

Project 1 - Vulnerabilities

Universidade de Aveiro

Segurança Informática e nas Organizações 2020-2021

Nuno Cunha (98124), Filipe Silveira (97981), Nuno Matos (97915), Ana Rosa (98678)



O relatório desenvolvido é composto por quatro vulnerabilidades.

A primeira é a CWE 89 (SQL injection) onde foram apresentadas três instâncias dessa vulnerabilidade no nosso trabalho.

As restantes vulnerabilidades referidas foram as CWE-79 (Cross-Site Scripting), CWE-22 e CWE-522, sendo que em todas é dada uma explicação do que consistem.

É de notar que no final das explicações de todas as vulnerabilidades foi acrescentado uma secção de Extras onde foram referidos alguns assuntos do nosso site Tech Store.

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

“Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data. This can be used to alter query logic to bypass security checks, or to insert additional statements that modify the back-end database, possibly including execution of system commands.”

“SQL injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.”

Vulnerabilidade na página de login

A vulnerabilidade que será descrita está na página de autenticação.

Neste caso é aplicado SQL injection básico que consiste em manipular o valor de uma das variáveis da query original criada e dessa forma a query manipulada irá permitir ao invasor entrar numa conta que não lhe pertence.

➤ Descrição de como aplicar a vulnerabilidade no site não seguro

Na página do login, o username e a palavra-passe não estão protegidos. Desta forma, o invasor ao saber o nome do utilizador e manipulando o valor da palavra-passe consegue entrar na conta.

A query utilizada na aplicação não segura é a seguinte:

```
"SELECT * from users WHERE user='"+user+"' AND pass='"+password+"';"
```

Figura 1

Se o invasor pretender atacar o utilizador 'bob' (username=bob) e colocar na palavra-passe, por exemplo, password=bob' or 1=1; --, a query passará a ser a seguinte:

```
SELECT * from users WHERE user=' bob' AND pass=' bob' or 1=1; -- ';
```

Como na segunda condição existe o valor lógico OR pass='bob' or 1=1 o que acontece é que se pelo menos um desses valores for verdadeiro então a palavra-passe passa no teste. Logo, como 1=1 é sempre verdade, o invasor consegue entrar na conta do utilizador bob.

Na prática o que acontece é o seguinte:

Login

Username

Password

Figura 2

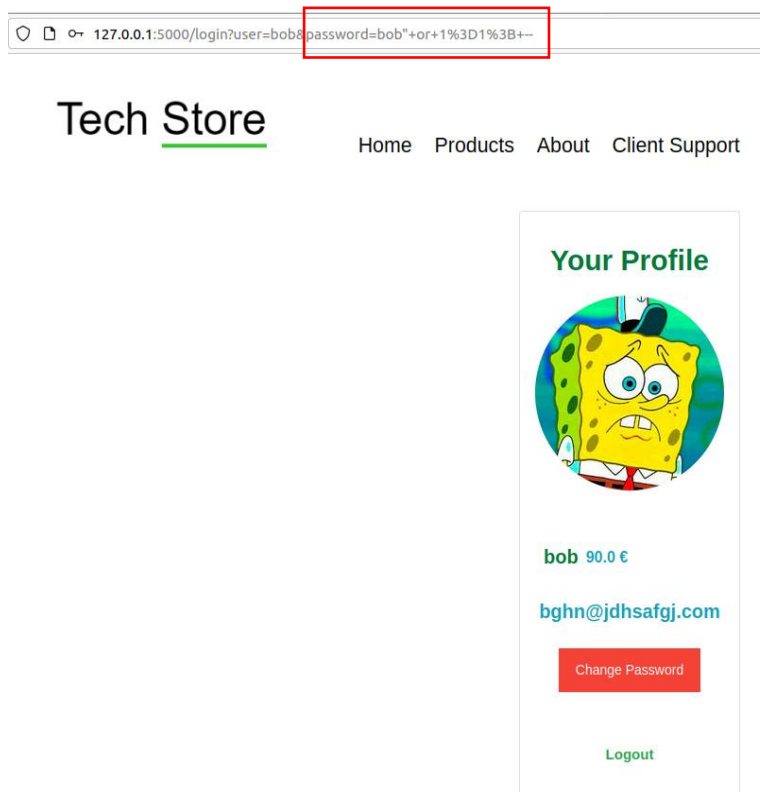


Figura 3

Concluindo, ao ter um alvo, isto é, ao saber o nome da pessoa que se quer atacar, e alterando a query através do valor da palavra-passe consegue-se facilmente passar a barreira da autenticação.

➤ Impacto da vulnerabilidade

Este tipo de vulnerabilidade é bastante fácil de aplicar em situações em que as autenticações do utilizador não estão corretamente protegidas. Apesar de ser SQL injection básico pode causar numerosas consequências à pessoa que sofreu o ataque.

Uma das consequências mais graves é de facto o roubo de identidade. Ora, o roubo de identidade leva a bastantes complicações, pois o hacker pode causar bastantes infrações como, por exemplo, roubo de dinheiro e obtenção de informação confidencial. Uma outra consequência seria, por exemplo, disponibilizar informação confidencial da própria vítima, o que poderia levar a outros problemas sérios.

➤ Comparação do site seguro com o site não seguro

Autenticação não segura

```

63 @app.route("/login",methods=['GET']) #! atenção ao erro do GET # Method Not Allowed
64 def login():
65     user=request.args.get('user', '')
66     password=request.args.get('password', '')
67
68     db = sql.connect("webDB.db")
69     result = db.execute("SELECT * from users WHERE user='"+user+"' AND pass='"+password+"'")
70     if result.fetchall():
71         db.close()
72         session['username'] = user
73         return account()
74     else:
75         db.close()
76         return login_page(1)

```

Figura 4 – Código da função login da aplicação não segura

Autenticação segura

```

86 @app.route("/login",methods=['POST'])
87 def login():
88     user=request.form['user']
89     password_input=request.form['password']
90
91     result = hashlib.sha256(password_input.encode())
92     password=result.hexdigest()
93
94     db = sql.connect("webDB.db")
95     result = db.execute("SELECT user,pass FROM users WHERE user=?",(user,))
96     data = result.fetchall()
97     db.close()
98     print(data)
99     if data != []: # user existe and password correta
100         if data[0][1] == password:
101             session['username'] = user
102             return account()
103
104     return login_page(1) #erro user n existe/pass errada

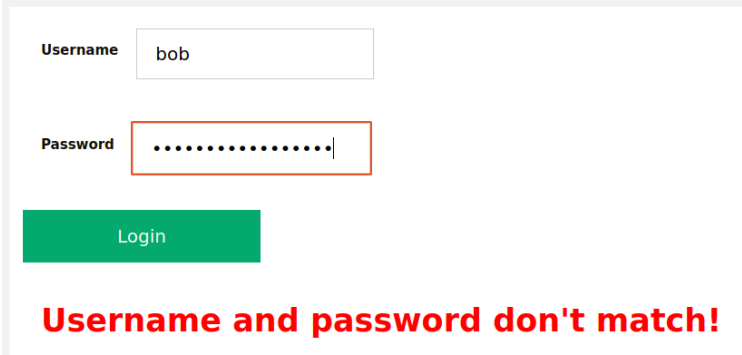
```

Figura 5 – Código da função login da aplicação segura

A primeira diferença que se pode observar é na linha 63(app não segura) e 86 (app segura). Basicamente o que acontece é que o método GET torna a autenticação mais vulnerável do que o método POST. O método GET permite enviar uma mensagem ao servidor e o servidor guarda essa informação e devolve-a, o que não é bom para o caso de logins. Na situação do método POST a única diferença é que os dados recebidos não são armazenados em memória pelo servidor. Na prática ao utilizar o método GET, o username e a password são armazenadas e possíveis de visualizar no url da página da conta do utilizador (figura 3), ao contrário do método POST que sendo a informação não armazenada não iria mostrar nenhum desses dados, mas sim são enviados no corpo do pedido HTTP.

Na linha 69 (app não segura) e 95 (app segura) as queries são desenvolvidas de forma diferente, tal que a query na app segura não pode ser manipulada, pois não permite a alteração do valor do user diretamente. O que acontece na app segura é que ao fazer a autenticação, a pessoa em questão só consegue entrar se o seu nome de utilizador corresponder corretamente à sua palavra-passe, sendo que a palavra-passe só é aceite se de facto for encontrada na base de dados esse valor que está atribuído ao user.

Na prática, se na aplicação segura tentarmos aplicar o que foi feito no site não seguro (user=bob e password=bob' or 1=1; --) o seguinte erro irá aparecer e não será possível por parte do invasor entrar na conta do utilizador bob:



The screenshot shows a login interface with a title "Login" in green. It contains two input fields: "Username" with the value "bob" and "Password" with masked characters. Below the fields is a green "Login" button. A red error message "Username and password don't match!" is displayed at the bottom.

Figura 6

Vulnerabilidade na barra de pesquisa na página de procura de produtos

Na página "products" está presente uma barra "search" que permite pesquisar produtos existentes na "Tech Store". A vulnerabilidade nesta situação estará presente nessa barra de pesquisa.

➤ Descrição de como aplicar a vulnerabilidade no site não seguro

Nesta barra de pesquisa está associada a seguinte query:

```
"SELECT ID_product,name,img_path FROM products WHERE name LIKE '%" + item + "%';"
```

Figura 7

Devido à query usada é possível pesquisar produtos por correspondência parcial de uma string ou caractere. Neste caso, a pesquisa é vulnerável pois é efetuada sem qualquer tipo de validação.

Deste modo, se injetarmos SQL simplesmente, por exemplo, no sentido de pesquisar um item (search=pen%';--) a query fica da seguinte forma:

```
" SELECT ID_product,name,img_path FROM products WHERE name LIKE '%pen%';--%'; "
```

No entanto, este tipo de vulnerabilidade pode ser usado de forma maliciosa. Tal que, neste caso, é possível aplicar enumeras injeções perigosas para o site.

Como demonstração, é possível ao colocar `search= pen%'; DROP TABLE products;--` apagar todos os dados armazenados na base de dados de products.

Na prática o que acontece é o seguinte:



Figura 8

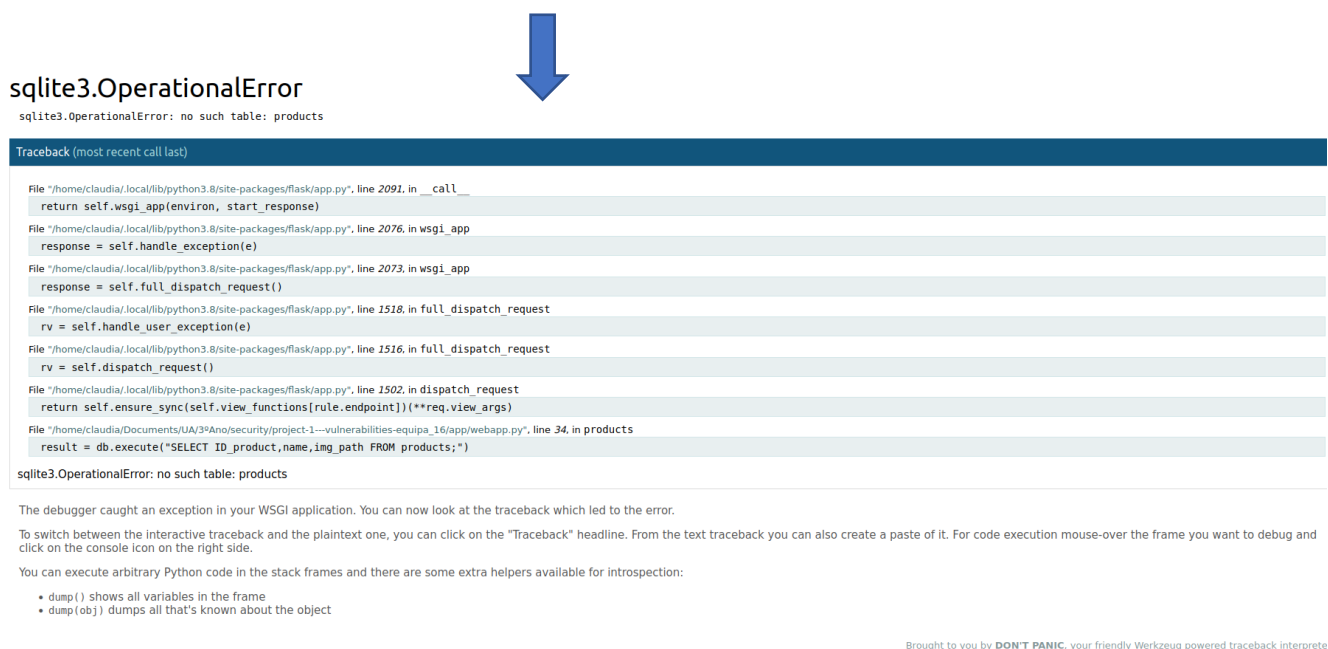


Figura 9

➤ Impacto da vulnerabilidade

Neste caso, a vulnerabilidade existente causa inúmeros perigos. Isto é, para cada SQL injection maliciosa realizada na barra de pesquisa poderá corresponder a um perigo diferente.

Como foi demonstrado na descrição da vulnerabilidade no nosso site, os riscos que se pode ter são, por exemplo, conseguir eliminar valores/colunas na base de dados da aplicação.

De forma a dar um exemplo real, imaginemos que o nosso site, Tech Store, sofre um ataque deste tipo e o que se irá suceder é que toda a informação disponível dos utilizadores será completamente eliminada. O que será bastante grave, pois com esta informação perde-se todas as contas e todo o dinheiro associado aos utilizadores.

➤ Comparação do site seguro com o site não seguro

Seguro

```
177 @app.route('/search', methods=['POST'])
178 def products_search():
179
180     item=request.form['search_name']
181
182     db = sql.connect("webDB.db")
183     result = db.execute("SELECT ID_product,name,img_path FROM products")
184     rows = result.fetchall()
185     db.close()
186
187     lista_produtos=[]#lista de tuplos
188
189     for row in rows:
190         name=row[1]
191         if item in name.lower():
192             id=row[0]
193             img=row[2].split("%")
194             lista_produtos.append([id,name,img[0]])
195
196     if 'username' in session:
197         return render_template('products.html',lista=lista_produtos,user=session["username"])
198
199     return render_template('products.html',lista=lista_produtos)
```

Figura 10

Não seguro

```
159 @app.route('/search', methods=['GET']) #! atenção ao erro do GET # Method Not Allowed
160 def products_search():
161
162     item=request.args.get('search_name', '')
163     query=item.split(";")
164
165     rows=[]
166     db = sql.connect("webDB.db")
167
168     item = query[0].replace("%", "")
169
170     result = db.execute("SELECT ID_product,name,img_path FROM products WHERE name LIKE '%" + item + "%'")
171     rows = result.fetchall()
172
173     if len(query)>1:
174         for i in range(1,len(query)):
175             db.execute(query[i])
176
177     db.close()
178
179     lista_produtos=[]#lista de tuplos
180
181     for row in rows:
182         name=row[1]
183         if item in name.lower(): #! DANGER sql possible injection
184             id=row[0]
185             img=row[2].split("%")
186             lista_produtos.append([id,name,img[0]])
187
188     if 'username' in session:
189         return render_template('products.html',lista=lista_produtos,user=session["username"])
190
191     return render_template('products.html',lista=lista_produtos)
192
193
```

Figura 11

No site seguro, o que acontece é que ao receber o valor do item (linha 180- fig. 10) é feito de seguida o acesso à base de dados (linhas 182-185-fig. 10) de forma a efetuar uma procura por todas as linhas do nome do produto ou produtos que se pretende. Nesta situação, não existe maneira de o atacante manipular a query utilizada (linha 183- fig. 10).

Por outro lado, no site não seguro, é possível manipular o valor final da query no sentido de obter algo malicioso (linha 11 – fig. 11).

É de notar que a utilização, neste caso, do método GET (linha 159 – fig. 11) contribui também para gerar uma maior vulnerabilidade na barra de pesquisa, ao contrário do que acontece com a utilização do método POST.

Vulnerabilidade no Suporte ao Cliente

Uma outra vulnerabilidade que está presente devido à possibilidade de SQL injection é na secção do "Client Support". Aqui, aplica-se também a manipulação da query final obtida, em função dos valores que inserimos nos campos de preenchimento por parte do cliente.

➤ Descrição de como aplicar a vulnerabilidade no site não seguro

A query criada é a seguinte:

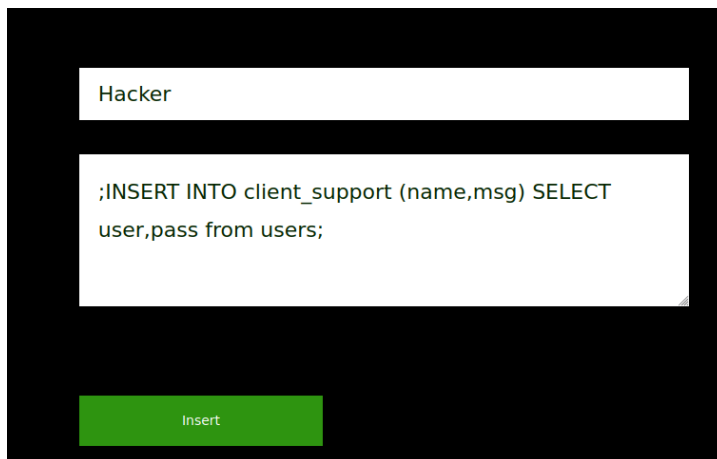
```
"INSERT INTO client_support VALUES (NULL,'" + nome + "', '" + query[0] + "');"

```

Figura 12

O invasor ao pretender, por exemplo, saber todas as palavras-passes dos utilizadores e inserir nos campos de suporte ao Cliente nome=hacker e mensagem=;INSERT INTO cliente_suport (name,msg) SELECT user,pass from users; consegue observar exatamente todos os utilizadores e as correspondentes palavras-passes, o que é extremamente grave.

Na prática o que acontece é o seguinte:



The image shows a web form with a dark background. It has two input fields. The first field contains the text "Hacker". The second field contains a SQL injection payload: ";INSERT INTO client_support (name,msg) SELECT user,pass from users;". Below the second field is a green button labeled "Insert".

Figura 13



Hacker

cunha

dfg123DSFF@

bob

password

voldemort





HarryPothaBriish

Figura 14

O que se consegue verificar que nas colunas user e pass são exatamente estes os valores mostrados no fórum de mensagens.

Table:

users

New Record

Delete

	ID	user	pass	money	email	avatar
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	cunha	dfg123DSFF@	144.13	oal@gmail.c...	cunha.jpg
2	2	bob	password	90.0	bghn@jdhs...	images.jpg
3	3	voldemort	HarryPotha...	87.0	fhjsdvfgdjhs...	Lord_Volde...

Figura 15

Outra injeção SQL que também se poderia aplicar seria `mensagem=;PRAGMA table_info(table_name);` Nesta situação é possível conseguir visualizar no fórum todas as colunas da tabela da base de dados. No entanto, não é possível demonstrar esta situação devido ao SQL Lite.

➤ Impacto da vulnerabilidade

Esta vulnerabilidade criada permite, num dos piores casos, ao invasor entrar na conta de qualquer utilizador, pois consegue ver a sua palavra-passe, sendo que esta não está encriptada, o que torna ainda mais fácil o ataque realizado.

Como foi referido na vulnerabilidade do login, este género de infrações causa bastantes consequências à vítima, sendo uma delas já referida no roubo de identidade.

➤ Comparação do site seguro com o site não seguro

Site seguro

```
311 @app.route('/client_support_update', methods=['POST']) #clique no form
312 def client_support2():
313
314     user=None
315     if 'username' in session:
316         user=session['username']
317
318     nome=request.form['name']
319     msg=request.form['msg']
320
321     db = sql.connect("webDB.db")
322     db.execute("INSERT INTO client_support VALUES (?,?);",(None,nome,msg))
323     db.commit()
324
325     result = db.execute("SELECT name,msg FROM client_support;")
326     msgs = result.fetchall()
327     db.close()
328
329     return render_template('client_support.html',user=user,msg_list=msgs)
```

Figura 16

Site não seguro

```
275 @app.route('/client_support', methods=['GET'])
276 def client_support():
277
278     user=None
279     if 'username' in session:
280         user=session['username']
281
282     nome=request.args.get('name', '')
283     msg=request.args.get('msg', '')
284     query=msg.split(";")
285
286     db = sql.connect("webDB.db")
287     if nome!="":#clique no form
288
289         result = db.execute("INSERT INTO client_support VALUES (NULL,'"+nome+"','"+query[0]+"');")
290
291         db.commit()
292         if len(query)>1:
293             for i in range(1,len(query)):
294                 db.execute(query[i])
295                 db.commit()
296
297     result = db.execute("SELECT name,msg FROM client_support;")
298     msgs = result.fetchall()
299     db.close()
300
301     return render_template('client_support.html',user=user,msg_list=msgs)
```

Figura 17

Na linha 275 (site não seguro) e 311(site seguro) a diferença presente está na utilização do método GET e POST, sendo que o método GET como já referido contribui para o site estar menos vulnerável ao contrário do POST que não armazena os dados em memória.

A grande diferença existente, que permite ao atacante injetar SQL no site, é na linha 289 (site não seguro) e 322 (site seguro). A query usada no site não seguro não tem qualquer tipo de validação e permite ser manipulada, ao contrário da query usada no site seguro que não permite qualquer tipo de manipulação.

CWE-22 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

"Many file operations are intended to take place within a restricted directory. By using special elements such as ".." and "/" separators, attackers can escape outside of the restricted location to access files or directories that are elsewhere on the system. One of the most common special elements is the "../" sequence, which in most modern operating systems is interpreted as the parent directory of the current location. This is referred to as relative path traversal. Path traversal also covers the use of absolute pathnames such as "/usr/local/bin", which may also be useful in accessing unexpected files. This is referred to as absolute path traversal."

"In many programming languages, the injection of a null byte (the 0 or NUL) may allow an attacker to truncate a generated filename to widen the scope of attack. For example, the software may add ".txt" to any pathname, thus limiting the attacker to text files, but a null injection may effectively remove this restriction."

EXPLICAÇÃO

Na vulnerabilidade CWE-22, utilizamos a tag img no HTML para mudar a imagem do perfil de um utilizador para outra, desde que essa imagem esteja dentro da pasta "static" e também para descobrir as pastas às quais o website acede.

WEBSITE VULNERÁVEL:

Exemplo de utilização:

Na página "account" e com a conta 'bob' ativa, ao inspecionarmos a imagem do avatar, verificamos que a tag de imagem vai buscar a mesma ao diretório "static/user_data/bob/images.jpg". A partir desta informação sabemos que existem três

```
> <div class="row text-start pt-3 justify-content-center"> ...  
</div> flex  
  
```

Figura 1: Código HTML do account



Figura 2: Página Account

pastas, sendo uma delas para elementos estáticos, outra para as pastas dos utilizadores e outra para armazenar a imagem do avatar respetivo.

Também podemos usar este método para descobrir outras pastas que possam existir no website ao navegar por diferentes páginas web. Por exemplo, ao ir para a página products, clicando num produto aleatório e inspecionando a sua imagem, descobrimos que dentro da página static temos mais uma página denominada img_db. Devido a esta pasta, podemos deduzir que o programador deste website usa a pasta img_db para armazenar as fotos relacionadas com os produtos do website.

```
<div class="client_img">  
    
</div>
```

Figura 3 Código HTML de um produto



Figura 4 Página HTML do produto

Juntando estas informações, podemos, por exemplo, mudar a imagem da pen para a imagem do utilizador, já que sabemos o caminho de ambas as imagens



Figura 5 Página HTML com CWE-22

WEBSITE SEGURO:

No site seguro ao inspecionar a imagem, continuamos a ver uma pasta static e outra pasta img para guardar a imagem do avatar.

```
</UIV> [LACA]
▼ <a href="/show_avatar">
  
```

Figura 6 Código HTML do account seguro

Product name

Pen USB INTEGRAL INFD32GBBLK

6.99€

Product Detail

The Integral Black is a fantastic value USB Fla
price, performance and durability.

Shop

Figura 7 Página HTML
do account seguro

No entanto, para mostrar a verdadeira imagem do utilizador, temos de aceder á página show_avatar para a conseguirmos ver. Nesta página não conseguimos ver o caminho para a imagem do utilizador, assegurando assim a privacidade do utilizador

```

<html>
  <head> </head>
  <body>
    
  </body>
</html>

```

Figura 8 Código HTML show_avatar seguro

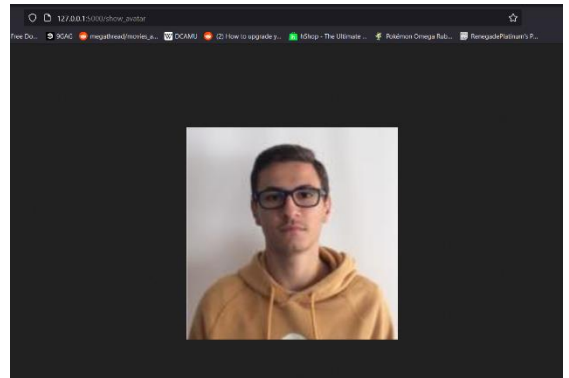


Figura 9 Página HTML do show_avatar

MUDANÇA DE CÓDIGO:

No código do website vulnerável, ao acedermos à base de dados, colocamos o caminho da imagem no html diretamente

```

@app.route("/account")

def account():
    if 'username' in session:
        db = sql.connect("webDB.db")
        result = db.execute("SELECT * FROM users WHERE user=?;",(session["username"],))
        data = result.fetchall()
        db.close()

        return
    render_template('account.html',user=data[0][1],money=data[0][3],email=data[0][4],avatar=os.path.join("static/user_data",session["username"],data[0][5]))
    return login_page()

```

No código seguro, vamos buscar todos os dados exceto a imagem. A imagem vai ser extraída através da função show_avatar presente no HTML. Esta função vê se o utilizador tem a sessão iniciada e depois vai á base de dados buscar a sua imagem.

```

@app.route("/show_avatar")
def show_avatar():

    if 'username' in session:
        db = sql.connect("webDB.db")
        result = db.execute("SELECT avatar FROM users WHERE
user=?;",(session['username'],))
        user_avatar = result.fetchall()
        db.close()

        path=safe_join(os.path.dirname(os.path.abspath(__file__)), "user_data", sess
ion['username'], user_avatar[0][0])

        return send_file(path, as_attachment=False)

    return index()

```

Alem disso, o atacante também não consegue aceder á pasta dos utilizadores, visto que esta agora está fora da página static.

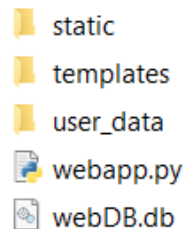


Figura 30 Organização das pastas no website seguro

- **Impacto da vulnerabilidade**

Com esta vulnerabilidade podemos aceder a imagens, violando a privacidade do utilizador que confiou na segurança da nossa aplicação “Tech Store”. Este ataque poderá gerar um acontecimento bastante grave, pois o atacante pode livremente colocar as imagens em locais não próprios.

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

EXPLICAÇÃO

Na vulnerabilidade CWE-79 é possível injetar código JavaScript numa página web em campos em que são aceites inputs do utilizador. Há 3 tipos de XSS: DOM-based, reflected e stored. XSS DOM-based é um pouco mais raro sendo que a injeção é feita do lado do servidor. No XSS reflected o servidor lê a informação tal e qual como é enviada pelo cliente e, como o input é refletido na página de resposta, o código JavaScript injetado é executado, mas apenas para o utilizador atual. XSS Stored acontece quando o utilizador comunica com a base de dados, sendo possível alojar código JS na base de dados que potencialmente será lido/executado por outros utilizadores.

IMPACTO DA VULNERABILIDADE:

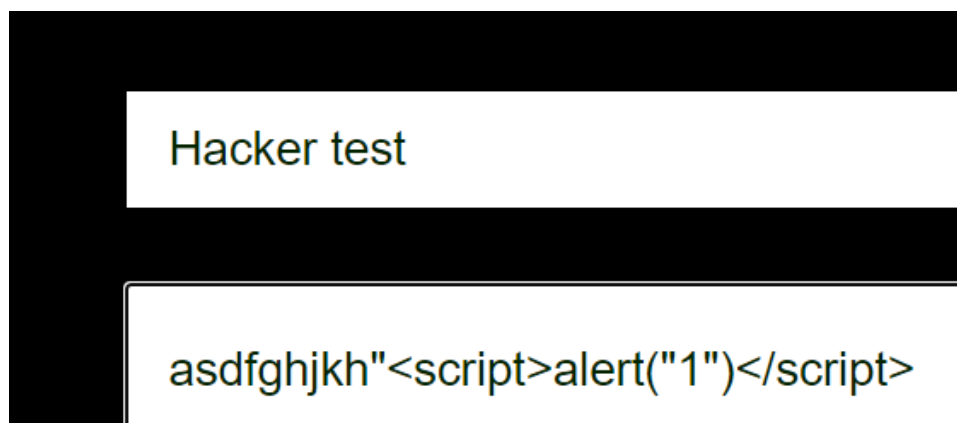
A vulnerabilidade XSS pode causar enormes complicações para as aplicações web e para os seus utilizadores. Contas de utilizador podem ser hackeadas, credenciais podem ser roubadas, dados sensíveis podem ser extraídos e ainda é possível a obtenção de acesso a computadores de utilizadores da aplicação através desta vulnerabilidade, tornando-a bastante perigosa.

WEBSITE VULNERÁVEL:

Exemplo de utilização:

Estando autenticado, na página “Client Support” é possível explorar esta vulnerabilidade de XSS nos parâmetros “Your name” e “Message”. Isto acontece porque o input do utilizador não é devidamente filtrado como podemos verificar nas imagens abaixo.

Input original



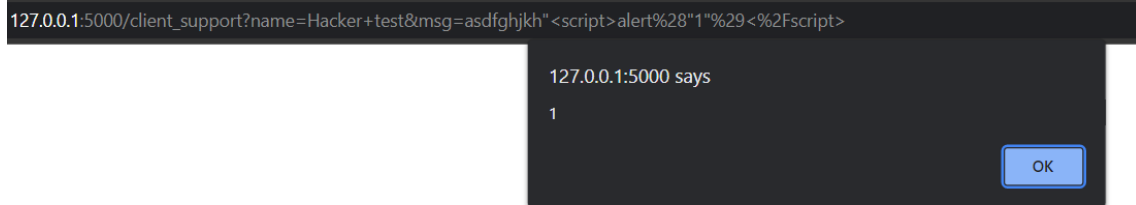
The diagram shows a black L-shaped frame representing a web form. Inside, there are two white rectangular input fields. The top field contains the text "Hacker test". The bottom field contains the malicious payload: `asdfghjkh"<script>alert("1")</script>`.

Código fonte após injeção do código JS

```
▼<a href="#" class="list-group-item list-group-item-ac  
mn align-items-start">  
  ▼<div class="d-flex w-100 justify-content-between">  
    <h5 class="mb-1">Hacker test</h5>  
  </div>  
  " asdfghjkh"  
  <script>alert("1")</script>  
</a>
```

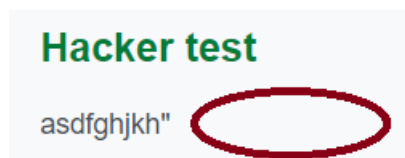
Podemos verificar que o browser não filtrou devidamente o input do utilizador, processando as tags script como código válido e legítimo, resultando na Alert box demonstrada abaixo.

Resultado após atualizar a página



Este é um caso de Stored XSS porque qualquer utilizador que aceda a esta página executará o payload (código JS malicioso).

Tags processadas como código legítimo e a mensagem não corresponde ao input completo



WEBSITE SEGURO:

Nesta versão do site já não é possível explorar a vulnerabilidade, como é visível na imagem seguinte. Sendo que as tags de script não estão a ser processadas como código, o input é tomado “como uma string inteira”.

**Tags processadas
correspondem ao input completo**

Hacker test

asdfghjkl"<script>alert(1)</script>

Desta vez o browser não assumiu o payload como código JavaScript, colocando o input dentro das tags apropriadas, nunca abrindo uma nova tag (script, neste caso).

```
▼<a href="#" class="list-group-item list-group-item-a  
t">  
  ▼<div class="d-flex w-100 justify-content-between">  
    <h5 class="mb-1">Hacker test</h5>  
  </div>  
  " asdfghjkl"<script>alert("1")</script> "  
</a>
```

MUDANÇA DE CÓDIGO:

No caso inseguro, o input do utilizador é concatenado na query, ignorando qualquer tipo de caracter especial, como se pode ver na linha 289.

```
276 @app.route('/client_support', methods=['GET'])
277 def client_support():
278     user=None
279     if 'username' in session:
280         user=session['username']
281
282     nome=request.args.get('name','')
283     msg=request.args.get('msg','')
284     query=msg.split(";")
285
286     db = sql.connect("webDB.db")
287     if nome!="":#clizou no form
288         result = db.execute("INSERT INTO client_support VALUES (NULL, '"+nome+"', '"+query[0]+'');")
289         db.commit()
290     if len(query)>1:
291         for i in range(1,len(query)):
292             db.execute(query[i])
293             db.commit()
294
295     result = db.execute("SELECT name,msg FROM client_support;")
296     msgs = result.fetchall()
297     db.close()
298
299     return render_template('client_support.html',user=user,msg_list=msgs)
```

No caso do site seguro estamos perante uma prepared statement, sendo que os inputs do utilizador são inseridos na query como uma string “inteira”, ignorando o contexto do código.

```
311 @app.route('/client_support_update', methods=['POST']) #clizou no form
312 def client_support2():
313     user=None
314     if 'username' in session:
315         user=session['username']
316
317     nome=request.form['name']
318     msg=request.form['msg']
319
320     db = sql.connect("webDB.db")
321     db.execute("INSERT INTO client_support VALUES (?, ?, ?);", (None, nome, msg))
322     db.commit()
323
324     result = db.execute("SELECT name,msg FROM client_support;")
325     msgs = result.fetchall()
326     db.close()
327
328     return render_template('client_support.html',user=user,msg_list=msgs)
```

Para além do ponto acima referido existe ainda uma grande diferença crucial entre a app insegura e a versão segura. Como é possível observar na imagen seguinte, a versão insegura apresenta o “safe”, o que faz com que não sejam traduzidos símbolos “perigosos” em entidades HTML e consequentemente a app confie exclusivamente no input do utilizador.

```
119 <div class="col-md-6" style="padding-right: 100px;">
120   <div class="map-responsive">
121     <div class="list-group float-right" style="width: 650px">
122       {% for msg in msg_list:%}
123       <a href="#" class="list-group-item list-group-item-action flex-column align-items-start">
124         <div class="d-flex w-100 justify-content-between">
125           <h5 class="mb-1">{{ msg[0]|safe }}</h5>
126         </div>
127         {{ msg[1]|safe }}
128       </a>
129       {% endfor %}
130     </div>
131   </div>
132 </div>
```

Na versão segura o mesmo já não acontece, estando a app mais protegida contra ataques XSS através do problema acima referido.

```
119 <div class="col-md-6" style="padding-right: 100px;">
120   <div class="map-responsive">
121     <div class="list-group float-right" style="width: 800px">
122       {% for msg in msg_list:%}
123       <a href="#" class="list-group-item list-group-item-action flex-column align-items-start">
124         <div class="d-flex w-100 justify-content-between">
125           <h5 class="mb-1">{{ msg[0] }}</h5>
126         </div>
127         {{ msg[1] }}
128       </a>
129       {% endfor %}
130     </div>
131   </div>
132 </div>
```

Conclusão:

No OWASP Top10 2021, XSS está incluído no A03-Injection em conjunto com outras vulnerabilidades semelhantes tais como SQLi ou XML injection. Posto isto, Cross-Site Scripting é ainda um dos maiores problemas de segurança das aplicações web atualmente.

OWASP Top10 A3: Injection - https://owasp.org/Top10/A03_2021-Injection/

CWE-522: Insufficiently Protected Credentials

EXPLICAÇÃO

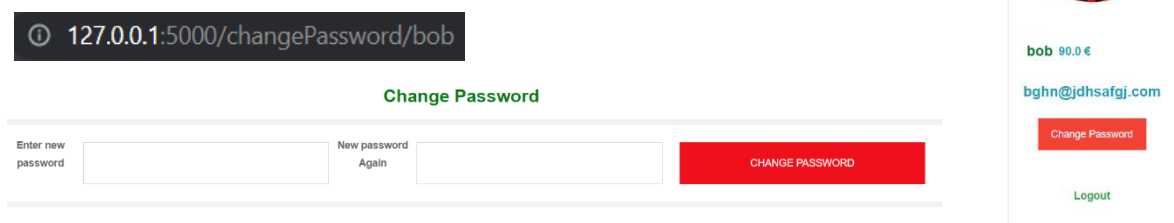
Na vulnerabilidade CWE-522, a aplicação em questão utiliza credenciais de acesso para a gestão de contas de utilizadores, no entanto a implementação de um dos métodos está feita de forma insegura. Neste caso em específico, o método em questão é o que permite a alteração da password noutra conta que não a que tem a sessão iniciada.

IMPACTO DA VULNERABILIDADE:

Esta vulnerabilidade pode permitir ao atacante ganhar acesso a contas dos utilizadores da aplicação web e a dados sensíveis utilizados pelos mesmos, tornando-se assim num problema importante de combater.

WEBSITE VULNERÁVEL:

O website permite a alteração da password de um outro utilizador sem a sessão iniciada. Iniciando sessão como utilizador Bob e acedendo à página de alteração de password, é apresentada a seguinte página:



The screenshot displays a web interface for changing a password. At the top, a terminal-style address bar shows the URL `127.0.0.1:5000/changePassword/bob`. Below this is a green header with the text 'Change Password'. The main form area contains two input fields: 'Enter new password' and 'New password Again', followed by a red 'CHANGE PASSWORD' button. To the right, a sidebar titled 'Your Profile' shows a cartoon image of Bob the Dinosaur, the name 'bob', a balance of '90.0 €', the email 'bghn@jdhsafgj.com', a 'Change Password' button, and a 'Logout' link.

No entanto, se for alterado o URL de `127.0.0.1:5000/changePassword/bob` para `127.0.0.1:5000/changePassword/cunha`, é possível alterar a password do utilizador Cunha ao invés do utilizador com sessão ativa. Inicialmente a base de dados encontrava-se da seguinte forma:

```
sqlite> select * FROM users;
1|cunha|dfg123DSFF@144.13|oal@gmail.com|cunha.jpg
2|bob|password|90.0|bghn@jdhsafgj.com|images.jpg
3|voldemort|HarryPothaBriish|87.0|fhjsdvfgdjhsf@ua.pt|Lord_Voldemort.jpg
```

Após explorar a vulnerabilidade descrita acima da seguinte maneira:

127.0.0.1:5000/changePassword/cunha

Change Password

Enter new password: abcABC_123

New password Again: abcABC_123

CHANGE PASSWORD

(Para efeitos demonstrativos a password encontra-se em clear text, o que não reflete o normal funcionamento do site)

A base de dados atualiza, alterando a password do utilizador cunha para a inserida acima, sem o mesmo nunca se ter autenticado com as suas credenciais originais, como é possível observar na imagem seguinte.

```
sqlite> select * FROM users;  
1|cunha|abcABC_123|144.13|oal@gmail.com|cunha.jpg  
2|bob|password|90.0|bghn@jdhsafgj.com|images.jpg  
3|voldemort|HarryPothaBriish|87.0|fhjsdvfgdjhsf@ua.pt|Lord_Voldemort.jpg
```

WEBSITE SEGURO:

No site seguro, ao acedermos à página de mudança de password, o nome do utilizador já não se encontra presente no URL, e o atual utilizador ativo apenas é determinado através de um cookie forte, ou seja, qualquer tentativa de adicionar um nome de utilizador ao URL como era possível no site inseguro, resulta numa resposta 404 Not Found.

127.0.0.1:5000/changePassword/bob

Not Found

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

Na nova versão (segura) também foi adicionado um novo campo que exige a password atual do utilizador antes da mesma ser alterada. Desta forma é possível evitar que um potencial atacante com uma sessão isolada ativa, seja capaz de alterar a password sem conhecimento da atual

127.0.0.1:5000/changePassword

Change Password

Current Password

New Password

Confirm new password

CHANGE PASSWORD

MUDANÇA DE CÓDIGO:

No código do website vulnerável, é possível verificar que o username devolvido pela função é o mesmo que se encontra no URL, ou seja, a password alterada é a correspondente ao username que é passado à função.

```
@app.route('/changePassword/<username>')
def changePassword(*args,username):
    if 'username' in session:
        if len(args)>0:
            return
    render_template('changePassword.html',alert=args[0],user=username)
    return render_template('changePassword.html',user=username)

return index()
```



```

254 @app.route('/password_changed/<username>', methods=['GET'])
255 def password_changed(username):
256     if 'username' in session:
257         pass1=request.args.get('pass1', '')
258         pass2=request.args.get('pass2', '')
259
260         if pass1==pass2:
261             if verify_password(pass1):
262                 db = sql.connect("webDB.db")
263                 db.execute("UPDATE users SET pass='"+pass1+"' WHERE user='"+username+"'")
264                 db.commit()
265                 db.close()
266                 return account(1)
267             return changePassword(2)
268         else:
269             return changePassword(1)
270
271
272     return index()

```

No código do website seguro o mesmo já não acontece. O form que exige a submissão da password atual está descrita no ficheiro HTML, que apesar de ter o mesmo nome, não é o mesmo ficheiro.

```

@app.route('/changePassword')
def changePassword(*args):
    if 'username' in session:
        if len(args)>0:
            return render_template('changePassword.html', alert=args[0])
        return render_template('changePassword.html')

    return index()

```

Para além do ponto acima referido, foi ainda adicionada encriptação à password, fortalecendo assim ainda mais a segurança da aplicação web.

```

262 @app.route('/password_changed', methods=['POST'])
263 def password_changed():
264     if 'username' in session:
265
266         currentPassInput=request.form['currentPass']
267
268         result = hashlib.sha256(currentPassInput.encode())
269         encrypted_currentPassInput=result.hexdigest()
270
271         db = sql.connect("webDB.db")
272         result = db.execute("SELECT pass FROM users WHERE user=?;",(session['username'],))
273         currentPass = result.fetchall()
274         db.close()
275
276         if currentPass[0][0] == encrypted_currentPassInput:
277             pass1=request.form['newPass1']
278             pass2=request.form['newPass2']
279             if pass1==pass2:
280                 if verify_password(pass1):
281                     result = hashlib.sha256(pass1.encode())
282                     encrypted_pass=result.hexdigest()
283
284                     db = sql.connect("webDB.db")
285                     db.execute("UPDATE users SET pass=? WHERE user=?;",(encrypted_pass,session['username']))
286                     db.commit()
287                     db.close()
288                     return account()
289                 return changePassword(2)
290             else:
291                 return changePassword(1)
292         else:
293             return changePassword(3)
294
295     return index()

```

CONCLUSÃO:

No OWASP Top10 2021, esta vulnerabilidade relacionada com identificação está incluída no A07:2021-Identification and Authentication Failures, em conjunto com outras vulnerabilidades relacionadas com identificação e autenticação.

Extras

Extra 1

Na versão insegura, não é necessário um password considerado forte, enquanto na versão segura isso é um requisito (a password deve conter letras minúsculas e maiúsculas, números, caracteres especiais e no mínimo um tamanho de 11 caracteres)

Very weak password!

Outra das grandes diferenças entre a aplicação segura e a não segura é o facto de a segura encriptar as passwords, logo se por alguma razão a base de dados for comprometida os atacantes não terão acesso direto às passwords. Apenas conseguiriam entrar através do método tentativa e erro (brute force), o que devido às características obrigatórias da password no site seguro deveria de demorar aproximadamente 400 anos de acordo com <https://howsecureismypassword.net/> o que é muito mais que a esperança de vida de um utilizador.

	ID	user	pass
	Filter	Filter	Filter
1	1	cunha	dfg123DSFF@
2	2	bob	password
3	3	voldemort	HarryPothaBriish

	ID	user	pass
	Filter	Filter	Filter
1	1	cunha	09713825d39878ca927cb0caa22a5bc4a63cc6667b7a96b3cecbf70ae0864339
2	2	bob	5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
3	3	voldemort	9688381e0b41a1d7b217225c6d5084c48fdd21112deb954dedb534ec08de28e6

Outra das diferenças é o change password, onde na app_sec o utilizador é obrigado a introduzir a password antiga para poder alterá-la, ao contrário da app onde o user só introduz a password nova e a repete como confirmação.

Change Password

Enter new password	<input type="password"/>	New password Again	<input type="password"/>	<input type="button" value="CHANGE PASSWORD"/>
--------------------	--------------------------	--------------------	--------------------------	--

Change Password

Current Password
<input type="password"/>
New Password
<input type="password"/>
Confirm new password
<input type="password"/>
<input type="button" value="CHANGE PASSWORD"/>

Extra 2

Há uma ligeira falha no sign_up da aplicação não segura que se localiza mais especificamente no campo de colocação do avatar do utilizador, onde quando é solicitado ao user o seu avatar, se sairmos da página irá aparecer o novo utilizador que está em sessão iniciada. No entanto, ao clicar no “account”, como não se encontra localizada lá a imagem irá ocorrer um erro que dará informações do nosso código ao atacante.

Nesta situação, pode-se concluir que com estes erros é possível fornecer informação critica aos atacantes.

sqlite3.OperationalError

sqlite3.OperationalError: no such table: products

Traceback (most recent call last)

File "/home/claudia/.local/lib/python3.8/site-packages/flask/app.py", line 2091, in __call__
return self.wsgi_app(environ, start_response)

File "/home/claudia/.local/lib/python3.8/site-packages/flask/app.py", line 2076, in wsgi_app
response = self.handle_exception(e)

File "/home/claudia/.local/lib/python3.8/site-packages/flask/app.py", line 2073, in wsgi_app
response = self.full_dispatch_request()

File "/home/claudia/.local/lib/python3.8/site-packages/flask/app.py", line 1518, in full_dispatch_request
rv = self.handle_user_exception(e)

File "/home/claudia/.local/lib/python3.8/site-packages/flask/app.py", line 1516, in full_dispatch_request
rv = self.dispatch_request()

File "/home/claudia/.local/lib/python3.8/site-packages/flask/app.py", line 1502, in dispatch_request
return self.ensure_sync(self.view_functions[rule.endpoint])(**req.view_args)

File "/home/claudia/Documents/UA/3ºAno/security/project-1---vulnerabilities-equipa_16/app/webapp.py", line 34, in products
result = db.execute("SELECT ID_product,name,img_path FROM products;")

sqlite3.OperationalError: no such table: products

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.

To switch between the interactive traceback and the plaintext one, you can click on the “Traceback” headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side.

You can execute arbitrary Python code in the stack frames and there are some extra helpers available for introspection:

- dump() shows all variables in the frame
- dump(obj) dumps all that's known about the object

Brought to you by **DON'T PANIC**, your friendly Werkzeug powered traceback interpreter

Enquanto que na versão segura:

Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.