

TP cybersécurité & API

Analyse et correction d'une application vulnérable

1. Faille dans la fonction recherche utilisateur

Où était le problème ?

En utilisant une injection SQL, lors d'une requête « GET/?search= » en insérant « ' OR 1=1 -- » dans le champ de recherche utilisateur, il est possible de récupérer tous les utilisateurs. Cela peut entraîner un accès à des données sensibles ou non autorisées, ou un contournement de l'authentification, ce qui peut provoquer des fuites de données.

Qu'avez-vous modifié ?

Afin d'éviter cette attaque, j'ai modifié le code après l'analyse des aides fourni dans le TP :

#code original :

```
cur.execute(f"SELECT username FROM users WHERE username LIKE '%{query}%'")
```

#modification :

```
params = (query,)
cur.execute("SELECT username FROM users WHERE username LIKE ?", params)
```

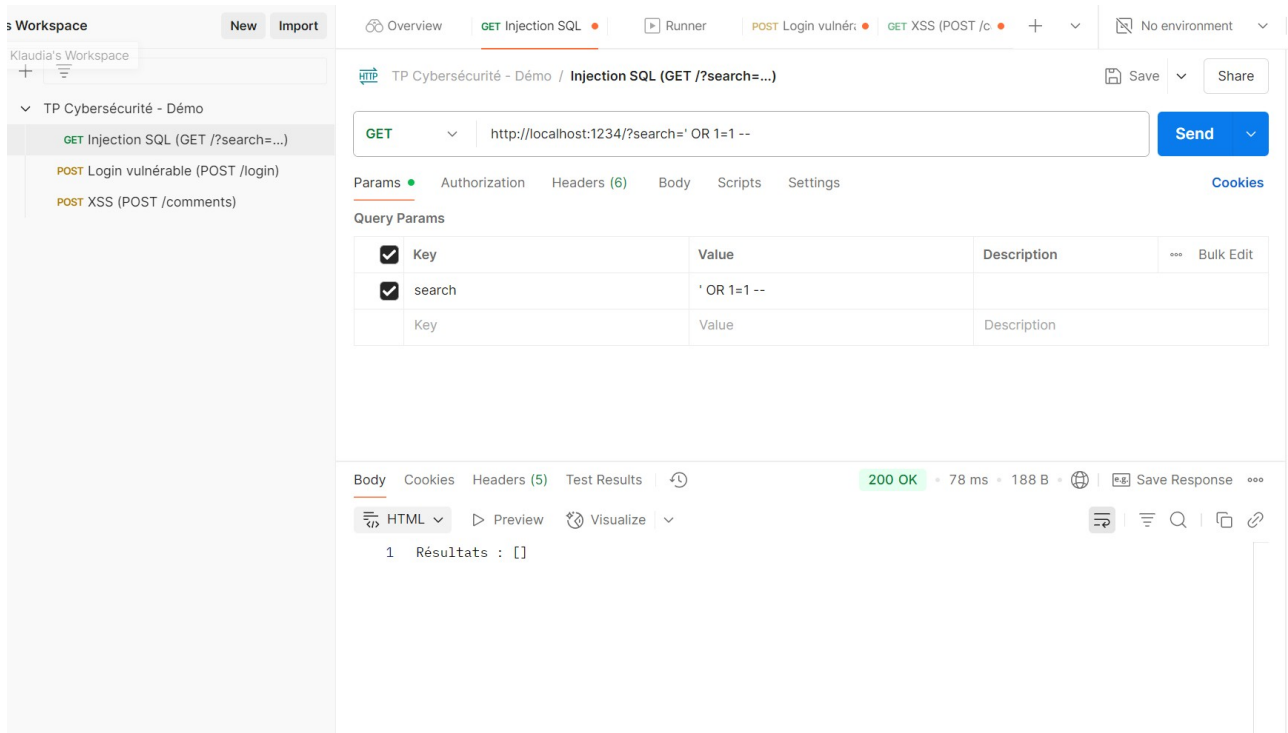
- Avec la variable « params », on crée un tuple qui contient « query »
- Afin que Python reconnaisse un tuple à un seul élément, il faut mettre une virgule après le « query »
- après le « LIKE » il faut mettre un « ? » pour empêcher les attaques par injection SQL, lequel est remplacé par « params » dans la requête
- on fait une requête pendant laquelle on filtre le résultat selon «query» grâce à « LIKE » tout en empêchant les attaques par injection SQL
- **si « query= 'Bob' » ⇒ « params = ('Bob',) » :**
qui signifie qu'on cherche que Bob
- **si « params = ('%' + Bob + '%') » ⇒ « '%Bob %' »**
qui signifie qu'on cherche Bob dans n'importe quel format.. p.ex. : Bobby, JohnBob...

Comment avez-vous testé la correction ?

Sur le site j'ai essayé faire une recherche avec « ' OR 1=1 -- » et il n'a pas retourner tous les utilisateurs, juste j'ai eu une réponse : Résultats : []

En faisant des requêtes *avec Postman*, j'ai pu également constater que les injections SQL ne fonctionnent plus.

Voici le capture d'écran :



2. Faille dans le fonction connexion

Où était le problème ?

Dans la base de données, les mots de passe sont visibles en clair, ce qui permet à un attaquant de voler les comptes des utilisateurs. Dans le pire des cas, il pourrait accéder et prendre le contrôle du compte administrateur, causant ainsi des dommages à l'entreprise

Qu'avez-vous modifié ?

#code original :

```
# △ Authentification avec mot de passe en clair
cur.execute("SELECT * FROM users WHERE username=? AND password=?", (username, password))
user = cur.fetchone()
conn.close()

if user:
    return f"Bienvenue {username} !"
error = "Échec de connexion"
```

#modification :

```
# Récupérer le mot de passe hashé depuis la base de données
cur.execute("SELECT password FROM users WHERE username=?", (username,))
user = cur.fetchone()
conn.close()

if user:
    stored_hashed_password = user[0]

    #Vérifier si le mot de passe fourni correspond au mot de passe hashé stocké
    if user and bcrypt.checkpw(password, stored_hashed_password.encode("utf-8")):
        return f"Bienvenue {username} !"
error = "Échec de connexion"
```

- J'ai changé les passwords claires des utilisateurs dans la base de donnée, en mettant passwords hashés utilisant la hachage en ligne <https://bcrypt-generator.com/>
- ```
password = request.form["password"].encode("utf-8")
```

 afin que je puisse encoder en bytes le mot de passe entré par l'utilisateur en le récupérant, j'ai ajouté « .encode('utf-8') »
- avec une requête, je récupère le mot de passe hashé associé au «username» de la base de donnée :
- si l'utilisateur existe, je récupère son mot de passe hashé et puis
- je compare le mot de passe entré par l'utilisateur et le mot de passe hashé qui se trouve dans la base de donnée

### ***SUPPORT :***

```
import bcrypt

example password
password = 'passwordabc'

converting password to array of bytes
bytes = password.encode('utf-8')

generating the salt
salt = bcrypt.gensalt()

Hashing the password
hash = bcrypt.hashpw(bytes, salt)

#mettre dans la BDD avec une requête SQL, l'utilisateur et son MDP hashé :

INSERT INTO users (username, password) VALUES ("admin",
"$2a$12$yM0kxpydj80PWlOgwbW6eCKPHkodZPuESUQ7EEbCr5MOo6lkmZpu");

Taking user entered password
userPassword = 'password000'

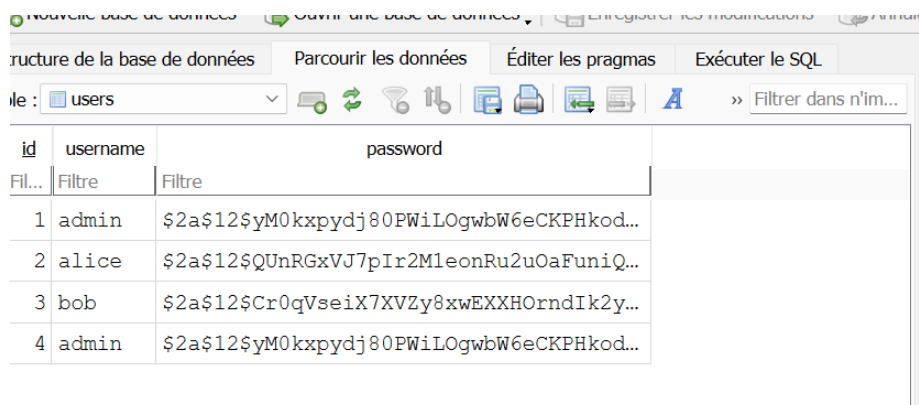
encoding user password
userBytes = userPassword.encode('utf-8')

checking password
result = bcrypt.checkpw(userBytes, hash)

print(result)
```

### ***Comment avez-vous testé la correction ?***

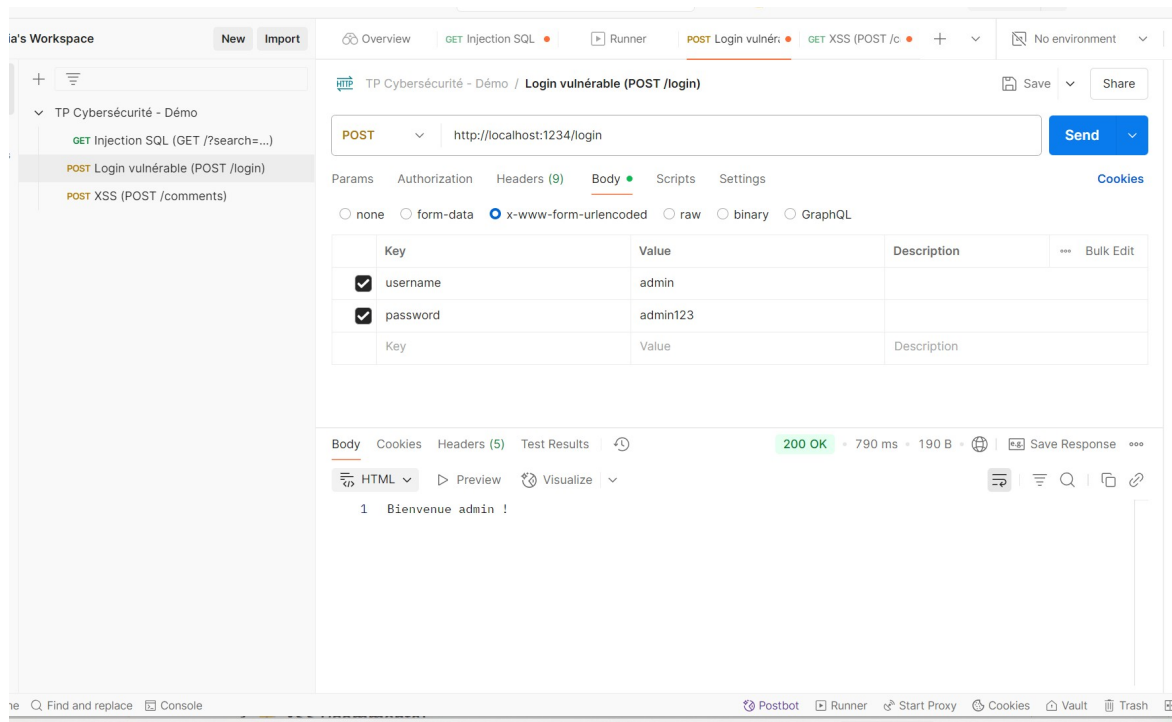
En ouvrant DB Browser (SQLite), on peut constater qu'aucun mot de passe n'est visible en clair :



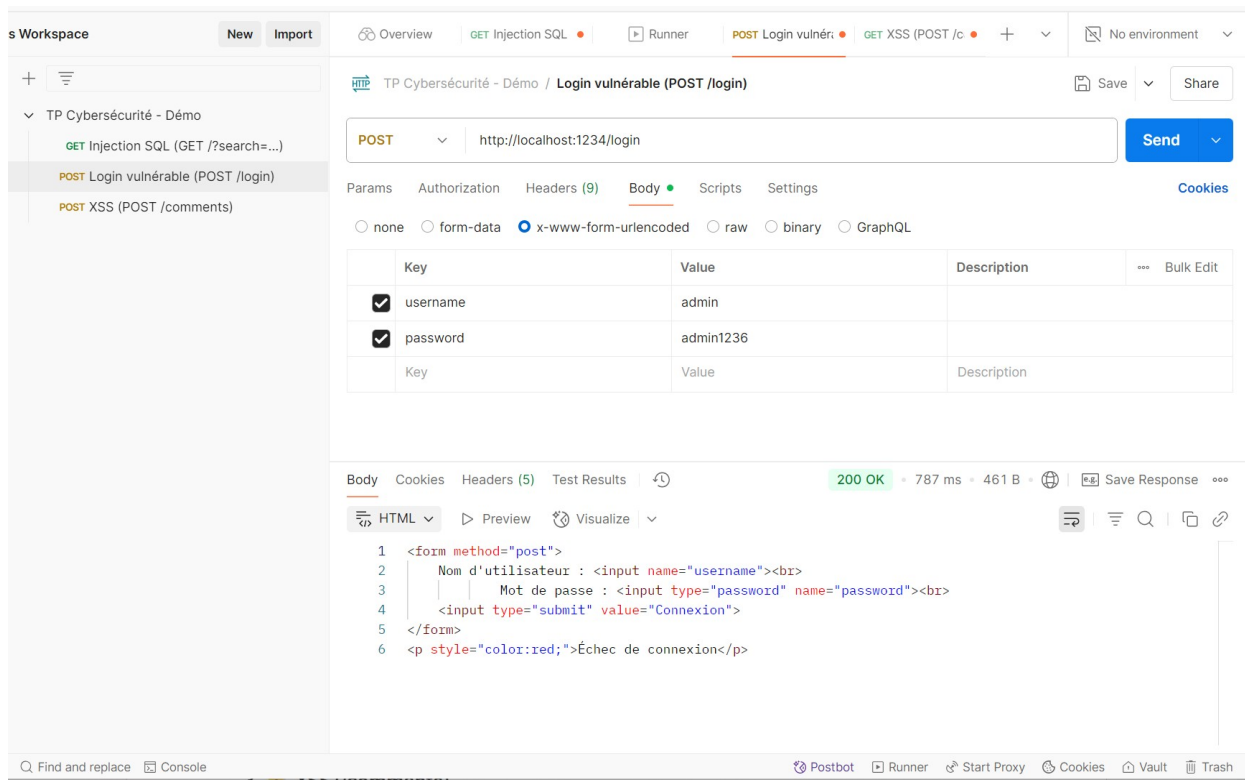
The screenshot shows the DB Browser for SQLite interface. The 'users' table is selected, and its data is displayed in a table view. The table has three columns: 'id', 'username', and 'password'. The 'password' column contains hashed values, demonstrating that the passwords are not stored in plain text.

| id | username | password                                   |
|----|----------|--------------------------------------------|
| 1  | admin    | \$2a\$12\$yM0kxpydj80PWlOgwbW6eCKPHkod...  |
| 2  | alice    | \$2a\$12\$QUnRGxVJ7pIr2M1eonRu2uOaFuniQ... |
| 3  | bob      | \$2a\$12\$Cr0qVseiX7XVZy8xwEXXHOrndIk2y... |
| 4  | admin    | \$2a\$12\$yM0kxpydj80PWlOgwbW6eCKPHkod...  |

En essayant de se connecter en tant qu'utilisateur, par exemple Bob ou admin, et en entrant son mot de passe, un message de bienvenue apparaît plutôt qu'un message d'erreur, lequel peut également être observé en effectuant une requête *avec Postman* :



Voici le résultat de la requête si j'essaie de se connecter avec un mauvais mot de passe :



### 3. Faille dans le fonction de commentaires

#### Où était le problème ?

En raison d'une faille XSS, le champ de commentaire accepte tout code HTML ou JavaScript qui peut être exécuté dans le navigateur. Par conséquent, un attaquant peut faire apparaître une boîte de dialogue, rediriger la page ou voler des données.

#### Qu'avez-vous modifié ?

#code original :

```
if request.method == "POST":

 # ⚠ Pas de filtrage du contenu (faille XSS)
 cur.execute("INSERT INTO comments (username, content) VALUES (?, ?)",
 (request.form["username"], request.form["content"]))
 conn.commit()
cur.execute("SELECT username, content FROM comments")
all_comments = cur.fetchall()
conn.close()

#escape() ==> afin de ne pas être interprété < et > comme code HTML
html = "<h2>Commentaires :</h2>"
for user, content in all_comments:
 html += f"{user} : {content}"
html += "<hr>"
html += ""
 <form method="post">
 Nom : <input name="username">

 Commentaire : <textarea name="content"></textarea>

 <input type="submit">
 </form>
return render_template_string(html)
```

#modification :

```
if request.method == "POST":
 username = escape(request.args.get("username")) # Protection contre XSS
 content = escape(request.args.get("content")) # Protection contre XSS

 cur.execute("INSERT INTO comments (username, content) VALUES (?, ?)",
 (username, content))
 conn.commit()
cur.execute("SELECT username, content FROM comments")
all_comments = cur.fetchall()
conn.close()

#escape() ==> afin de ne pas être interprété < et > comme code HTML
html = "<h2>Commentaires :</h2>"
for user, content in all_comments:
 html += f"{escape(user)} : {escape(content)}"
html += "<hr>"
html += """
 <form method="post">
 Nom : <input name="username">

 Commentaire : <textarea name="content"></textarea>

 <input type="submit">
 </form>
"""
return render_template_string(html)
```

- j'ai utilisé escape() afin d'empêcher l'exécution du script dans les variables «username» et «content» pendant l'insertion de l'utilisateur
- Pour sécuriser l'affichage des commentaires, j'ai ajouté escape(), qui permet que <script> soit afficher en texte sans l'exécuter :

```
html += f"{escape(user)} : {escape(content)}"
```

## Comment avez-vous testé la correction ?

**En accédant à la page** (<http://127.0.0.1:1234/comments>), j'ai remarqué que les boîtes de dialogue n'apparaissent plus ; les codes `<script>` s'affichent sous forme de texte dans la page.

En faisant des requêtes avec **Postman**, on peut voir que le code « `<script>alert('XSS')</script>` » s'affiche en text : « `&lt;script&gt;alert(&#x27;XSS&#x27;)&lt;/script&gt;` » :

The screenshot shows the Postman interface with a GET request to `http://localhost:1234/comments`. The request body is set to `<script>alert('XSS')</script>`. The response is a 200 OK status with a response time of 16 ms and a size of 1.16 KB. The response body is displayed in HTML format, showing the rendered output of the script tag as text.

| Key      | Value                         | Description |
|----------|-------------------------------|-------------|
| username | Alice                         |             |
| content  | <script>alert('XSS')</script> |             |

```
1 <h2>Commentaires :</h2>
2
3 alice : Hi everyone!
4 bob : Hello everybody!
5 attacker : <script>alert('XSS')</script>
6 attacker : <script>alert('XSS')</script>
7 attacker : <script>alert('XSS')</script>
8 alice : <script>alert('XSS')</script>
9 attacker : <script>alert('XSS')</script>
10
```