



Vulnerabilities in software products

Relatório

Gonçalo Lima - 108254
Tiago Fonseca - 107266
Beatriz Ferreira - 107214
Tomás Fonseca - 107245
Francisco Murcela - 108815
Grupo 10

Universidade de Aveiro
Segurança Informática e
Nas Organizações

Índice

Introdução	3
Vulnerabilidades	4
<i>CWE-79 - Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')..</i>	<i>4</i>
<i>CWE-89 - Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').....</i>	<i>8</i>
<i>CWE-256 - Plaintext Storage of a Password.....</i>	<i>12</i>
<i>CWE-285 - Improper Authorization.....</i>	<i>16</i>
<i>CWE-756 - Missing Custom Error Page.....</i>	<i>19</i>
<i>CWE-798 - Use of Hard-coded Credentials.....</i>	<i>20</i>
<i>CWE-620 - Unverified Password Change.....</i>	<i>29</i>
<i>CWE-640 - Weak Password Recovery Mechanism for Forgotten Password.....</i>	<i>34</i>
Bibliografia	38



Introdução

Este relatório apresenta os resultados de um projeto abrangente que se centra no desenvolvimento de uma loja virtual especializada em produtos de memorabilia do DETI (Departamento de Eletrónica, Telecomunicações e Informática) da Universidade de Aveiro. Além de satisfazer os requisitos convencionais de uma plataforma de comércio eletrónico, este projeto implica um desafio adicional: identificar e mitigar vulnerabilidades não imediatamente visíveis, que poderiam comprometer a integridade e a segurança do sistema.

Ao longo deste relatório, iremos analisar minuciosamente diversas vulnerabilidades, com um enfoque específico num conjunto designado de *Common Weakness Enumeration (CWE)*, que inclui as categorias *CWE-79*, *CWE-89*, *CWE-256*, *CWE-285*, *CWE-756*, *CWE-798*, *CWE-620* e *CWE-640*. Cada secção dedicada a estas *CWEs* fornecerá uma análise detalhada das suas características, exemplos práticos e o potencial impacto que podem acarretar.

O nosso principal objetivo com este projeto é ampliar o nosso conhecimento sobre vulnerabilidades e aprimorar a nossa capacidade de identificar soluções para as mesmas. Almejamos uma compreensão mais profunda das fragilidades nos sistemas e a habilidade para implementar medidas eficazes visando a sua proteção. Acreditamos que este projeto contribuirá para o aperfeiçoamento das nossas competências e para a promoção de ambientes online mais seguros.

A aplicação Web (versão segura) foi completamente implementada e pode ser encontrada [aqui](#).



Vulnerabilidades

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

A *CWE-79* é uma vulnerabilidade de segurança relacionada com a inadequada neutralização de entradas durante a construção de páginas *Web*, sendo conhecida como *Cross-Site Scripting (XSS)*. Esta vulnerabilidade ocorre quando uma aplicação cria parcial ou totalmente uma página web usando entradas que podem ser influenciadas por fontes externas, sem efetuar a neutralização adequada ou ao realizar uma neutralização incorreta de elementos especiais. Estes elementos podem, por sua vez, alterar o comando SQL pretendido quando são enviados para um componente subsequente.

Ao analisar a funcionalidade de adição de análises de produtos, detetámos uma vulnerabilidade de *Cross-Site Scripting (XSS)* na área de entrada de texto. Na visualização da página de adição de análises de produtos, ao incluir uma análise, a entrada de texto permitia a obtenção de imagens de perfil de outros utilizadores através do seguinte código:

```

```

Isso foi possível devido ao facto de que, durante a adição de uma análise, a entrada no campo de texto não estava a ser sujeita a verificações quanto a *Cross-Site Scripting (XSS)* e não estava a ser devidamente depurada antes de ser incorporada na análise. Isso permitia a inserção de código malicioso no campo de texto da análise, que poderia ser executado no navegador do utilizador de forma não intencional.



Aqui encontra-se o excerto de código que ilustra a vulnerabilidade:

```
def add_review(product_id):  
  
    # Get the user's id and username from the session  
    user_id = session.get("id")  
    username = session.get("username")  
  
    # Get the review and rating from the request  
    review = request.form.get("userReview")  
    rating = request.form.get("rating")  
  
    # Create the review  
    create_review(product_id, user_id, review, rating)  
  
    # Return a JSON response with the correct content type  
    response_data = {'message': 'Review added successfully', "username":  
username}  
    return jsonify(response_data), 200, {'Content-Type': 'application/json'}
```

Neste código, a variável `review` armazena a entrada do utilizador, a qual não é alvo de verificação para possíveis elementos maliciosos de XSS, tais como *scripts* ou *tags HTML*. Consequentemente, um atacante poderia inserir código malicioso no campo de texto da análise e, quando essa análise é apresentada a outros utilizadores, o código malicioso é executado nos respetivos navegadores.

Neste [link](#), é viável observar uma demonstração que exemplifica a vulnerabilidade em funcionamento.



Para resolver esta vulnerabilidade, implementámos a função `is_valid_input(review_text)` para verificar se a entrada (ou seja, a crítica) não inclui padrões, expressões ou caracteres associados a *Cross-Site Scripting* (XSS).

```
def add_review(product_id):

    # Get the user's id and username from the session
    user_id = session.get("id")
    username = session.get("username")

    if user_id == None:
        return redirect(url_for("views.login"))

    # Get the review and rating from the request
    review = request.form.get("userReview")
    rating = request.form.get("rating")

    if not is_valid_input(review) or verify_id_exists(product_id, "products") == False or rating == None:
        return jsonify({'error': 'Invalid review.'}), 500

    # Create the review
    create_review(product_id, user_id, review, rating)

    # Return a JSON response with the correct content type
    response_data = {'message': 'Review added successfully', "username": username}
    return jsonify(response_data), 200, {'Content-Type': 'application/json'}
```



```
def is_valid_input(review_text):  
    # Define a regular expression pattern to match valid characters  
    valid_characters_pattern = re.compile(r'^[a-zA-Z0-9,.!?\()\'" @]+$')  
  
    # Check if the review contains only characters not in the valid pattern  
    if not valid_characters_pattern.match(review_text):  
        return False  
  
    if re.search(r"<script>", review_text) or re.search(r"onload=", review_text)  
or re.search(r"<img>", review_text):  
        return False  
  
    # Check if the review contains a single quote  
    if "'" in review_text:  
        return False  
  
    # If none of the checks above returned False, the review is valid  
    return True
```

A função `is_valid_input(review_text)` foi desenvolvida com o propósito de verificar a ausência de elementos potencialmente perigosos nas análises, tais como `<script>`, `onload=`, ``, e aspas simples, que poderiam ser explorados em ataques de *Cross-Site Scripting* (XSS). Caso tais elementos sejam detetados, a função retornará `False`, sinalizando uma análise insegura; em contrapartida, retornará `True`, indicando que a análise está protegida contra XSS.

Deste modo, ao recorrer a esta função, qualquer entrada XSS no campo de texto das análises não será incluída, contribuindo para a prevenção desta vulnerabilidade de segurança.



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

A CWE-89 refere-se a uma vulnerabilidade de segurança associada à incorreta neutralização de elementos especiais utilizados num comando SQL, conhecida como Injeção SQL. Esta vulnerabilidade surge quando um produto constrói parte ou a totalidade de um comando SQL utilizando entrada externamente influenciada a partir de um componente upstream, porém, não procede à neutralização adequada ou realiza uma neutralização incorreta dos elementos especiais que poderiam alterar o comando SQL original quando transmitidos a um componente downstream.

No contexto de um processo de login, a aplicação *Web* pode enfrentar falhas se o nome de utilizador for:

```
SELECT password FROM users WHERE username = 'malicious' OR '1'='1' ;  
SELECT password FROM users WHERE username = ' ' ' ;
```

Na página de *Login*, recorria-se à função `validate_login(username, password)` no ficheiro *UserManagement.py* para efetuar a validação do *Login*.

```
def validate_login(username, password):  
  
    # If username is None, return False (user not found)  
    if username is None:  
        return False  
    else:  
        # Fetch the user's password  
        # Secure Query  
        # query = "SELECT password FROM users WHERE username = %s"  
        # result = db_query(query, (username,))  
  
        query = "SELECT password FROM users WHERE username = '"+username + "';"  
        result = db_query(query)  
  
        # Check if there is a password  
        if not result:  
            return None  
        # Check if the provided password matches the user's password  
        if result[0][0] == password:  
            # Return True to indicate the login has been validated  
            return True  
  
        else:  
            # Return False to indicate the login credentials aren't valid  
            return False
```




Como é evidente, esta função constrói uma consulta *SQL* ao juntar uma variável de nome de utilizador à string da consulta, tornando-a vulnerável a Injeção *SQL*.

Pode verificar uma demonstração que exemplifica o problema original seguindo este [link](#).

Para sanar esta vulnerabilidade, adotamos o uso de uma consulta parametrizada na função `search_user_by_username(username)` e adaptamos a nossa consulta personalizada `db_query(query, params=None)` para funcionar com parâmetros.



Aqui encontram-se as descrições das duas funções:

```
def search_user_by_username(username):  
  
    # Secure Query  
    query = "SELECT * FROM users WHERE username = %s"  
    result = db_query(query, (username,))  
  
    # If no user is found, return None  
    if not result:  
        return None  
  
    # Return the user data  
    return result[0]
```

```
def db_query(query, params=None):  
  
    # Get the credentials for accessing the database  
    credentials = read_json("/credentials/DataBaseCredentials.json")  
  
    # Connect to the database  
    conn = psycopg2.connect(  
        host=credentials["host"],  
        dbname=credentials["dbname"],  
        user=credentials["user"],  
        password=credentials["password"],  
        port=credentials["port"]  
    )  
  
    # Initiate the cursor  
    cur = conn.cursor()  
  
    # Check if there is any parameters  
    if params:  
        # Execute query with parameters  
        cur.execute(query, params)  
  
    else:  
        # Execute query without parameters  
        cur.execute(query)  
  
    # Define select_in_query as False by default  
    select_in_query = False
```



```
# Check if the query has SELECT
if "SELECT" in query:

    # Fetch all the data
    data = cur.fetchall()
    select_in_query = True

# Commit the connection
conn.commit()

# Close the cursor
cur.close()

# Close the connection
conn.close()

# Check if the query has SELECT
if select_in_query:

    # Return the requested data
    return data
```

Deste modo, ao recorrermos a um cursor para a execução das consultas, conseguimos eliminar a vulnerabilidade de Injeção SQL.

Esta abordagem contribui para assegurar que as consultas SQL não possam ser alvo de manipulações maliciosas por parte de potenciais atacantes, reforçando a segurança dos dados e da aplicação.



CWE-256 - Plaintext Storage of a Password

Problemas de gestão de palavras-passe ocorrem quando a palavra-passe é armazenada em texto simples, não cifrado, nas configurações de uma aplicação, num ficheiro de configuração ou na memória.

Quando uma palavra-passe é guardada desprotegida num ficheiro de configuração, isso permite que qualquer pessoa que tenha acesso a esse ficheiro possa aceder aos recursos protegidos pela palavra-passe. Em alguns contextos, a prática de armazenar uma palavra-passe não cifrada na memória é considerada um risco de segurança, a menos que a palavra-passe seja imediatamente eliminada após o seu uso.

Para mitigar estas vulnerabilidades, é aconselhável evitar o armazenamento de palavras-passe em locais de fácil acesso e, em alternativa, armazenar *hashes* criptográficos das palavras-passe em vez de as guardar em texto simples.

Caso ocorra uma falha de segurança na base de dados, um atacante poderá visualizar todas as palavras-passe, uma vez que não se encontram cifradas.

```
# This view is used to enroll new users into the platform
@views.route("/signup", methods=["GET", "POST"])
def signup():
    if request.method == "POST":
        # Get the username, password and email from the request
        username = request.form.get("username")
        password = request.form.get("password")
        email = request.form.get("email")

        # Check if there is no user in the database using the same email or the
        same username
        if search_user_by_email(email) != None or
        search_user_by_username(username) != None:

            # Return signup page if there is
            return render_template("signup.html", message="User already
exists.")
        else:

            # Create the user in the database
            create_user(username, password, email)

            # Return the 2FA signup page, in order to validate the email
            return redirect(url_for("views.login"))
    else:
        return render_template("signup.html")
```



```
# This view is used to check if the email exists
@views.route('/update_account/<id>', methods=['POST'])
def update_account(id):

    # Set the user's account image file path
    file_path = os.getcwd()+f"/database/accounts/{id}.png"

    # Get the new uploaded user's account image
    profile_photo = request.files.get("profile_photo")

    # If there is a image
    if profile_photo:

        # Save the image to the user's account
        profile_photo.save(file_path)

    # Get the username, email and password, from the user's session
    username = request.form.get("username")
    email = request.form.get("email")
    password = request.form.get("psw")

    # Check if the username field wasn't empty and occupied by another user
    if username != "" and not check_username_exists(username):

        # Update the username
        update_username(id, username)

        # Set the sessions username
        session["username"] = username

    else:

        # If there is a problem with the username, get the username based on the
ID        username = search_user_by_id(id)[1]

    # Check if the email field wasn't empty and occupied by another user
    if email != "" and not check_email_exists(email):

        # Update the email
        update_email(id, email)

    else:

        # If there is a problem with the username, get the username based on the
ID        email = search_user_by_id(id)[3]
```



```
# Check if the password wasn't empty
if password != "":

    # Update the password
    update_password(id, password)

# Return the profile page
return redirect(url_for("views.catalog", id=id))
```

Para resolver esta vulnerabilidade, optamos por utilizar o **bcrypt** para gerar o *hash* da palavra-passe e armazená-lo. Quando necessário, comparamos simplesmente os dois hashes das palavras-passe para verificar se são idênticos.

```
@views.route("/signup", methods=["GET", "POST"])
def signup():
    if request.method == "POST":
        username = request.form.get("username").lower()
        password = request.form.get("password")
        email = request.form.get("email")

        if is_valid_input(username) == False or is_valid_input(email) == False:
            return render_template("signup.html", message="Invalid username.")

        # Check if there is no user in the database using the same email or the
        same username
        if search_user_by_email(email) != None or
search_user_by_username(username) != None:
            return render_template("signup.html", message="User already
exists.")
        else:
            # Hash the password before storing it in the database
            hashed_password =
bcrypt.generate_password_hash(password).decode("utf-8")

            # Create the user in the database with the hashed password
            create_user(username, hashed_password, email)

            return redirect(url_for("views.login"))
        else:
            return render_template("signup.html")
```



```
@views.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == "POST":
        username = request.form.get("username").lower()
        password = request.form.get("password")

        if is_valid_input(username) == False:
            return render_template("login.html", message="Invalid
username.")

        user = search_user_by_username(username)

        if user and bcrypt.check_password_hash(user[2], password):
            # Password is correct
            session["username"] = username
            session["id"] = get_id_by_username(username)
            session["admin"] = get_user_role(session["id"])
            check_database_table_exists(username.lower() + "_cart")
            check_database_table_exists(f"{username.lower()}_orders")
            return redirect(url_for("views.catalog", id=session["id"]))

            # Password is incorrect
            return render_template("login.html", message="Invalid login
credentials.")

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

Antes

	id [PK] integer	username character varying (255)	password character varying (255)	email character varying (255)	admin boolean
1	224366	Tomas	ola	t.fonseca@ua.pt	false
2	926927	Tiago	coldplay	tiagofonseca2002@hotmail.com	true

Depois

	id [PK] integer	username character varying (255)	password character varying (255)	email character varying (255)	admin boolean
1	224366	Tomas	\$2b\$12\$imUi4i6U7E30Jc.JJeungufh0UqVUhYpV20TpxP5iwcJjuMsUPmy	t.fonseca@ua.pt	false
2	926927	Tiago	\$2b\$12\$to7AoKbtFms67GCCyaSn8ux73qNSq5Jjx8uHe62pQizc0lINISoEMC	tiagofonseca2002@hotmail.com	true



CWE-285 - Improper Authorization

A vulnerabilidade *CWE-285* diz respeito a questões de segurança relacionadas com autenticação incorreta. Isso abrange falhas na validação de credenciais, o que pode permitir acesso não autorizado. Tais vulnerabilidades podem ser exploradas através de ataques de força bruta ou injeção de credenciais, colocando em risco a integridade e segurança dos sistemas de *software*.

A autenticação pode ser contornada ao introduzir um *URL* específico, permitindo, assim, o acesso fácil a qualquer página, incluindo o perfil do utilizador e a totalidade da sua conta.

Qualquer cliente poderá aceder ao perfil, carrinho de compras, pedidos, entre outros, de outros utilizadores. Esta é uma vulnerabilidade crítica, uma vez que possibilita a qualquer cliente alterar as informações da conta de outros utilizadores, como, por exemplo, a sua palavra-passe.

Esta situação é possível devido à ausência de verificação do *ID* do utilizador em todas as vistas, o qual é concedido aquando do processo de *Login*.

```
@views.route("/profile/<username>")
def profile(username):

    # Get ID based on the username
    id = get_id_by_username(username)

    # Return the account settings page
    return render_template("profile.html", username=username, id=id)
```




```
@views.route('/login', methods=['GET', 'POST'])
def login():

    if request.method == "POST":
        # Get the password and username and email that were set in the
        # signup session
        username = request.form.get("username")
        password = request.form.get("password")

        # Check if the username has a space
        if " " in username and len(username.split(" ")) == 2:

            # Merge the string into one username
            username = username.replace(" ", "")

        # Check if the username inserted is a email
        if "@" in username:

            # Search the username based on the email
            username = search_user_by_email(username)

        # Check if the login credentials are valid
        if validate_login(username, password) == True:
            # Set the session variables
            session["username"] = username
            session["id"] = get_id_by_username(username)
            session["admin"] = get_user_role(session["id"])
            check_database_table_exists(username.lower() + "_cart")
            check_database_table_exists(f"{username.lower()}_orders")

            return redirect(url_for("views.catalog", id=session["id"]))

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

Neste [link](#), é possível visualizar uma breve demonstração que ilustra como é possível aceder ao perfil de outro utilizador sem a necessidade de autenticação, bastando apenas inserir um *URL* específico.

Para mitigar esta vulnerabilidade, efetuamos uma verificação para garantir a presença do *ID* do utilizador quando examinamos a vista. Caso esse *ID* esteja em falta, redirecionamos a vista para a página de *Login*.



```
# This view returns the account settings page
@views.route("/profile/<username>")
def profile(username):

    if is_valid_input(username) == False:
        return render_template("index.html", message="Invalid username.")

    # Get ID based on the username
    id = session.get("id")

    if id == None:
        return redirect(url_for("views.login"))

    # Return the account settings page
    return render_template("profile.html", username=username, id=id)
```

No seguinte [link](#), é possível assistir a uma demonstração que ilustra a recusa de acesso não autorizado ao perfil de outro utilizador. Neste caso, o utilizador é direcionado para a página de *Login*, onde é necessário autenticar-se.

CWE-756 - Missing Custom Error Page

A vulnerabilidade em questão envolve a aplicação que não disponibiliza páginas de erro personalizadas aos utilizadores, expondo, assim, informações potencialmente sensíveis.

Os atacantes podem tirar partido de informações adicionais fornecidas por uma página de erro padrão para criar ataques direcionados à *framework*, base de dados ou outros recursos utilizados pela aplicação, ou ainda para assumir uma identidade falsa.

Uma abordagem mais segura consiste em definir mensagens de erro padrão apenas para utilizadores remotos.

No seguinte [link](#), é possível visualizar uma demonstração desta vulnerabilidade em ação, onde as páginas de erro personalizadas não são apresentadas.

Para mitigar este problema, implementámos páginas de erro personalizadas.

Desta forma, em caso de erro na aplicação *Web* ou falta de uma página, as páginas de erro personalizadas serão exibidas. Estas rotas estão definidas no ficheiro *app.py*.

```
# Define a custom error handler for 403 (Not Found) errors
@app.errorhandler(403)
def page_not_found(error):
    return render_template('403.html'), 403

# Define a custom error handler for 404 (Not Found) errors
@app.errorhandler(404)
def page_not_found(error):
    return render_template('404.html'), 404

# Define a custom error handler for 500 (Not Found) errors
@app.errorhandler(500)
def page_not_found(error):
    return render_template('500.html'), 500
```

Neste [link](#), é possível visualizar a demonstração após termos aplicado a solução para a vulnerabilidade em análise.



CWE-798 - Use of Hard-coded Credentials

O termo *Hard-coded* implica a inclusão de valores fixos e fundamentais para o funcionamento do sistema diretamente no código-fonte, em vez de obtê-los de uma fonte externa. Por esse motivo, em muitos casos, essa prática é considerada inadequada no desenvolvimento de *software*.

A vulnerabilidade *CWE-798* reporta que as credenciais para autenticar um utilizador no produto foram deixadas visíveis no código-fonte, ficando ao alcance de qualquer utilizador mal-intencionado.

```
def db_query(query, params=None):

    credentials = {
        "host": "Your_Machine_IP_Address",
        "dbname": "Your_Database_Name",
        "user": "Your_Database_Username",
        "password" : "Your_Database_Password",
        "port" : "Your_Database_Port (Standard PostgreSQL Port - 5432)"
    }

    # Connect to the database
    conn = psycopg2.connect(
        host=credentials["host"],
        dbname=credentials["dbname"],
        user=credentials["user"],
        password=credentials["password"],
        port=credentials["port"]
    )

    # Initiate the cursor
    cur = conn.cursor()

    # Check if there is any parameters
    if params:

        # Execute query with parameters
        cur.execute(query, params)

    else:

        # Execute query without parameters
        cur.execute(query)
```



```
# Define select_in_query as False by default
select_in_query = False

# Check if the query has SELECT
if "SELECT" in query:

    # Fetch all the data
    data = cur.fetchall()
    select_in_query = True

    # Commit the connection
    conn.commit()

    # Close the cursor
    cur.close()

    # Close the connection
    conn.close()

# Check if the query has SELECT
if select_in_query:

    # Return the requested data
    return data
```

```
def send_email_with_attachment(to, subject, body, attachment_path):
# Read Email credentials file
credentials = {
    "email": "Your_Gmail_Address",
    "password": "Your_Gmail_API_Key"
}

# Create a MIMEText object to represent the email body
msg = MIMEMultipart()
msg['From'] = credentials["email"]
msg['To'] = to
msg['Subject'] = subject

# Attach the body of the email
msg.attach(MIMEText(body, 'html'))

# Attach the PDF file as an attachment
with open(attachment_path, "rb") as pdf_file:
    pdf_attachment = MIMEApplication(pdf_file.read(), _subtype="pdf")
    pdf_attachment.add_header('Content-Disposition', f'attachment;
filename={os.path.basename(attachment_path)}')
    msg.attach(pdf_attachment)
```



```
# Connect to the SMTP server
server = smtplib.SMTP_SSL('smtp.gmail.com', 465) # Replace with your email
provider's SMTP server

try:
    # Login to your email account
    server.login(credentials["email"], credentials["password"])

    # Send the email with attachment
    server.sendmail(credentials["email"], to, msg.as_string())

except Exception as e:
    print(f"An error occurred: {str(e)}")

finally:
    # Close the connection to the SMTP server
    server.quit()

# Return True to indicate the email was sent successfully
return True
```

```
def send_email(to, subject, body):
    # Read Email credentials file
    credentials = {
        "email": "Your_Gmail_Address",
        "password": "Your_Gmail_API_Key"
    }

    # Create a MIMEText object to represent the email body
    msg = MIMEText(body, 'html')
    msg['From'] = credentials["email"]
    msg['To'] = to
    msg['Subject'] = subject

    # Attach the body of the email
    msg.attach(MIMEText(body, 'html'))

    # Connect to the SMTP server
    server = smtplib.SMTP_SSL('smtp.gmail.com', 465) # Replace with your
email provider's SMTP server

    try:
        # Login to your email account
        server.login(credentials["email"], credentials["password"])

        # Send the email
        server.sendmail(credentials["email"], to, msg.as_string())
```

```
except Exception as e:
    print(f"An error occurred: {str(e)}")

finally:
    # Close the connection to the SMTP server
    server.quit()

# Return True to indicate the email was sent successfully
return True
```

Na vulnerabilidade *CWE-798*, as credenciais de uma conta de administração foram incorporadas no código-fonte. Isso significa que, quando o produto for publicado, se algum utilizador conseguir ter acesso a essas credenciais, terá a capacidade de controlar o produto, além de poder expor informações sensíveis, recursos e funcionalidades do mesmo. A deteção desta vulnerabilidade pode ser bastante complexa, e a correção deste erro geralmente requer intervenção manual, resultando frequentemente na desativação temporária do produto.

Para resolver esta vulnerabilidade, é recomendado armazenar senhas, chaves, parâmetros de credenciais e informações sensíveis num ficheiro de configuração ou num tipo de arquivo altamente cifrado e protegido contra acessos não autorizados, incluindo outros utilizadores locais no mesmo sistema, evitando a inclusão destas informações diretamente no código-fonte.

No nosso caso, introduzimos a função `read_json(filename)` para armazenar as credenciais num ficheiro *JSON*. Quando necessário, as credenciais são obtidas a partir desse ficheiro, garantindo assim que não estão hard-coded no código-fonte.

```
def read_json(filename):

    if os.name == "nt":
        # Get the current working directory
        current_directory =
os.path.dirname(os.path.abspath(__file__)).split("\\handlers")[0]
    else:
        # Get the current working directory
        current_directory =
os.path.dirname(os.path.abspath(__file__)).split("/handlers")[0]

    full_file_path = current_directory + filename

    with open(full_file_path, "r", encoding="utf8") as file:
        data = json.load(file)
    return data
```



```
def db_query(query, params=None):

    credentials = {
        "host": "Your_Machine_IP_Address",
        "dbname": "Your_Database_Name",
        "user": "Your_Database_Username",
        "password" : "Your_Database_Password",
        "port" : "Your_Database_Port (Standard PostgreSQL Port - 5432)"
    }

    # Connect to the database
    conn = psycopg2.connect(
        host=credentials["host"],
        dbname=credentials["dbname"],
        user=credentials["user"],
        password=credentials["password"],
        port=credentials["port"]
    )

    # Initiate the cursor
    cur = conn.cursor()

    # Check if there is any parameters
    if params:

        # Execute query with parameters
        cur.execute(query, params)

    else:

        # Execute query without parameters
        cur.execute(query)

    # Define select_in_query as False by default
    select_in_query = False

    # Check if the query has SELECT
    if "SELECT" in query:

        # Fetch all the data
        data = cur.fetchall()
        select_in_query = True

    # Commit the connection
    conn.commit()

    # Close the cursor
    cur.close()
```




```
# Close the connection
conn.close()

# Check if the query has SELECT
if select_in_query:

    # Return the requested data
    return data
```



```
def send_email_with_attachment(to, subject, body, attachment_path):  
    # Read Email credentials file  
    credentials = {  
        "email": "Your_Gmail_Address",  
        "password": "Your_Gmail_API_Key"  
    }  
  
    # Create a MIMEText object to represent the email body  
    msg = MIMEMultipart()  
    msg['From'] = credentials["email"]  
    msg['To'] = to  
    msg['Subject'] = subject  
  
    # Attach the body of the email  
    msg.attach(MIMEText(body, 'html'))  
  
    # Attach the PDF file as an attachment  
    with open(attachment_path, "rb") as pdf_file:  
        pdf_attachment = MIMEApplication(pdf_file.read(), _subtype="pdf")  
  
        pdf_attachment.add_header('Content-Disposition', f'attachment;  
filename={os.path.basename(attachment_path)}')  
    msg.attach(pdf_attachment)  
  
    # Connect to the SMTP server  
    server = smtplib.SMTP_SSL('smtp.gmail.com', 465) # Replace with your email  
    provider's SMTP server  
  
    try:  
        # Login to your email account  
        server.login(credentials["email"], credentials["password"])  
  
        # Send the email with attachment  
        server.sendmail(credentials["email"], to, msg.as_string())  
  
    except Exception as e:  
        print(f"An error occurred: {str(e)}")  
  
    finally:  
        # Close the connection to the SMTP server  
        server.quit()  
  
    # Return True to indicate the email was sent successfully  
    return True
```



```
def send_email(to, subject, body):

    # Read Email credentials file
    credentials = {
        "email": "Your_Gmail_Address",
        "password": "Your_Gmail_API_Key"
    }

    # Create a MIMEText object to represent the email body
    msg = MIMEMultipart()
    msg['From'] = credentials["email"]
    msg['To'] = to
    msg['Subject'] = subject

    # Attach the body of the email
    msg.attach(MIMEText(body, 'html'))

    # Connect to the SMTP server
    server = smtplib.SMTP_SSL('smtp.gmail.com', 465) # Replace with your email
    provider's SMTP server

    try:
        # Login to your email account
        server.login(credentials["email"], credentials["password"])

        # Send the email
        server.sendmail(credentials["email"], to, msg.as_string())

    except Exception as e:
        print(f"An error occurred: {str(e)}")

    finally:
        # Close the connection to the SMTP server
        server.quit()

    # Return True to indicate the email was sent successfully
    return True
```



Ficheiro *DataBaseCredentials.json*

```
{
  "host": "Your_Machine_IP_Address",
  "dbname": "Your_Database_Name",
  "user": "Your_Database_Username",
  "password" : "Your_Database_Password",
  "port" : "Your_Database_Port (Standard PostgreSQL Port - 5432)"
}
```

Ficheiro *EmailCredentials.json*

```
{
  "email": "Your_Gmail_Address",
  "password": "Your_Gmail_API_Key"
}
```



CWE-620 - Unverified Password Change

A vulnerabilidade ocorre quando o produto não exige que o utilizador forneça a sua *password* original ou realize outra forma de autenticação ao tentar alterar a sua *password*.

Esta vulnerabilidade pode ser explorada por um atacante que, ao alterar a *password* de outro utilizador, assume o controlo da conta desse utilizador.

Uma solução eficaz pode passar por exigir a introdução da *password* original sempre que um utilizador solicitar uma alteração de *password*.

Outra opção consiste em assegurar a autenticidade do utilizador, requerendo uma confirmação por *email* ou respostas a perguntas secretas definidas aquando da criação da conta, as quais só o próprio utilizador conhece, sempre que for feito um pedido de alteração de *password*.

```
# This view is used to check if the email exists
@views.route('/update_account/<id>', methods=['POST'])
def update_account(id):

    # Set the user's account image file path
    file_path = os.getcwd()+f"/database/accounts/{id}.png"

    # Get the new uploaded user's account image
    profile_photo = request.files.get("profile_photo")

    # If there is a image
    if profile_photo:

        # Save the image to the user's account
        profile_photo.save(file_path)

    # Get the username, email and password, from the user's session
    username = request.form.get("username")
    email = request.form.get("email")
    password = request.form.get("psw")
```



```
# Check if the username field wasn't empty and occupied by another user
if username != "" and not check_username_exists(username):

    # Update the username
    update_username(id, username)

    # Set the sessions username
    session["username"] = username

else:

    # If there is a problem with the username, get the username based on the
ID    username = search_user_by_id(id)[1]

    # Check if the email field wasn't empty and occupied by another user
    if email != "" and not check_email_exists(email):

        # Update the email
        update_email(id, email)

    else:

        # If there is a problem with the username, get the username based on the
ID        email = search_user_by_id(id)[3]

        # Check if the password wasn't empty
        if password != "":

            # Update the password
            update_password(id, password)

# Return the profile page
return redirect(url_for("views.catalog", id=id))
```

Neste [link](#), é possível visualizar uma breve demonstração que exemplifica como o controlo de acesso à plataforma pode ser contornado, permitindo a alteração da *password* do utilizador sem qualquer verificação.



Para mitigar esta vulnerabilidade, procedemos à atualização da página *HTML* e *Update Account View* para incluir a introdução da *password* anterior.

```
@views.route('/update_account/<id>', methods=['POST'])
def update_account(id):

    if id == None or session.get("id") == None:
        return redirect(url_for("views.login"))

    try:

        if os.name == "nt":
            # Get the current working directory
            current_directory =
os.path.dirname(os.path.abspath(__file__)).split("\\handlers")[0]
        else:
            # Get the current working directory
            current_directory =
os.path.dirname(os.path.abspath(__file__)).split("/handlers")[0]

        accounts_directory = os.path.join(current_directory, "database",
"accounts")
        os.makedirs(accounts_directory, exist_ok=True) # Ensure the directory
exists

        file_path = os.path.join(accounts_directory, f"{id}.png").replace("\\",
"/")

        # Get the new uploaded user's account image
        profile_photo = request.files.get("profile_photo")

        # If there is an image
        if profile_photo:
            # Save the image to the user's account
            profile_photo.save(file_path)

        # Get the username, email, and password from the user's session
        username = request.form.get("username")
        email = request.form.get("email")
        password = request.form.get("psw")
        old_password = request.form.get("psw-old")
```



```
# Check if the username field wasn't empty and occupied by another user
if username != "" and not check_username_exists(username) and
is_valid_input(username):

    # Update the username
    update_username(id, username)

    # Set the session's username
    session["username"] = username

else:
    # If there is a problem with the username, get the username based on
the ID
    username = search_user_by_id(id)[1]

    # Check if the email field wasn't empty and occupied by another user
    if email != "" and not check_email_exists(email) and
is_valid_input(email):

        # Update the email
        update_email(id, email)

    else:
        # If there is a problem with the email, get the email based on the
ID
        email = search_user_by_id(id)[3]

    # Check if the password wasn't empty
    if password != "":
        # Update the password
        # Hash the password before storing it in the database


        if bcrypt.check_password_hash(search_user_by_id(id)[2],
old_password):
            hashed_password =
bcrypt.generate_password_hash(password).decode("utf-8")
            username = search_user_by_id(id)[1]
            update_password(username, hashed_password)
        else:
            return render_template("profile.html", message="Invalid
password.", username=username, id=id)

    # Return the profile page
    return redirect(url_for("views.catalog", id=id))
```




página *HTML* com um novo campo de introdução para a *password* antiga:

Account Settings


tomas

Username

Email

Old Password

Password

Repeat Password

Profile Photo



CWE-640 - Weak Password Recovery Mechanism for Forgotten Password

A vulnerabilidade *CWE-640* aborda questões de segurança relacionadas com a falta de verificação adequada de privilégios em operações sensíveis. Isso pode resultar em acesso não autorizado ou na manipulação de dados por parte de utilizadores mal-intencionados. A falta de um controlo adequado de privilégios pode dar origem a falhas de segurança significativas em sistemas de *software*.

No contexto da configuração de uma nova senha para um utilizador, recorre-se ao sistema de recuperação de senhas `reset_password()`, concebido para gerar uma nova senha, armazená-la na base de dados e enviá-la para o *email* do utilizador. No entanto, esta abordagem gera problemas graves, como falta de autenticação, exposição à interceção de *emails*, divulgação de senhas, rotação de senhas, entre outros.

```
# This route is used to let the user reset his password
@views.route("/reset-password", methods=["GET", "POST"])
def reset_password():

    # Check if the requested method is POST
    if request.method == "POST":

        # Get the user's email from the request
        email = request.form.get("email")
        # Get the username based on the email
        user = search_user_by_email(email)

        # Check if the user exists
        if user is None:

            # If the user doesn't exist, return the signup page
            return redirect(url_for("views.signup"))
        else:

            # Send recovery password to the user's email
            send_recovery_password(email)

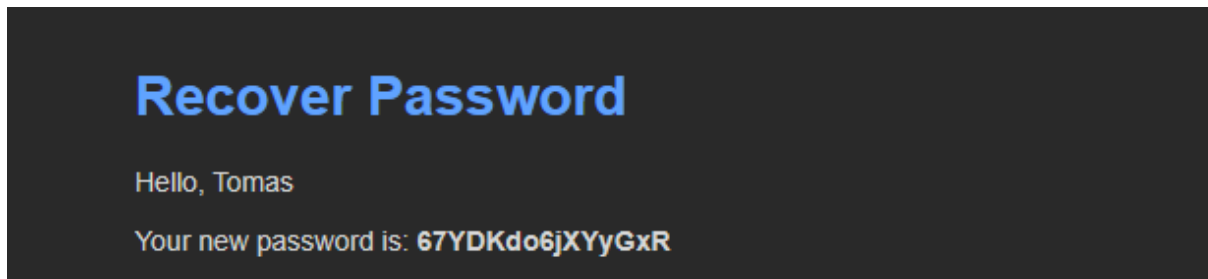
            # Return the login page
            return redirect(url_for("views.login"))
    else:

        # If it isn't, return the same page
        return render_template("reset-password.html")
```

Neste [link](#), é possível assistir a uma breve demonstração que ilustra o funcionamento do sistema de recuperação de senhas.



Na imagem que se segue, é possível observar o *email* que o utilizador irá receber com a nova senha que lhe foi atribuída.



Para abordar esta vulnerabilidade, introduzimos um novo sistema de recuperação de senhas que faz uso de *tokens* com carimbos de data/hora. Para acomodar este sistema, realizamos alterações na tabela *users* da base de dados.

```
# This route is used to let the user reset their password
@views.route("/reset-password", methods=["GET", "POST"])
def reset_password():
    if request.method == "POST":
        email = request.form.get("email")
        if is_valid_input(email) == False:
            return render_template("reset-password.html", message="Invalid
email.")

        user = search_user_by_email(email)
        if user is None:
            # If the user doesn't exist, return the signup page
            return redirect(url_for("views.signup"))
        else:
            # Generate a unique reset token
            reset_token = generate_reset_token()
            # Store the reset token in the user's record in the database
            set_reset_token_for_user(user, reset_token)
            # Send a password reset email with the token
            send_password_reset_email(email, reset_token)
            return redirect(url_for("views.login"))
    else:
        return render_template("reset-password.html")
```



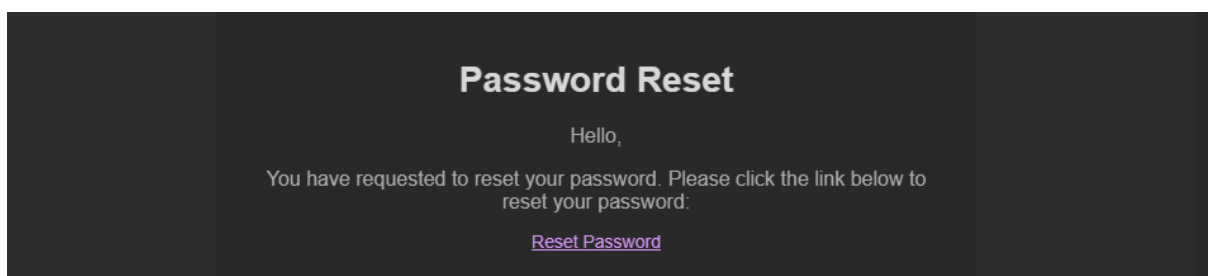
```
@views.route('/reset_password/<reset_token>', methods=['GET', 'POST'])
def reset_password_confirm(reset_token):
    # Check if the reset token is valid and not expired
    if is_valid_reset_token(reset_token):
        if request.method == 'POST':
            new_password = request.form.get('new_password')
            confirm_password = request.form.get('confirm_password')

            if new_password == confirm_password:
                # Update the user's password in the database with the new hashed
                password
                username = get_user_by_reset_token(reset_token)[1]

                if username:
                    hashed_password =
                    bcrypt.generate_password_hash(new_password).decode("utf-8")
                    update_password(username, hashed_password)

                    # Clear the reset token for security
                    clear_reset_token(username)
                    return redirect(url_for('views.login'))
            return render_template('reset_password.html', reset_token=reset_token)
        else:
            return redirect(url_for('views.login'))
```

Nas imagens que se seguem, é possível visualizar o resultado após o utilizador submeter o pedido de recuperação de senha, seguindo o mesmo procedimento demonstrado anteriormente. O utilizador receberá um *email* contendo instruções e um *link* para efetuar a reposição da senha.





127.0.0.1:5000/reset_password/9HpmhvEp9S5yy8WOPD01rmeF4x1XXGN

Reset Password

New Password

Confirm New Password

RESET PASSWORD

Or

[LOGIN](#)



Bibliografia

<https://cwe.mitre.org/>

<https://www.cvedetails.com/>