

Project 1 – Vulnerable eHealth App

Universidade de Aveiro

Segurança Informática e nas Organizações 2022-2023

Catarina Barroqueiro (103895)

Ricardo Covelo (102668)

Rui Campos(103709)

Telmo Sauce (104428)



Introdução

Este relatório foi desenvolvido no âmbito da unidade curricular de Segurança Informática e nas Organizações, tendo como objetivo expor, explicar e demonstrar todas as vulnerabilidades testadas e posteriormente corrigidas no site implementado.

Testámos um total de 8 vulnerabilidades:

- CWE 89 - SQL Injection
- CWE 79 - Cross-Site Scripting
- CWE 256 - Armazenamento de uma password em texto simples
- CWE 620 - Password sem verificação
- CWE 20 - Inserção de dados sem validação
- CWE 311 - Dados sensíveis armazenados sem encriptação
- CWE 549 - Falha na máscara da password
- CWE 756 - Página de erro ausente

Descrição do projeto

Neste trabalho o nosso objetivo foi desenvolver uma página web simples para uma clínica de saúde chamada *eHealth Corp*. O nosso *Frontend* mostra informações básicas sobre os serviços, mostrar também a possibilidade de entrar em contato com a clínica, na secção 'Contactos', mostra como solicitar consultas com médicos e, por fim, dá também a possibilidade de fazer o *download* dos resultados dos exames de um utilizador fornecendo um código. O nosso *Backend* contém uma base de dados que serve para armazenar as informações, por exemplo, o *username* e a *password*, dos utilizadores.

Assim, é possível criar utilizadores registando-se no website e fazer login logo de seguida com as credenciais colocadas durante o registo. Essas credenciais ficam armazenadas na nossa Base de Dados. É possível o utilizador alterar a sua palavra-passe, recorrendo à sua Área de Cliente. É possível também o utilizador com *login* feito realizar comentários acerca do que o mesmo quiser carregando em 'Help' e depois em 'Comments'. Outra funcionalidade do nosso site é a possibilidade de marcar uma consulta com um dos nossos doutores carregando em 'Appointments', seguido de 'Book Here', onde o utilizador poderá escolher o tipo de especialidade, a Data, a Hora, o Doutor e, por fim, a Razão que o leva a consultar o médico.

É importante realçar que quando um utilizador que não está *loggen in* tenta, por exemplo, realizar um comentário, este apenas poderá visualizar os comentários efetuados pelos utilizadores que estão *logged in*.

Para a nossa *app.py*, é possível verificar que não existe a sanitização do input, logo é possível realizar *SQLInjection* no login, na *search bar* ou mesmo nos comentários efetuados pelo utilizador. Como a *search bar* não tem qualquer tipo de sanitização, é possível um atacante ejetar código *JavaScript* nos mesmos.

Para a nossa *app_sec.py* que é a nossa app segura, podemos fazer uso de todas as funções que a nossa aplicação tem, sem introduzir qualquer tipo de vulnerabilidade.

Por fim, um último aspeto a salientar no nosso trabalho, é o facto de a nossa app insegura contém um *bug* na interação do *Chrome* com a extensão do *Flask session* e, por isso, a app insegura só irá correr bem noutro navegador, como por exemplo, o *Microsoft Edge* ou o *Mozilla Firefox*.

CWEs

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') **SCORE = 6,4**

O CWE-89, mais conhecido por 'SQL Injection', refere-se à não neutralização ou neutralização imprópria de elementos especiais que podem modificar o comando SQL. Este tipo de vulnerabilidade é bastante utilizada para alterar a lógica de consulta para ignorar as verificações de segurança ou para inserir instruções adicionais que modificam o banco de dados de *backend*, possivelmente incluindo a execução de comandos do sistema.

Fix das Vulnerabilidades

Para que o atacante não consiga realizar uma SQL Injection, o que fizemos foi alterar o formato de como as informações vão para a database. Na aplicação insegura as informações são enviadas como é visível na Figura 1, o que faz com que seja possível efetuar o ataque.

```
db=sql.connect("webDB.db")
result=db.execute("SELECT * FROM users WHERE name='"+name+"' OR email='"+emailAddress+"';")
data=result.fetchall()
```

Figura 1- Site inseguro

Para a versão segura, o que fizemos foi modificar o trecho de código apresentado anteriormente para o que está apresentado na Figura 2.

```
result=db.execute("SELECT * FROM users WHERE name = ? OR email = ?", (name, emailAddress))
data=result.fetchall()
```

Figura 2- Site seguro

Assim, se o atacante tentar realizar um ataque de 'SQL Injection', todos os tipos de elementos especiais inseridos pelo mesmo já não irão funcionar.

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

SCORE = 6,1

O CWE-79, mais conhecido por 'Cross Site Scripting', refere-se à não neutralização ou neutralização imprópria do *input* durante a geração da página *web*. Este tipo de vulnerabilidades tira partido do código que está em conteúdo do website, podendo executar o código do lado do utilizador involuntariamente.

Fix das Vulnerabilidades

```
<div class="container px-5 my-5">
  {% for comment in comments :%}
    <div class="card mb-4">
      <div class="card-body">
        <h5 style="color: □ black">{{ comment[0]|safe }}</h5>
        <div class="small text-muted">
          {{ comment[1]|safe }}
        </div>
      </div>
    </div>
  {%endfor%}
```

Figura 3- Código do site inseguro

Para que o atacante não realize um ataque de 'Cross-Site-Scripting', teremos de retirar certos caracteres do nosso código. No trecho de código apresentado na Figura 3, estamos a realizar um ciclo *for* e, assim, carregar cada um dos comentários para o site.

O *comment* apresentado funciona com um *array* de *arrays*, ou seja, é do seguinte tipo `[[Nome, Comentário], [Nome, Comentário]]`, logo o *comment[0]* é o nome da pessoa que comentou e o *comment[1]* é o comentário efetuado por essa pessoa.

Para que seja possível o atacante realizar o ataque, neste trecho de código o mais importante está no '|safe' (depois do *comment[1]*), porque a ferramenta utilizada (*Flask*) bloqueia todo o tipo de símbolos que possam gerar código. Com o '|safe' o *Flask* para de os bloquear podendo, assim, ser possível realizar o ataque. Por isso, para o site ficar seguro basta apenas retirar o '|safe', como é possível visualizar na Figura 4.

```
<div class="container px-5 my-5">
  {% for comment in comments :%}
    <div class="card mb-4">
      <div class="card-body">
        <h5 style="color: □ black">{{ comment[0] }}</h5>
        <div class="small text-muted">
          {{ comment[1] }}
        </div>
      </div>
    </div>
  {%endfor%}
```

Figura 4- Código sem o '|safe'

CWE-256: Plaintext Storage of a Password

SCORE = 7,5

O CWE-256 refere-se à não neutralização ou neutralização imprópria do *input* durante a geração da página *web*. Este tipo de vulnerabilidades ocorrem quando uma palavra-passe é armazenada em texto simples nas propriedades, no arquivo de configuração ou na memória de uma aplicação. Armazenar uma palavra-passe de texto simples num arquivo de configuração permite que qualquer pessoa que possa ler o arquivo tenha acesso ao recurso protegido por essa mesma palavra-passe.

Fix das Vulnerabilidades

Para evitar o uso desta vulnerabilidade, o que fizemos foi encriptar todas as informações dos utilizadores no momento do registo da conta.

No site não seguro, o que fazemos é apenas enviar os dados que nos dão para a base de dados, como é visível na Figura 5, sem nenhum tipo de encriptação.

```
#FEIHO
@app.route("/register" , methods=["POST", "GET"])
def register(*args):

    if len(args) != 0:
        return render_template('register.html', error=args[0])

    if(request.method == "POST"):
        name = request.form["Name"]
        emailAddress = request.form["EmailAddress"]
        password = request.form["Password"]
        db=sql.connect("webDB.db")
        result=db.execute("SELECT * FROM users WHERE name='"+name+"' OR email='"+emailAddress+"'")
        data=result.fetchall()
        if(data!=[]):
            if(data[0][1]==name):
                print("Username already exists")
                return register(1)
            else:
                print("email")
                return register(2)
        else:
            session['name']=name
            db.execute("INSERT INTO users VALUES (NULL,'"+name+"','"+emailAddress+"','"+password+"');")
            db.commit()
            db.close()
            return profile()
    else:
        return render_template('register.html')
```

Figura 5- Envio de dados sem nenhum tipo de encriptação

Com o uso deste código, a base de dados iria ficar como é mostrado na Figura 6.

	id	name	email	password
	Filter	Filter	Filter	Filter
1	1	hey	hey@w.w	hey
2	2	cat	cata@u...	212
3	3	ricar...	rui@ua.pt	123
4	4	admin	admin...	123

Figura 6- Base de dados sem dados encriptados

Para evitar este tipo de vulnerabilidades, no nosso site seguro, como já referimos em cima, o que fizemos foi encriptar a informação fornecida para o utilizador e só depois enviá-la para a base de dados, como é possível verificar na Figura 7. A encriptação foi feita pela função mostrada na Figura 8.

```
@app.route("/register", methods=["POST", "GET"])
def register(*args):

    if len(args) != 0:
        return render_template('register.html', error=args[0])

    if(request.method == "POST"):
        nome = request.form["name"]
        emailAddress = request.form["EmailAddress"]
        password = request.form["Password"]
        db=sql.connect("webDB.db")

        password=hash(password)
        nome=hash(nome)
        emailAddress=hash(emailAddress)

        result=db.execute("SELECT * FROM users WHERE name = ? OR email = ?", (name, emailAddress))
        data=result.fetchall()
        if(data!=[]):
            if(data[0][1]==name):
                print("Username already exists")
                return register(1)
            else:
                print("email")
                return register(2)
        else:
            session['name']=nome
            db.execute("INSERT INTO users (name, email, password) VALUES (?, ?, ?)", (name, emailAddress, password))
            db.commit()
            db.close()
            return profile()
    else:
        return render_template('register.html')
```

Figura 7- Encriptação do nome, email e password

```
def hash(input):
    input=bytes(input, 'utf-8')
    digest = hashes.Hash(hashes.SHA256())
    digest.update(input)
    return digest.finalize()
```

Figura 8- Função para encriptar os dados

Assim, se o atacante entrar na base de dados não irá conseguir ver absolutamente nada, como mostrado na Figura 9. Mesmo que o mesmo tente captar mensagens cliente-servidor também não irá conseguir.

	id	name	email	password
	Filter	Filter	Filter	Filter
1	1	BLOB	BLOB	BLOB
2	2	BLOB	BLOB	BLOB
3	3	BLOB	BLOB	BLOB

Figura 9- Base de dados com os dados encriptados

CWE-620: Unverified Password Change

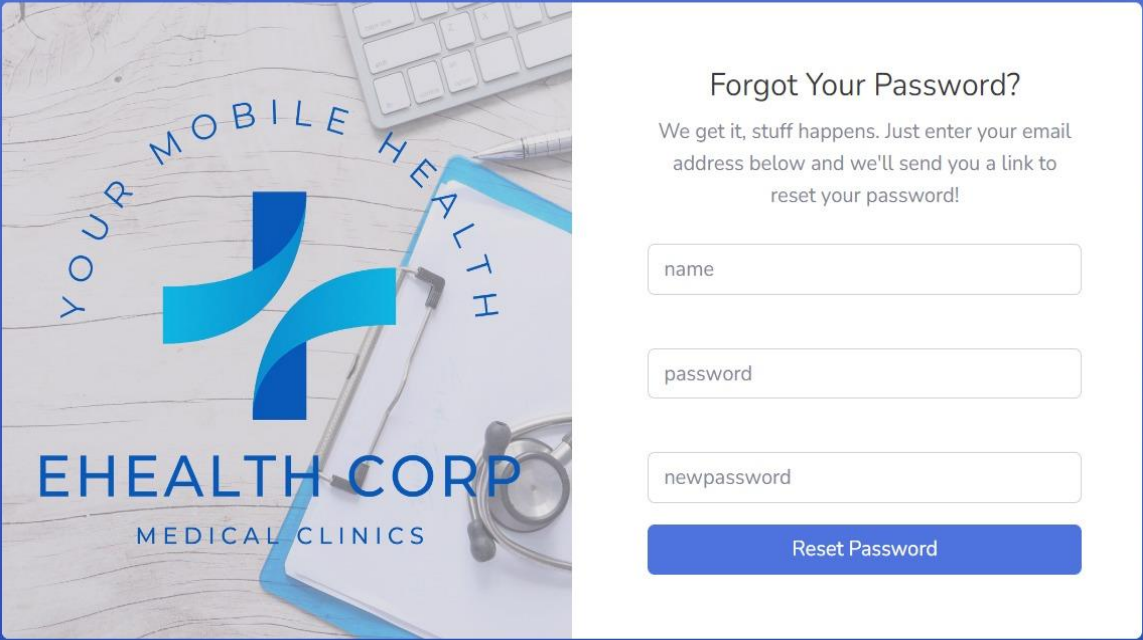
SCORE = 7,3

O CWE-620 refere-se ao momento em que, ao definir uma nova palavra-passe para um utilizador, o produto exibido não exija o conhecimento da sua palavra-passe original ou, de forma alternativa, o uso de outra forma de autenticação. Este tipo de vulnerabilidades pode ser usado por um atacante para alterar as palavras-passe de outro utilizador e, caso consiga aceder à conta do utilizador, este irá conseguir negar o acesso à mesma sem muito esforço.

Fix das Vulnerabilidades

Para evitar o uso desta vulnerabilidade, apenas foi preciso adicionar um parâmetro, o de colocar a *password* que irá ser alterada. Assim que tudo esteja correto (*password* antiga), a *password* do respetivo utilizador será alterada.

O formulário apresentado é o seguinte:



The image shows a web form titled "Forgot Your Password?" for "EHEALTH CORP MEDICAL CLINICS". The form is set against a blue background with a medical-themed graphic on the left. The graphic includes a stethoscope, a clipboard, and the text "YOUR MOBILE HEALTH" and "EHEALTH CORP MEDICAL CLINICS". The form itself is white and contains the following elements:

- Title: "Forgot Your Password?"
- Text: "We get it, stuff happens. Just enter your email address below and we'll send you a link to reset your password!"
- Input fields: "name", "password", and "newpassword".
- Button: "Reset Password".

Figura 10- Formulário do site seguro

Assim, para a *password* de um utilizador ser alterada, quem a está a alterar necessita de saber a *password* anterior. Deixamos assim de ter uma vulnerabilidade.

CWE-20: Improper Input Validation

SCORE = 6,1

O CWE-20 refere-se ao momento em que, o produto, ao receber *input* ou dados de entrada, não os valida ou valida incorretamente se esse *input* contém as propriedades necessárias para processar os dados de forma correta e segura. Este tipo de vulnerabilidades pode ser usado como uma técnica para verificar entradas potencialmente perigosas, a fim de garantir que as entradas sejam seguras para processamento dentro do código ou para comunicar com outros componentes.

Fix das Vulnerabilidades

Para garantir que não ocorrem quaisquer ataques devido a esta vulnerabilidade, na página de *login*, onde são introduzidos os dados necessários (*username* e *password*) foi aplicado um padrão que garante que não são inseridos caracteres especiais nesses campos, como é possível verificar na figura 11.

```
<form class="user" method="post">
  <br>
  <div class="form-group">
    <input type="text" pattern="^[a-zA-Z0-9]+$" class="form-control form-control-user"
      id="exampleInputEmail" aria-describedby="emailHelp"
      placeholder="name" name="name">
    </div>
  <br>
  <div class="form-group">
    <input type="password" pattern="^[a-zA-Z0-9]+$" class="form-control form-control-user"
      id="exampleInputPassword" placeholder="Password" name="Password">
    </div>
  <br>
  <input class="btn btn-primary btn-user btn-block" type="submit" value="Login">
  <br>
  <br>
</form>
```

Figura 11- Padrão para não serem introduzidos caracteres especiais

CWE-311: Missing Encryption of Sensitive Data

SCORE = 6,8

O CWE-311 consiste na não encriptação de informação crítica antes da mesma ser armazenada, deste modo a informação é guardada sem qualquer garantia de confidencialidade, integridade ou responsabilidade que a encriptação garante.

FIX DAS VULNERABILIDADES:

De modo a evitar o problema explicado implementámos *Hashing* de maneira a que os dados sejam armazenados de forma segura e viável, tanto no login como na página de registo. Mesmo que o atacante consiga aceder à base de dados e extrair informação esta será inútil pois trespassará como bit aleatórios. Evitámos este problemas implementando o código que é possível verificar nas figura 12 e 13.

```
@app.route("/login", methods=["POST", "GET"])
def login(*args):

    if len(args) != 0:
        return render_template('login.html', error=args[0])

    db=sql.connect("webDB.db")
    if(request.method == "POST"):
        name = request.form["name"]
        password = request.form["Password"]

        password = hash(password)
        nome=hash(name)

        result=db.execute("SELECT * FROM users WHERE name = ? AND password = ?", (nome, password))

        if result.fetchall():
            db.close()
            session['name']=name
            return index()
        else:
            db.close()
            return login(1)
    else:
        return render_template('login.html') I
```

Figura 12- password e nome enviados para a função 'hash' no login

```
@app.route("/register", methods=["POST", "GET"])
def register(*args):

    if len(args) != 0:
        return render_template('register.html', error=args[0])

    if(request.method == "POST"):
        nome = request.form["name"]
        emailAddress = request.form["EmailAddress"]
        password = request.form["Password"]
        db=sql.connect("webDB.db")

        password=hash(password)
        name=hash(nome)
        emailAddress=hash(emailAddress)

        result=db.execute("SELECT * FROM users WHERE name = ? OR email = ?", (name, emailAddress))
        data=result.fetchall()
        if(data!=[]):
            if(data[0][1]==name):
                print("Username already exists")
                return register(1)
            else:
                print("email")
                return register(2)
        else:
            session['name']=nome
            db.execute("INSERT INTO users (name, email, password) VALUES (?, ?, ?)", (name, emailAddress, password))
            db.commit()
            db.close()
            return profile()
    else:
        return render_template('register.html')
```

Figura 13- password e nome enviados para a função 'hash' no registo

Tudo isto resulta da ação da função *hash* que faz com que o valor recebido em texto seja convertido para bits encryptados, tendo nós utilizado SHA256.

```
def hash(input):  
    input=bytes(input, 'utf-8')  
    digest = hashes.Hash(hashes.SHA256())  
    ⚡ digest.update(input)  
    return digest.finalize()
```

Figura 14- Função 'hash'

CWE-549: Missing Password Field Masking

SCORE = 6,6

Fix das Vulnerabilidades

De modo a evitar os constrangimentos relativos à falta de proteção na introdução da *password* foi adicionada uma máscara à mesma:

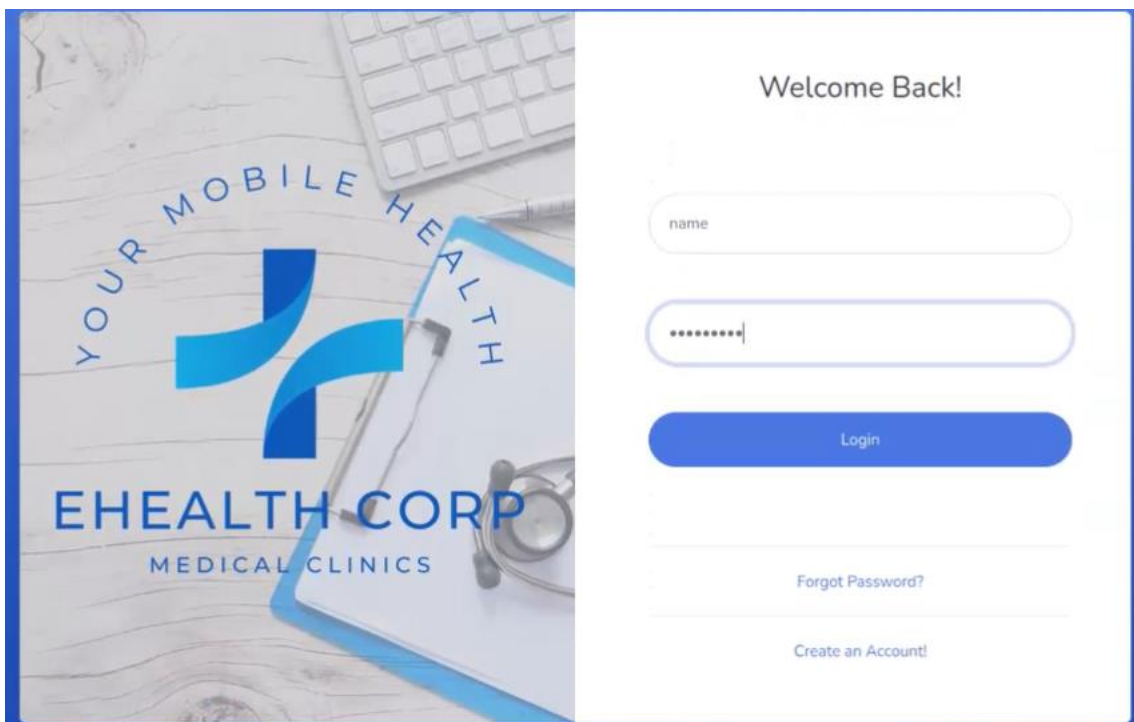


Figura 15- Introdução da password com máscara

```
<div class="form-group">
  <input type="password" pattern="^[a-zA-Z0-9]+$" class="form-control form-control-user"
    id="exampleInputPassword" placeholder="Password" name="Password">
</div>
```

Figura 16- código para aplicação da máscara na introdução da password

CWE-756: Missing Custom Error Page

SCORE = 4,9

De modo a corrigir a exposição de dados vulneráveis sempre que sucede um erro, foi atribuído valor 'false' à *flag debug* aquando da definição do modo de execução da *main* da App (figura 18):

```
if(__name__ == "__main__"):
    app.run(debug=False)
```

Figura 18 – debug=False

Além disso, caso o utilizador tente aceder a uma página que não exista, de modo a evitar que o sistema exponha todas as páginas e ficheiros existentes relacionados à página em causa foi ainda adicionada a exceção abaixo demonstrada:

```
@app.errorhandler(404)
def page_not_found(e):
    # note that we set the 404 status explicitly
    return render_template('404.html'), 404
```

Figura 19 – Page not found