

## Project 1 -SIO

# Index

### 1. Introdução

### 2. Vulnerabilidades

- CWE - 89
- CWE - 79
- CWE - 862
- CWE - 307
- CWE - 20
- CWE - 200
- CWE - 620
- CWE - 756
- CWE - 256

### 3. Conclusão

## 1. Introdução

No nosso projeto foi desenvolvido um website de uma loja online que vende exclusivamente produtos alusivos ao DETI. Dentro do mesmo é possível criar-se um utilizador e fazer login com o mesmo, pode ainda adicionar os seus produtos preferidos a uma lista de favoritos assim como adiciona-los diretamente ao carrinho de compras. Para implementar este site foi usado com base uma aplicação em flask com base em paginas de html e uma base de dados em SQLite.

Para satisfazer os requisitos do trabalho que nos foi entregue foram criadas duas versões da aplicação, a app.py é a versão que foi sujeita a vários ataques provocados propositalmente para mostrar as suas vulnerabilidades, já a app\_sec.py é uma app que foi desenvolvida de modo a ser segura sobre todos os ataques feitos no primeiro caso.

## 2. Vulnerabilidades

Foram escolhidas entre as inúmeras possibilidades 9 CWE's para explorar neste trabalho entre elas estão:

### CWE-89

A CWE-89, comumente conhecida como "SQL-Injection", é uma vulnerabilidade bastante comum caracterizada pela neutralização imprópria de elementos especiais usados em comandos SQL. Os danos potenciais causados por SQLI são roubo de informação armazenada na base de dados ou remoção/alteração de conteúdo essencial das mesmas.

## Demonstração

Durante o processo de login se o atacante inserir um nome de utilizador correto, seguido de ' OR 1=1;-- no campo da password consegue entrar na conta do user que colocou sem ter na sua posse a password correta.

Nesse caso, a consulta SQL resultante será:

```
SELECT * FROM user WHERE username = '' OR '1'='1' AND password = '';
```

Como '1' é sempre igual a '1', a consulta acima retornará que o campo da password é verdadeiro logo aceita o login do utilizador sem requerer um password válida.

Na versão em .pdf consultar o vídeo v1.gif em [link](#)

No excerto de código fornecido, podemos identificar a presença de uma vulnerabilidade de SQL na nossa API. A vulnerabilidade ocorre na seguinte linha:

```
result = db.session.execute("SELECT * FROM user WHERE username = '" +  
username + "' AND password = '" + password + "';").fetchall()
```

Neste trecho de código, o valor das variáveis username e password fornecidas pelo utilizador não é devidamente validado antes de serem verificados na database. Isso significa que um atacante pode manipular essas variáveis de entrada para injetar código SQL malicioso.

## Correção

De modo a corrigir este problema na nossa app\_sec foi implementado um método diferente de acesso à database, no novo método o acesso é feito com recurso ao SQLAlchemy.

Como podemos ver no seguinte trecho de código.

```
if not user:  
    flash('User does not exist.', 'error')  
    return redirect(url_for('auth.login'))  
  
if not check_password_hash(user.password, password):  
    flash('Please check your login details and try again.')  
    user.increment_failed_login_attempts()  
    db.session.commit()  
    return redirect(url_for('auth.login'))
```

Usámos também a biblioteca werkzeug.security para verificar se o input colocado corresponde à password esperada, uma vez que todas as passwords são armazenadas na base de dados de forma cifrada.

## CWE-79

A CWE-79, comumente conhecida como "Cross-site Scripting", é caracterizada pela neutralização imprópria dos comandos de input feitos pelo utilizador antes deste se tornar em um output que pode ser usado/consultado por outros utilizadores.

### Demonstração

Na barra de pesquisa onde é possível procurar por nomes de produtos, o input que o utilizador coloca é passado diretamente para um campo com `id_search_result` usando para isso o `.html()`, o que provoca o input do utilizador ser renderizado como html.

```
$("#search_result").html("Search results for: " + $('#search').val());
```

A vulnerabilidade foi explorada dentro da nossa app como demonstra este vídeo:

Na versão em .pdf consultar o vídeo v2.gif em [link](#)

### Correção

De modo a evitar ataques do tipo, a solução foi implementar o seguinte código:

```
$("#search_result").text("Search results for: " + $('#search').val());
```

De forma que o input do utilizador seja interpretado como **texto** ao invés de **html**.

## CWE-862

A CWE-862, esta vulnerabilidade verifica-se quando o servidor não verifica se o usuário tem ou não permissão para fazer determinadas ações.

Esta falta de cuidado pode levar a que qualquer usuário mal intencionado acesse, altere ou divulgue informação privada.

### Demonstração

No nosso site caso o utilizador seja admin, este tem acesso a uma página para adicionar produtos à loja. Porém na app vulnerável não é feita a verificação de cargo dos utilizadores, permitindo assim a qualquer utilizador aceder a essa página.

```
@main.route('/addproduct', methods=['GET'])
@login_required
def add_product():
    return render_template('addproduct.html')
```

No trecho de código em cima é possível observar que qualquer utilizador que tente aceder a esse endpoint, vai conseguir fazê-lo sem passar por qualquer verificação.

## Correção

De forma a solucionar essa vulnerabilidade foi implementada na nossa app segura uma verificação de cargo aos utilizadores que tentem aceder a esse endpoint.

```
@main.route('/addproduct', methods=['GET'])
@login_required
def add_product():
    user=User.query.filter_by(id=current_user.id).first()
    if user.isAdmin == False:
        return redirect(url_for('main.index'))
    else:
        return render_template('addproduct.html')
```

## CWE-307

A CWE-307 verifica-se quando não existe segurança contra ataques de brute force, permitindo aos atacantes colocar diversas vezes credenciais de acesso inválidas num curto período de tempo.

## Demonstração

No primeiro video podemos observar que o utilizador colocando varias vezes a palavra passe errada, a app vulneravel nao impõe qualquer medida de segurança contra brute force permitindo ao utilizador estar sempre a experimentar passwords

Na versão em .pdf consultar o vídeo v6.gif em [link](#)

## Correção

No segundo video podemos observar que após 5 tentativas mal sucedidas o servidor bloqueia durante alguns segundos a página, protegendo assim contra ataques de brute force.

Na versão em .pdf consultar o vídeo v5.gif em [link](#)

## CWE-20

A CWE-20, esta vulnerabilidade verifica-se quando a API não faz a validação correta de dados que são introduzidos pelo utilizador na página web.

## Demonstração

Ao adicionar um produto ao carrinho é dado ao utilizador um campo onde pode inserir a quantidade de unidades do produto que este pretende comprar. Como a API não faz a devida verificação do input "quantidade", esta permite ao utilizador inserir uma quantidade negativa de produtos o que faz com que ao invés de pagar pelos produtos o atacante receba essa mesma quantia.

Um exemplo do que foi enunciado:

Na versão em .pdf consultar o vídeo v3.gif em [link](#)

O erro está presente no seguinte trecho de código:

```
if product:
    cart_item = Cart.query.filter_by(user_id=current_user.id,
product_id=product_id).first()
    if cart_item:
        cart_item.quantity = int(request.form.get('quantity'))
        flash("item quantity updated")
        db.session.commit()
```

O erro acontece pois a quantidade introduzida pelo utilizador não é validada pelo servidor, permitindo assim a introdução de valores negativos.

## Correção

O código abaixo apresenta a correção do erro enunciado pois o valor introduzido pelo utilizador passa por um processo de validação antes de ser introduzido na database.

```
if product:
    cart_item = Cart.query.filter_by(user_id=current_user.id,
product_id=product_id).first()
    if cart_item:
        cart_item.quantity = int(request.form.get('quantity'))
        if cart_item.quantity == 0:
            cart_item.quantity = 1
        if cart_item.quantity < 0:
            cart_item.quantity = 1
        flash("item quantity updated")
        db.session.commit()
```

## CWE-200

A vulnerabilidade CWE-200, verifica-se quando o servidor expõe informação sensível a utilizadores não autorizados a recebê-la.

## Demonstração

Esta CWE é inúmeras vezes aplicada conjuntamente com a CWE-89 (SQLI), pois no caso explorado no nosso site, caso o utilizador errasse o nome de usuário no processo de login, era emitida a mensagem descritiva **"user does not exist."**, enquanto se o erro estivesse apenas na password a mensagem emitida será, **"Please check your login details and try again"**, ao ter mensagens de erro diferentes o site indiretamente dá a informação que aquele nome de usuário é válido.

Aliado com SQLI esta informação é suficiente para que o atacante faça login no site.

Erro está presente no seguinte seguimento de código:

```
@auth.route('/login', methods=['POST'])
def login_post():
    username = request.form.get('username')
    password = request.form.get('password')

    result = db.session.execute(
        "SELECT * FROM user WHERE username = '" + username + "' AND
password = '" + password + "';").fetchall()

    user = User.query.filter_by(username=username).first()
    if not user:
        print('User does not exist.', 'error')
        return redirect(url_for('auth.login'))

    if not result:
        print('Please check your login details and try again.')
        return redirect(url_for('auth.login'))

    login_user(user)
    return redirect(url_for('main.index'))
```

## Correção

A correção do problema em cima enunciado pode ser feita alterando as mensagens de erro apresentadas quando são introduzidos dados de login incorretos.

Ao alterar as mensagem de erro para uma mais geral deixa-mos de dar ao atacante informação que poderá comprometer o site noutros aspetos.

Para uma versão segura atualizamos o código em cima:

```
def login_post():
    username = request.form.get('username')
    password = request.form.get('password')

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

    if not user:
        flash('User does not exist.', 'error')
```

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

    if user.failed_login_attempts >= 5:
        time.sleep(60)
        user.reset_failed_login_attempts()
        db.session.commit()
        flash('Your account is block, try angain later!', 'error')
        return redirect(url_for('auth.login'))

    if not check_password_hash(user.password, password):
        flash('Please check your login details and try again.')
        user.increment_failed_login_attempts()
        db.session.commit()
        return redirect(url_for('auth.login'))

    user.reset_failed_login_attempts()
    db.session.commit()
    login_user(user)
    return redirect(url_for('main.index'))
```

## CWE-620

A CWE-620, esta vulnerabilidade está presente quando a API permite a um utilizador atualizar a password sem precisar de introduzir a password que está a ser substituída.

## Demonstração

Na app vunerável o servidor permite alterar a password do utilizador sem ser necessário confirmar a password atual, o que pode levar a um atacante mudar a password da conta sem saber a password atual.

 Alt text

Na versão em .pdf consultar a imagem i2.png em [link](#)

## Correção

Na app segura o servidor não permite alterar a password sem antes colocar a password atual. Prevenindo assim possiveis roubos de contas.

 Alt text


Na versão em .pdf consultar a imagem i3.png em [link](#)

## CWE-756

A CWE-756, esta vulnerabilidade refere-se à falta de uma página de error própria do servidor. Quando esta página não é implementada a página que é mostrada aos utilizadores é default e contém informação sensível que pode ser usada contra o servidor em questão.

## Demonstração

Caso o utilizador tente aceder a um produto que não existe a página de erro a página de erro retornada será a seguinte:

 Alt text

Na versão em .pdf consultar a imagem 404-.png em [link](#)

A imagem anteriormente exposta permite aos atacantes receberem informações referentes à database.

## Correção

De modo a corrigir esta vulnerabilidade deve ser criada uma página de erro default que não tenha qualquer informação relevante para um possível atacante.

Para que a página mostrada na ocorrência de um erro seja a pretendida foi acrescentado o seguinte segmento de código na API.

```
def product(product_id):  
    product = Product.query.filter_by(id=product_id).first()  
    if not product:  
        return render_template('404.html')
```

Assim a cada ocorrência de erro a página a ser mostrada será:

 Alt text

Na versão em .pdf consultar a imagem 404.png em [link](#)

## CWE-256

A CWE-256, esta vulnerabilidade pode ser explorada quando a database seja comprometida o atacante pode ter acesso facilmente a todas as passwords dos utilizadores.

## Demonstração

Como é possível observar na imagem seguinte da database da app vulnerável as passwords estão guardadas em **plain text**, caso o site sofra um ataque de SQLI que consiga ter acesso à tabela de users da database, o atacante fica com acesso às passwords de todos os utilizadores.

 Alt text

Na versão em .pdf consultar a imagem database.png em [link](#)

## Correção

Já na nossa app segura o problema foi resolvido com recurso à biblioteca `werkzeug.security` que permite encriptar as passwords antes de as guardar na tabela da database.

 Alt text

Na versão em .pdf consultar a imagem database2.png em [link](#)



### 3. Conclusão

Tendo em conta o que foi mencionado anteriormente, este trabalho permitiu não só aprofundar os nossos conhecimentos de backend e frontend relacionados com o desenvolvimento web como também compreender e solucionar vulnerabilidades que existem na criação de um website. O projeto foi muito proveitoso para o grupo sendo de realçar a importância deste método de avaliação