fname, Lname, email, password)) {
    return;
  }
  regentry.FName = Fname;
  regentry.LName = Lname;
  regentry.Email = email;
  if (firsterror === false) {
    regPerson(regentry).then(function (result) {
      setRes(result);
    });
  }
};
```

Figura 38: CWE-472: Versão Segura: RegisterPage (app_sec/src/Components/Pages/RegisterPage.js)

```
const isValidInput = (Fname, Lname, email, password) => {
  return (
    Fname.length > 0 &&
    Lname.length > 0 &&
    email.includes("@") &&
    password.length >= 8
  );
};
```

Figura 39: CWE-472: Versão Segura: função presente na LoginPage e na RegisterPage

3.11 CWE-521: Weak Password Requirements

A vulnerabilidade CWE-521, refere-se a requisitos de password fracos. Ocorre quando não são impostos os requisitos de segurança adequados para as passwords dos utilizadores. Pode resultar em passwords facilmente descobertas, comprometendo a segurança geral da aplicação e podendo expor os dados dos utilizadores.

Para mitigar esta vulnerabilidade, é crucial implementar políticas para passwords robustas que exijam alguma complexidade das mesmas, incluindo uma combinação de caracteres alfanuméricos, símbolos especiais e letras maiúsculas e minúsculas. Além disso, é importante impor limites de comprimento e garantir que as senhas não sejam facilmente previsíveis ou baseadas em informações pessoais identificáveis.

Demonstração e Impacto (app)

Como se pode verificar a partir da figura, o registo de uma entidade não pressupõe qualquer tipo de requerimento de segurança sobre a password, deixando assim, a aplicação vulnerável a partir das entidades cujas passwords não sejam seguras.

```
const handleSubmit = () => {
    regentry.FName = Fname
    regentry.LName = Lname
    regentry.Email = email
    if(firsterror === false){
        regPerson(regentry).then(function(result){setRes(result)})
    }
}
```

Figura 40: CWE-521: Exemplo 1 - Versão Insegura: RegisterPage (app/src/Components/Pages/RegisterPage.js)

Correcção (app_sec):

A criação de uma função que garante que a password cumpre todos os requisitos no registo de uma conta assegurou a mitigação da vulnerabilidade. Essa função pode ser vista na figura 42.

```
const handleSubmit = () => {
  if (!passwordRequirements(password)) {
    setPassError(true);
    return;
  }
```

Figura 41: CWE-521: Versão Segura: RegisterPage (http://localhost:3000/Register)

```
const passwordRequirements = (password) => {
  const minLength = 8;
  const hasUppercase = /[A-Z]/.test(password);
  const hasLowercase = /[a-z]/.test(password);
  const hasNumber = /[0-9]/.test(password);
  return (
    password.length >= minLength && hasUppercase && hasLowercase && hasNumber
  );
};
```

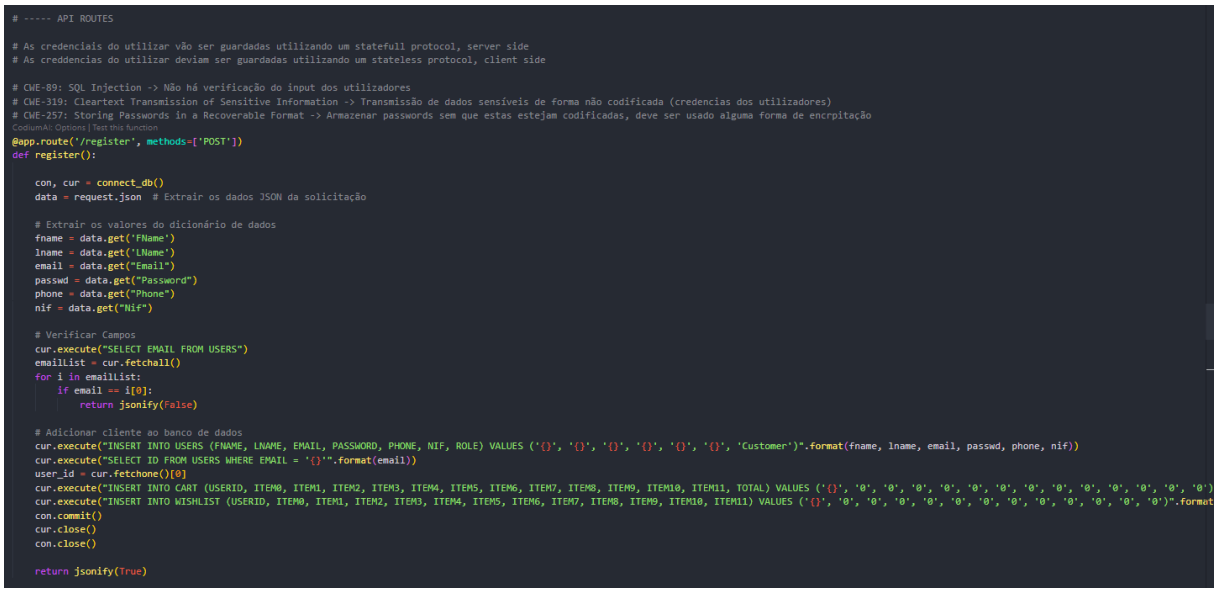
Figura 42: CWE-521: Versão Segura: LoginPage (http://localhost:3000/Login)

3.12 CWE-541: Inclusion of Sensitive Information in an Include File

Quando informações críticas (sensíveis) são inadvertidamente expostas devido à sua inclusão em arquivos (em forma de comentário, por exemplo) que não são adequadamente protegidos, pode criar riscos significativos, uma vez que estes arquivos podem ser acedidos por atacantes. Se as informações sensíveis não forem devidamente protegidas, um atacante com acesso a esses arquivos pode usá-las para fins maliciosos como exploração de vulnerabilidades reveladas ou até obter acesso não autorizado a dados confidenciais.

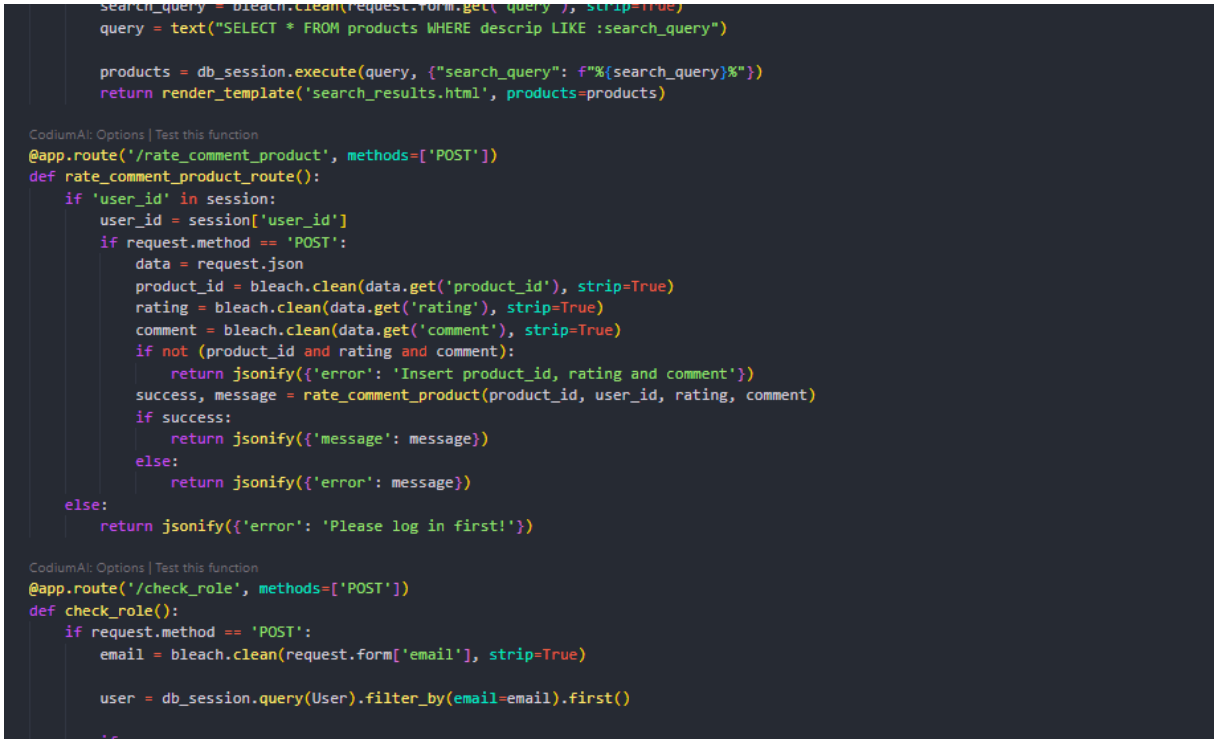
Demonstração e Impacto (app)

No decorrer do desenvolvimento da app.py (insegura), efectuámos diversos comentários sobre vulnerabilidades presentes no código. Este comentário podem ser usados por um atacante, facilitando a sua visão sobre a aplicação e a sua consequente exploração das vulnerabilidades incorrectamente indicadas. Seguidamente nas Figuras 43 e 45, apresentamos dois exemplos deste problema na nossa base de dados insegura (app.py):



```
# ----- API ROUTES

# As credenciais do utilizar vão ser guardadas utilizando um statefull protocol, server side
# As creddencias do utilizar deviam ser guardadas utilizando um stateless protocol, client side

# CWE-89: SQL Injection -> Não há verificação do input dos utilizadores
# CWE-319: Cleartext Transmission of Sensitive Information -> Transmissão de dados sensíveis de forma não codificada (credencias dos utilizadores)
# CWE-257: Storing Passwords in a Recoverable Format -> Armazenar passwords sem que estas estejam codificadas, deve ser usado alguma forma de encrpitação
CodiumAI: Options | Test this function
@app.route('/register', methods=['POST'])
def register():

    con, cur = connect_db()
    data = request.json  # Extrair os dados JSON da solicitação

    # Extrair os valores do dicionário de dados
    fname = data.get('FName')
    lname = data.get('LName')
    email = data.get("Email")
    passwd = data.get("Password")
    phone = data.get("Phone")
    nif = data.get("Nif")

    # Verificar Campos
    cur.execute("SELECT EMAIL FROM USERS")
    emailList = cur.fetchall()
    for i in emailList:
        if email == i[0]:
            return jsonify(False)

    # Adicionar cliente ao banco de dados
    cur.execute("INSERT INTO USERS (FNAME, LNAME, EMAIL, PASSWORD, PHONE, NIF, ROLE) VALUES ('{}', '{}', '{}', '{}', '{}', '{}', 'Customer')".format(fname, lname, email, passwd, phone, nif))
    cur.execute("SELECT ID FROM USERS WHERE EMAIL = '{}'".format(email))
    user_id = cur.fetchone()[0]
    cur.execute("INSERT INTO CART (USERID, ITEM0, ITEM1, ITEM2, ITEM3, ITEM4, ITEM5, ITEM6, ITEM7, ITEM8, ITEM9, ITEM10, ITEM11, TOTAL) VALUES ('{}', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0')
    cur.execute("INSERT INTO WISHLIST (USERID, ITEM0, ITEM1, ITEM2, ITEM3, ITEM4, ITEM5, ITEM6, ITEM7, ITEM8, ITEM9, ITEM10, ITEM11) VALUES ('{}', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0')".format
    con.commit()
    cur.close()
    con.close()

    return jsonify(True)
```

Figura 43: CWE-541: Exemplo 1 - Versão Insegura: Base de Dados (app.py)



```
        search_query = bleach.clean(request.form.get('query'), strip=True)
        query = text("SELECT * FROM products WHERE descrip LIKE :search_query")

        products = db_session.execute(query, {"search_query": f"%{search_query}%"})
        return render_template('search_results.html', products=products)

CodiumAI: Options | Test this function
@app.route('/rate_comment_product', methods=['POST'])
def rate_comment_product_route():
    if 'user_id' in session:
        user_id = session['user_id']
        if request.method == 'POST':
            data = request.json
            product_id = bleach.clean(data.get('product_id'), strip=True)
            rating = bleach.clean(data.get('rating'), strip=True)
            comment = bleach.clean(data.get('comment'), strip=True)
            if not (product_id and rating and comment):
                return jsonify({'error': 'Insert product_id, rating and comment'})
            success, message = rate_comment_product(product_id, user_id, rating, comment)
            if success:
                return jsonify({'message': message})
            else:
                return jsonify({'error': message})
    else:
        return jsonify({'error': 'Please log in first!'})

CodiumAI: Options | Test this function
@app.route('/check_role', methods=['POST'])
def check_role():
    if request.method == 'POST':
        email = bleach.clean(request.form['email'], strip=True)

        user = db_session.query(User).filter_by(email=email).first()
```

Figura 44: CWE-541: Exemplo 2 - Versão Insegura: Base de Dados (app.py)

Correcção (app_sec):

A remoção de comentários sensíveis (neste caso em específico sobre vulnerabilidades) resolve este problema. Devem ser apenas usados comentários de natureza relevante para os programadores sendo que a informação relativa a potenciais vulnerabilidades deve ser documentada em locais protegidos e acessíveis apenas por programadores da aplicação.

```
        search_query = bleach.clean(request.form.get('query'), strip=True)
        query = text("SELECT * FROM products WHERE descrip LIKE :search_query")

        products = db_session.execute(query, {"search_query": f"%{search_query}%"})
        return render_template('search_results.html', products=products)

CodiumAI: Options | Test this function
@app.route('/rate_comment_product', methods=['POST'])
def rate_comment_product_route():
    if 'user_id' in session:
        user_id = session['user_id']
        if request.method == 'POST':
            data = request.json
            product_id = bleach.clean(data.get('product_id'), strip=True)
            rating = bleach.clean(data.get('rating'), strip=True)
            comment = bleach.clean(data.get('comment'), strip=True)
            if not (product_id and rating and comment):
                return jsonify({'error': 'Insert product_id, rating and comment'})
            success, message = rate_comment_product(product_id, user_id, rating, comment)
            if success:
                return jsonify({'message': message})
            else:
                return jsonify({'error': message})
    else:
        return jsonify({'error': 'Please log in first!'})

CodiumAI: Options | Test this function
@app.route('/check_role', methods=['POST'])
def check_role():
    if request.method == 'POST':
        email = bleach.clean(request.form['email'], strip=True)

        user = db_session.query(User).filter_by(email=email).first()
```

Figura 45: CWE-541: Exemplo 2 - Versão Segura: Base de Dados (app_sec.py)

3.13 CWE-549: Missing Password Field Masking

Campos de dados sensíveis, como passwords, em interfaces de utilizador devem ser "mascarados"durante a entrada de dados, para ocultar os caracteres enquanto o campo é preenchido pelo utilizador.É uma prática comum que envolve a substituição dos caracteres do campo por símbolos, como asteriscos ou pontos. A ausência desta medida de segurança pode resultar em riscos de segurança, não protegendo visualmente os caracteres secretos, ficando expostos a qualquer pessoa que esteja próximo ou que esteja a monotorizar a interface gráfica (GUI) de forma ilegal, facilitando assim o acesso indevido a contas de utilizador e/ou roubo de informações sensíveis.

Demonstração e Impacto (app)

Na app insegura, nenhum dos dados sensíveis está mascarado de modo a garantir a anonimidade desses dados. Isto é evidente nas figuras 46 e 47 abaixo.

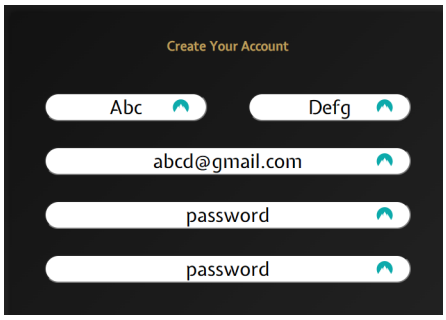


Figura 46: CWE-549: Versão Insegura: RegisterPage (http://localhost:3000/Register)

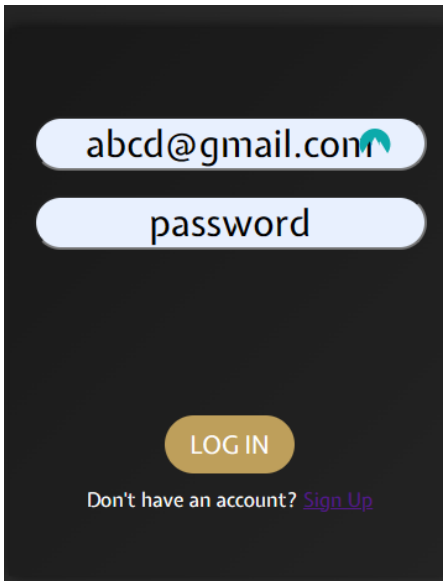


Figura 47: CWE-549: Versão Insegura: LoginPage (http://localhost:3000/Login)

```
<fieldset className="form-groupR">
    <input className="form-inputR" value={password} onChange={(e) => setPassword(e.target.value)} type="text" name="password" id="password" placeholder='Password'/>
</fieldset>
<fieldset className="form-groupR">
    <input className="form-inputR" value={confpassword} onChange={(e) => setConfPassword(e.target.value)} type="text" name="password" id="password" placeholder='Confirm Password'/>
</fieldset>
```

Figura 48: CWE-549: Versão Insegura: RegisterPage (app/src/Components/Pages/RegisterPage.js)

```
<fieldset className="form-group">
    <input className="form-input1" value={password} onChange={(e) => setPassword(e.target.value)} type="text" name="password" id="password" placeholder='Password'/>
</fieldset>
```

Figura 49: CWE-549: Versão Insegura: LoginPage (app/src/Components/Pages/LoginPage.js)

Correcção (app_sec):

Ao alterar o atributo type da password para "password", como demonstra nas figuras 52 e 53, os caracteres da password passam a ser representados por pontos, mascarando-a. Figuras 50 e 51

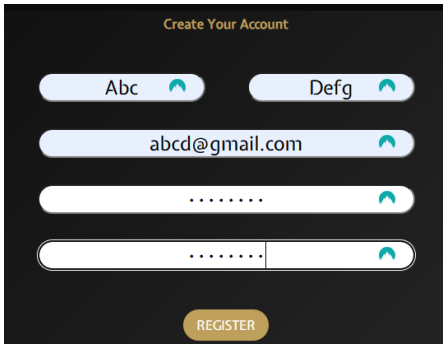


Figura 50: CWE-549: Versão Segura: RegisterPage (http://localhost:3000/Register)

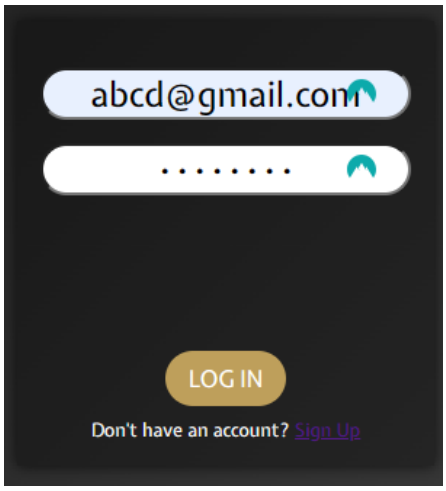


Figura 51: CWE-549: Versão Segura: LoginPage (http://localhost:3000/Login)

```
<fieldset className="form-groupR">
    <input className="form-inputR" value={password} onChange={(e) => setPassword(e.target.value)} type="password" name="password" id="password" placeholder='Password'/>
</fieldset>
<fieldset className="form-groupR">
    <input className="form-inputR" value={confpassword} onChange={(e) => setConfPassword(e.target.value)} type="password" name="password" id="password" placeholder='Confirm Password'/>
</fieldset>
```

Figura 52: CWE-549: Versão Segura: RegisterPage (app_sec/src/Components/Pages/RegisterPage.js)

```
<fieldset className="form-group">
    <input className="form-input1" value={password} onChange={(e) => setPassword(e.target.value)} type="password" name="password" id="password" placeholder='Password'/>
</fieldset>
```

Figura 53: CWE-549: Versão Segura: LoginPage (app_sec/src/Components/Pages/LoginPage.js)

3.14 CWE-620: Unverified Password Change

A vulnerabilidade CWE-620 refere-se à capacidade de alterar a password do utilizador sem ser necessário fornecer a password antiga. Como consequência, na eventualidade de um atacante conseguir acessar a conta, este poderá alterar a password sem dificuldade e negar o acesso à conta ao utilizador.

Demonstração e Impacto (app)

Como é demonstrado nas figuras 54 e 55, para a mudança de password apenas é necessário introduzir a nova password pretendida, tornando-se assim uma ação de fácil execução

```
function handlepass(){
    let name = ""
    const aCookie = document.cookie.split(';')
    for(let i = 0; i < aCookie.length; i++){
        let cookie = aCookie[i].split('=')[0].trim()
        if(cookie === 'userDetitalism'){
            name = aCookie[i].split('=')[1]
        }
    }
    let data = []
    data[0] = name
    data[1] = newpass
    updatePass(data)
}
```

Figura 54: CWE620: Versão Insegura: ChangePassPage
(app/src/Components/Pages/ChangePassPage.js)

```
@app.route('/change_password', methods=['POST'])
def change_password():
    con, cur = connect_db()
    data = request.json

    email = data[0]
    new_password = data[1]

    if '%40' in email:
        email = email.replace('%40', '@')


    cur.execute("SELECT EMAIL FROM USERS")
    emailList = cur.fetchall()
    for i in emailList:
        if email == i[0]:
            cur.execute(f"SELECT PASSWORD FROM USERS WHERE EMAIL = '{email}'")
            cur.execute(f"UPDATE USERS SET PASSWORD = '{new_password}' WHERE EMAIL = '{email}'")
            con.commit()
            cur.close()
            con.close()
            return jsonify(1)
        else:
            return jsonify(2)
    cur.close()
    con.close()
    return jsonify(0)
```

Figura 55: CWE620: Versão Insegura: app
(app/src/backend/app.py)

Correcção (app_sec):

Para garantir que a password não é alterada por algum atacante, a alteração da password requer a introdução da password antiga. Para tal, adicionámos a variável *oldpass*, figura 56 que será necessária para a operação em causa, figura 57 .

```
CodiumAI: Options | Test this function
function PassPage() {
/************ Const Declaration ************/
    const [oldpass, setOldPass] = useState('');
    const [newpass, setNewPass] = useState('');


    function handlepass(){
        let name = ""
        const aCookie = document.cookie.split(';')
        for(let i = 0; i < aCookie.length; i++){
            let cookie = aCookie[i].split('=')[0].trim()
            if(cookie === 'userDetitalism'){
                name = aCookie[i].split('=')[1]
            }
        }
        let data = []
        data[0] = name
        data[1] = oldpass
        data[2] = newpass
        updatePass(data)
    }
```

Figura 56: CWE620: Versão Insegura: ChangePassPage (app_sec/src/Components/Pages/ChangePassPage.js)

```
@app.route('/change_password', methods=['POST'])
def change_password():
    if 'user_id' in session:
        user_id = session['user_id']
        if request.method == 'POST':
            current_password = bleach.clean(request.form['current_password'], strip=True)
            new_password = bleach.clean(request.form['new_password'], strip=True)

            user = db_session.query(User).filter_by(user_id=user_id).first()

            if not (user and check_password_hash(user.passwd, current_password)):
                flash("Current password is incorrect!", 'error')
                return redirect('/dashboard')

            hashed_password = bcrypt.hashpw(new_password.encode('utf-8'), bcrypt.gensalt())
            user.passwd = hashed_password
            db_session.commit()

            flash("Password changed successfully!", 'success')
            return redirect('/dashboard')
    else:
        flash("You need to login first! ", 'error')
        return redirect('/login')
```

Figura 57: CWE620: Versão Insegura: app (app_sec/src/backend/app_sec.py)

3.15 CWE-710: Improper Adherence to Coding Standards e CWE-1006: Bad Coding Practices

Uma solução de software deve aderir rigorosamente às diretrizes e práticas do paradigma de programação do seu domínio, a fim de prevenir potenciais vulnerabilidades.

Demonstração e Impacto (app)

A base de dados insegura, foi criada num único ficheiro *app.py* (sendo não modelar) e verifica-se um grande agrupamento de código que pode tornar as suas leitura, compreensão e/ou correcção complexas, tanto para o programador como para o analista de segurança responsável por auditorias subsequentes, tal como demonstrado nas Figuras seguintes:

```
def change_product(product_id, new_title, new_description, new_category, new_price, new_stock):
    try:
        con, cur = connect_db()

        cur.execute(
            """ UPDATE products
            SET title = ?, descrip = ?, category = ?, price = ?, stock = ?
            WHERE product_id = ?
            """,
            (new_title, new_description, new_category, new_price, new_stock, product_id))

        con.commit()
        con.close()

        return "Product updated!"
    except Exception as e:
        return f"Error: {str(e)}"


#
##############################################################################################
##############################################################################################
# ----------------- CREATE APP, ROUTES ------------------------------------------------ #
# -----                                                                          ----- #

app = Flask(__name__)
__init_db__()

@app.route('/register', methods=['POST'])
def register():

    con, cur = connect_db()
    data = request.json  # Extract the JSON data from the request

    # Extract the values from the data dictionary
    fname = data.get('FName')
    lname = data.get('LName')
    email = data.get("Email")
    passwd = data.get("Password")
    phone = data.get("Phone")
    nif = data.get("Nif")

    # Verify Fields
    cur.execute("SELECT EMAIL FROM USERS")
    emailList = cur.fetchall()
    for(i) in emailList:
        if email == i[0]:
            return jsonify(False)
```

Figura 58: CWE: 710 - Exemplo 1 - Versão insegura (app)

```
# ------------------------------------------------------------------------------ #
# ----------------- TABLE: PAYMENTS ------------------------------------------- #
# -----                                                                          ----- #
    cur.execute("""DROP TABLE IF EXISTS payments""")
    cur.execute(
        """CREATE TABLE payments (
        cart_id     INTEGER                         REFERENCES  cart(cart_id)   ,
        mbway       VARCHAR(256)    NOT NULL                                    ,
        ref_mb      VARCHAR(256)    NOT NULL                                    ,
        paypal      VARCHAR(256)    NOT NULL                                    );
        """
    )
#
#
# ------------------ END: TABLES & INSERTS OF DATABASE ------------------------------ #
###############################################################################
###############################################################################
#
    con.commit()
    con.close()
#
###############################################################################
###############################################################################
# ----------------- BEGIN: AUX. FUNCITONS FOR DATABASE ----------------------------- #
# -----                                                                          ----- #
#
# ----- FUNCTION connect_db():
#
def connect_db():
        directory = os.path.dirname(os.path.abspath(__file__))
        db_path = os.path.join(directory, 'DOSG26.db')
        con = sqlite3.connect(db_path, check_same_thread=False)
        cur = con.cursor()
        return con, cur

# VULNERABILIDADE SUBTIL: Permite que um atacante se registe multiplas veses usando Upper e Lower cases (letras maiusculas e minusculas)
def new_customer(email, password):
    con, cur = connect_db()
    cur.execute("SELECT email FROM users WHERE email = ?", (email,))
    existing_email = cur.fetchone()

    # Verificação não é case-sensitive:
    if existing_email:
        con.close()
        return "Email address already registered."
```

Figura 59: CWE: 710 - Exemplo 2 - Versão insegura (app)

Correcção (app_sec):

Para corrigir o problema de boas práticas e modelaridade presentes na aplicação insegura, esta foi restruturada em diversos modulos, dividindo o seu código em "blocos"o que, com uso de boas práticas, torna toda a organização, leitura, compreensão e correcção mais simples e eficiente do ponto de vista do programador ou analista.

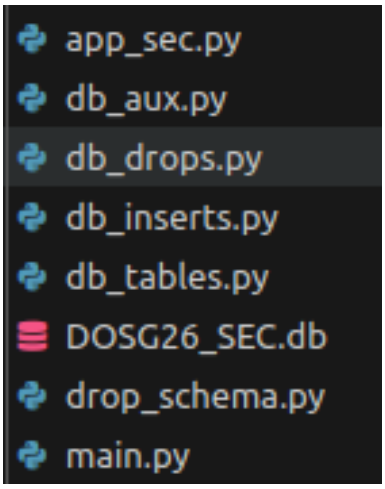


Figura 60: CWE: 710 - Versão Segura (backend modular)

3.16 Vulnerabilidade subtil - Light Analysis Failure

Existe uma vulnerabilidade subtil que pode não ser detectada por *"light analysis"*. Esta encontra-se implementada na função ***new_ customer***.

Demonstração e Impacto (app)

Esta função permite a um atacante efectuar um registo múltiplas vezes usando as mesmas credenciais, alterando letras maiúsculas e minúsculas, tal como demonstrado na Figura 61

```
def new_customer(email, password):
    con, cur = connect_db()
    cur.execute("SELECT email FROM users WHERE email = ?", (email,))
    existing_email = cur.fetchone()

    if existing_email:
        con.close()
        return "Email address already registered."

    cur.execute("INSERT INTO users (email, password) VALUES (?, ?)", (email, password))
    con.commit()
    con.close()
    return "Registration successful."
```

Figura 61: Vulnerabilidade subtil - Base de Dados(app.py)

Correcção (app_sec):

Para corrigir, tal como demonstrado na Figura 62, a função foi rescrita em SQLAlchemy fazendo uso de confirmações de *"password matching"* e do método *"filter_ by"* para efectuar a *query* de forma *case-sensitive*, corrigindo efectivamente o problema descrito.

```python
#
# ----- FUNCTION new_customer:
def new_customer(session, fname, lname, email, nick, passwd, confirm):
    if passwd != confirm:
        error = "Passwords do not match."
        return False, error

    existing_user = session.query(Customer).filter_by(email=email, nick=nick).first()
    if existing_user:
        error = "Email or nickname is already in use."
        return False, error

    customer = Customer(fname=fname, lname=lname, email=email, nick=nick, passwd=passwd)
    session.add(customer)
    session.commit()
    return True, "Registration complete!"
```

Figura 62: Vulnerabilidade subtil corrigida: Base de Dados(app_sec.py)

Conclusão:

No decorrer deste projecto, foi efectuada uma análise abrangente do ciclo de vida de desenvolvimento da aplicação, partindo de uma implementação insegura até alcançar uma versão segura e corrigida. Este processo destacou a importância crítica da cibersegurança e das práticas adequadas de programação no desenvolvimento de soluções adequadas para *deployment* na Web.

Foi possível efectuar com sucesso todos os objectivos espectados:

- Na versão inicial da solução aplicacional foram implementadas, identificadas, testadas e analisadas diversas Common Weakness Enumeration (CWE) que simulam uma solução com riscos de segurança significativos para o sistema e futuros utilizadores.
- Posteriormente, dada a análise (auditoria) anteriormente efectuada, a solução insegura foi corrigida adequadamente tornando-a assim, mais segura.

Este projecto ilustra claramente as consequências significativas no mundo real das práticas seguras de programação para aplicações e/ou soluções Web. A negligência em relação às boas práticas e segurança pode resultar em violações de dados e privacidade, perdas financeiras, furto de identidade e danos à reputação de uma organização ou utilizador comum.

Contribuições dos Autores

Os autores responsáveis pela pesquisa e desenvolvimento do presente projecto, contribuiram de forma coletiva, refletindo-se diversidade de experiências, opiniões e competências que enriqueceram a abordagem ao mesmo. Este projecto pode ser visualizado na sua totalidade na página: `https://github.com/detiuaveiro/1st-project-group_26`.

Autores:

- João Pedro Nunes Vieira, Nºmec.: 50458
- José Miguel Guardado Silva, Nº Mec.: 103248
- Henrique Miguel Escudeiro Cruz, Nº Mec.: 103442
- Luís Manuel Trindade Diogo, Nº Mec.: 108668

Tabela 1: Contribuições dos Autores.

Contribuição	João Vieira	José Silva	Henrique Cruz	Luís Diogo
Produção de Relatório (LaTex)	25 %	25 %	25 %	25 %
Identificação e descrição de CWEs	25 %	25 %	25 %	25 %
Desenvolvimento de Front-End inseguro (REACT)	0 %	50 %	0 %	50 %
Desenvolvimento de Front-End seguro (REACT)	0 %	50 %	0 %	50 %
Desenvolvimento de Back-End inseguro (SQLite e Flask)	50 %	0 %	50 %	0 %
Desenvolvimento de Back-End seguro (SQLAlchemy e Flask)	50 %	0 %	50 %	0 %
Testes e Depuração	25 %	25 %	25 %	25 %

Bibliografia

[1] Allen Downey *Think Python - How to Think Like a Computer Scientist.* | Green Tea Press, 2nd Edition, Version 2.4.0, 2015

[2] André Zúquete *Segurança em Redes Informáticas.* | FCA - Editora de Informática LDA, 5th Edition, 2018

[3] The MITRE Corporation https://cwe.mitre.org/data/published/cwe_latest.pdf . | CWE - Version 4.13 - (2023-10-26), Copyright 2023, The MITRE Corporation

[4] Wikipédia: Enciclopédia livre. | pt.wikipedia.org, acedido 10/10/2023.

[5] SQLite Website. | https://www.sqlite.org/docs.html, acedido 11/10/2021.

[6] SQLAlchemy Website. | https://docs.sqlalchemy.org/en/20/, acedido 11/10/2021.

[7] MITRE: A Community-Developed List of Software and Hardware Weakness Types. | https://cwe.mitre.org/index.html, acedido 16/10/2021.