Rapport TP Programmation sécurisée

Formation : IRA5
Étudiante : Camille TOURET
Date : 28/01/2026

Lien du repository Github :
https://github.com/camtrt0912/ProgrammationSecurisee/tree/main

Table des matières

Contexte

Ce projet a pour objectif de vous confronter aux problématiques réelles de la sécurité applicative. Vous devrez concevoir, développer et auditer une application Web en appliquant les bonnes pratiques de programmation sécurisée.

Concernant l'organisation, je réalise l'entièreté du projet, comprenant la partie développeur et analyste sécurité.

Méthodologie de Test

Préparation

- Déploiement de l'application en environnement local sécurisé.
- Vérification que toutes les dépendances sont à jour.
- Création de comptes utilisateurs test (User et Admin) pour simuler différents scénarios.

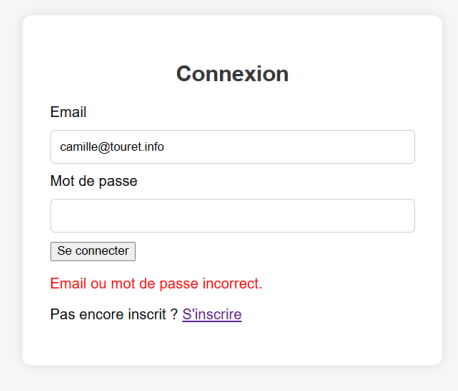
Tests Réalisés

Tests SQL Injection

Objectif : Vérifier que les champs de formulaire ne permettent pas l'injection de code SQL.

Lorsque l'utilisateur saisit une adresse email valide mais un mot de passe incorrect, la validation du formulaire est acceptée par Flask-WTF. Le traitement côté serveur est donc exécuté : la requête SQL est envoyée à la base de données, puis l'authentification échoue proprement. Dans ce cas, un message d'erreur explicite est affiché à l'utilisateur indiquant que les identifiants sont incorrects.

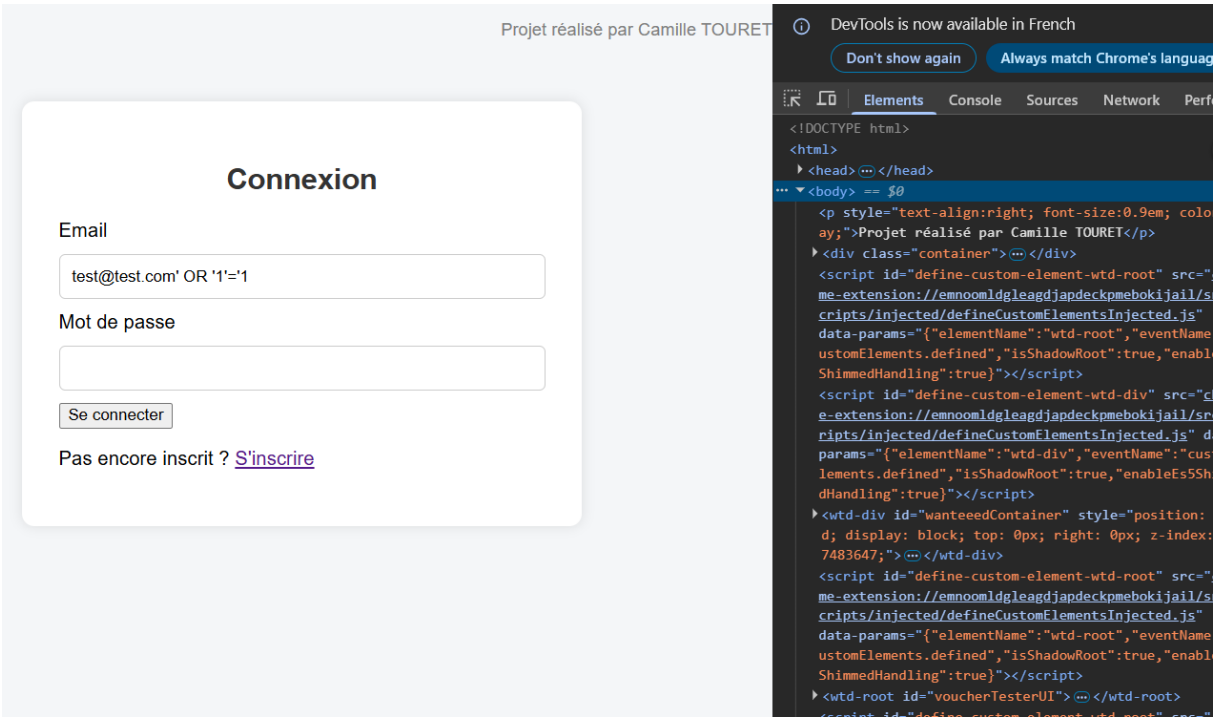
Ce comportement permet d'informer l'utilisateur sans divulguer d'informations sensibles sur l'existence des comptes.



Lorsqu'une tentative d'injection SQL est saisie dans le champ email (ex : test@test.com' OR '1'='1), la valeur ne respecte pas le format attendu d'une adresse email. Le validateur Email() de Flask-WTF bloque la soumission du formulaire avant même que le code applicatif ne soit exécuté.

La requête SQL n'est donc jamais envoyée à la base de données et aucun message d'erreur applicatif n'est affiché.

Ce comportement explique l'absence de message visible lors des tentatives d'injection : la requête est rejetée au niveau de la validation serveur.



Cette différence de comportement illustre la présence de plusieurs niveaux de protection :

- Validation côté client (HTML5)
- Validation côté serveur via Flask-WTF
- Requêtes SQL paramétrées

Ces mécanismes empêchent les entrées malveillantes d'atteindre la couche base de données et réduisent la surface d'attaque de l'application.

Preuve dans le script app.py :

Dans *@app.route("/login", methods=["GET", "POST"])*

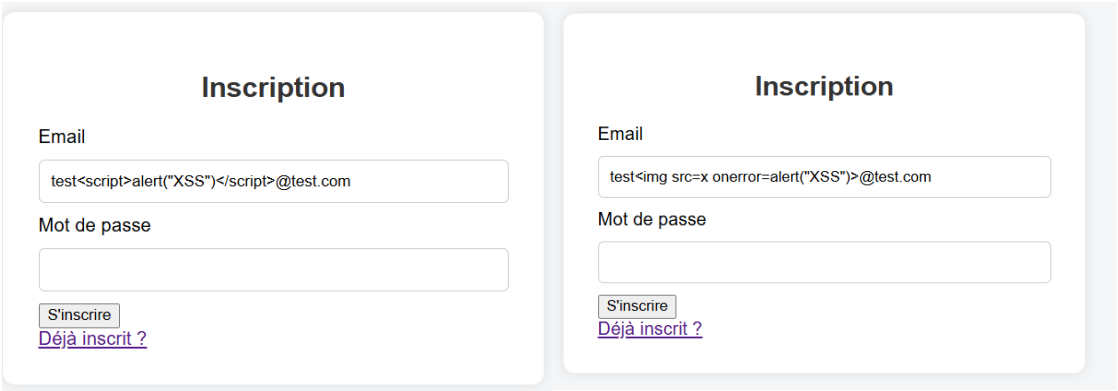
```
cursor = conn.cursor()
cursor.execute("SELECT * FROM users WHERE email = ?", (email,))
user = cursor.fetchone()
```

L'utilisation de requêtes paramétrées empêche l'interprétation du contenu utilisateur comme du SQL.

Tests XSS (Cross-Site Scripting)

Objectif : Vérifier que l'application échappe ou filtre les entrées utilisateurs pour éviter l'exécution de scripts malveillants.

Les tentatives d'injection XSS sont bloquées dès la validation serveur du champ email via le validateur Flask-WTF. Les données malformées ne sont jamais traitées par l'application, ce qui réduit la surface d'attaque.



Contrairement à une tentative de connexion avec un email valide où un message d'erreur est affiché, les tentatives d'injection XSS ne génèrent aucun message visible. Cela s'explique par la validation serveur du champ email via Flask-WTF qui rejette toute valeur non conforme avant l'exécution du code applicatif. Cette validation constitue une première barrière de sécurité empêchant toute injection de contenu malveillant.

De plus, les données sont échappées via la fonction escape() et le moteur de template Jinja2, garantissant l'absence d'exécution de code JavaScript.

Preuve dans le script app.py :

Dans *@app.route("/register", methods=["GET", "POST"])*

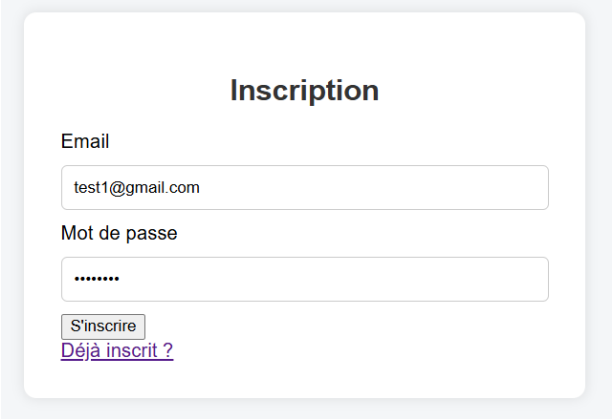
```
# Protection XSS
email = escape(form.email.data)
password = form.password.data
```

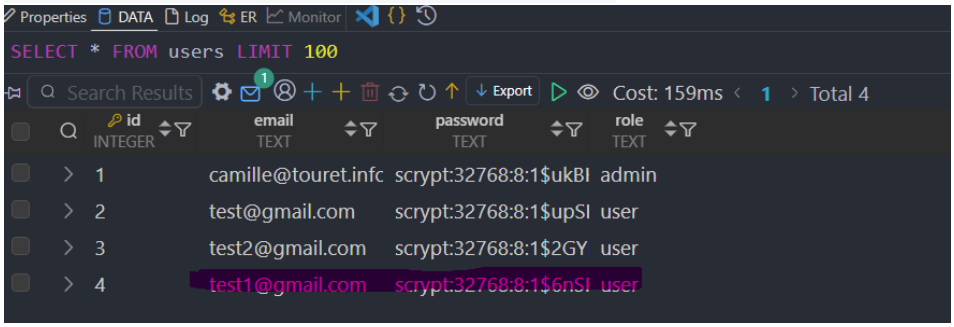
Des payloads XSS ont été injectés dans les champs utilisateurs, mais l'échappement des données via escape() et Jinja2 a empêché toute exécution de script.

Tests CSRF (Cross-Site Request Forgery)

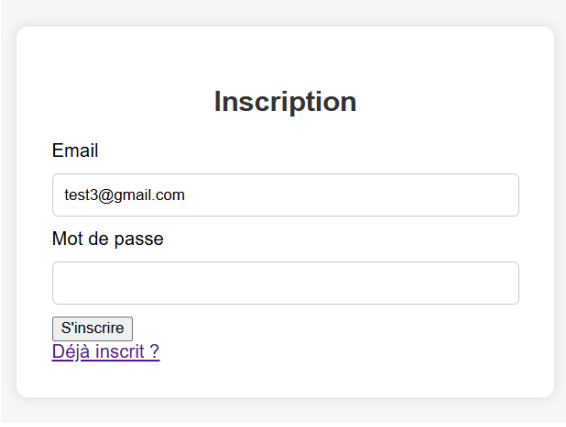
Objectif : Vérifier que les actions sensibles nécessitent un token CSRF valide.

Une comparaison a été réalisée entre la création d'un utilisateur avec et sans jeton CSRF. Lorsque le formulaire contient le token CSRF généré par Flask-WTF, l'inscription est validée et le compte est correctement enregistré en base de données.



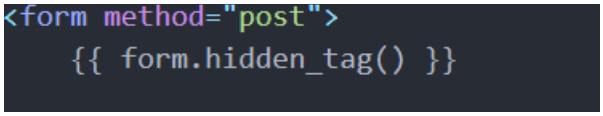


En revanche, après suppression manuelle du champ CSRF dans le formulaire via l'outil d'inspection du navigateur, toute tentative d'inscription est automatiquement refusée par le serveur. Aucun utilisateur n'est créé et aucune redirection n'est effectuée. Ce comportement démontre l'efficacité de la protection CSRF et garantit que seules les requêtes légitimes provenant de l'application sont acceptées.



Preuve dans le script app.py :

Dans *register.html* et *login.html*



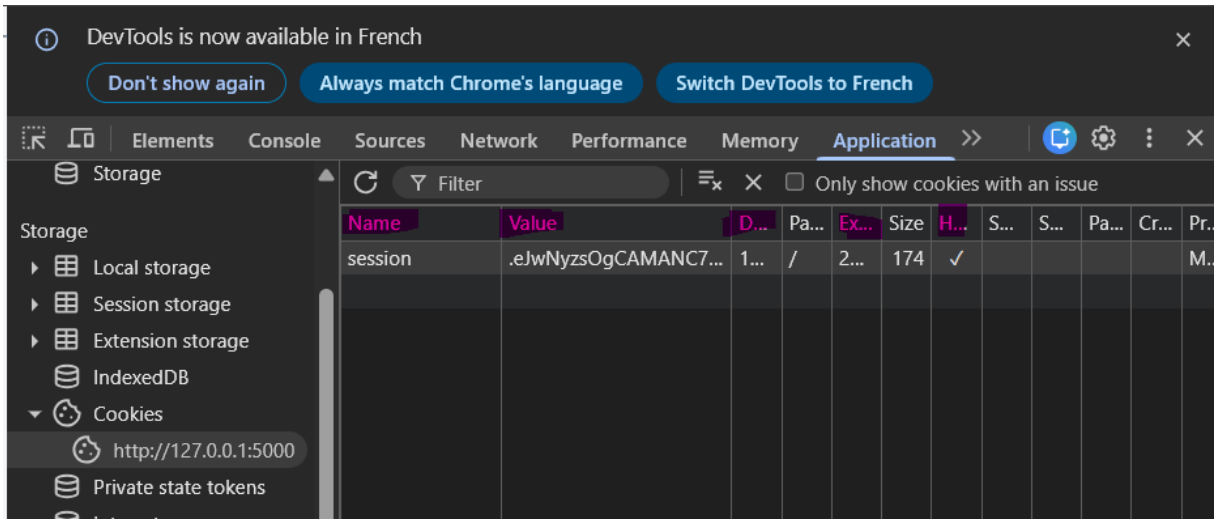
La création d'un compte a été testée sans le token CSRF, ce qui a bloqué la requête, alors qu'avec le token valide l'inscription a été acceptée

Analyse des sessions

Objectif : Vérifier la sécurité des cookies et la gestion des sessions.

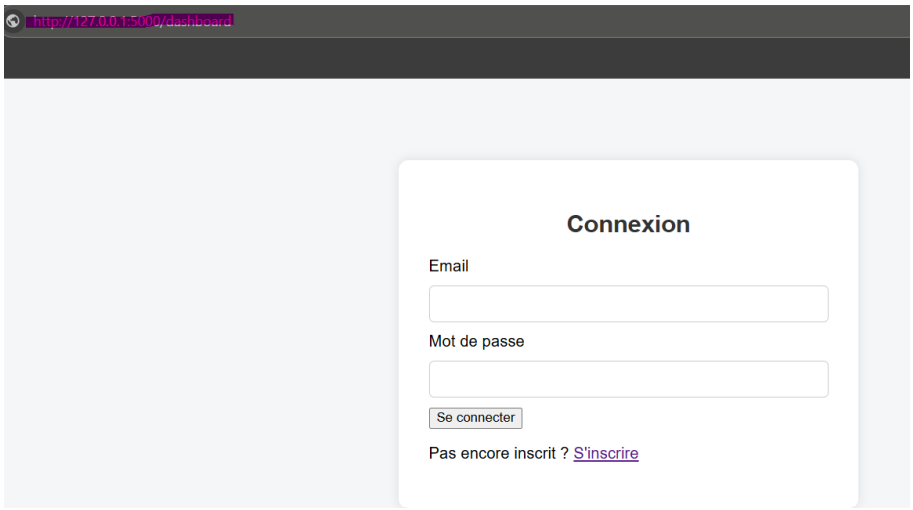
Test 1 - Vérification du cookie de session :

Après authentification sur l'application, les outils de développement du navigateur ont été utilisés pour analyser les cookies stockés par le site. Le cookie de session a été inspecté afin de vérifier ses attributs de sécurité. Il a été constaté que le cookie est configuré avec l'attribut *HttpOnly*, empêchant tout accès via JavaScript, et que sa valeur est chiffrée. Une date d'expiration est également présente, confirmant une durée de session limitée. Ces éléments garantissent une meilleure protection contre le vol de session.



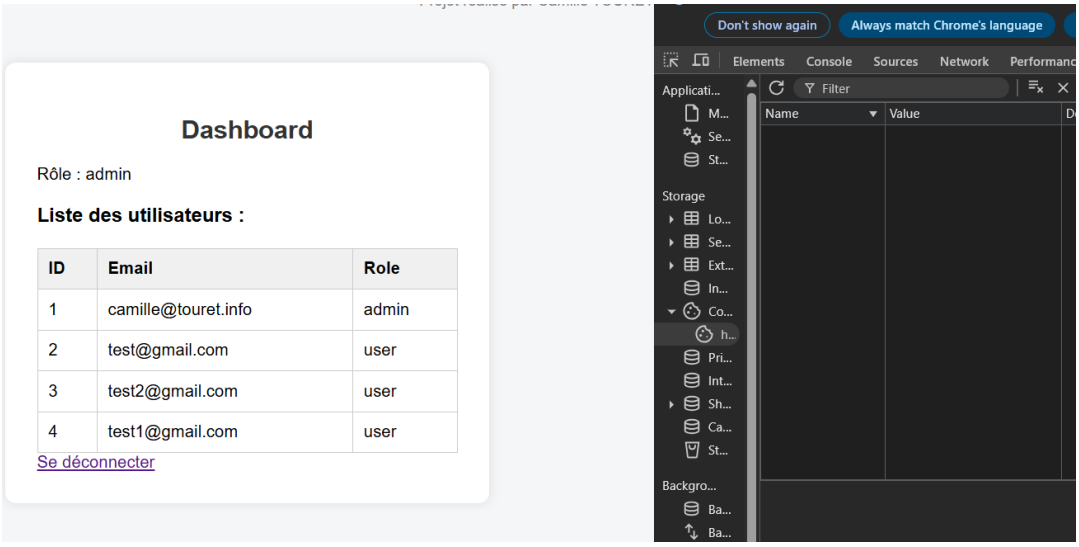
Test 2 - Accès sans authentification :

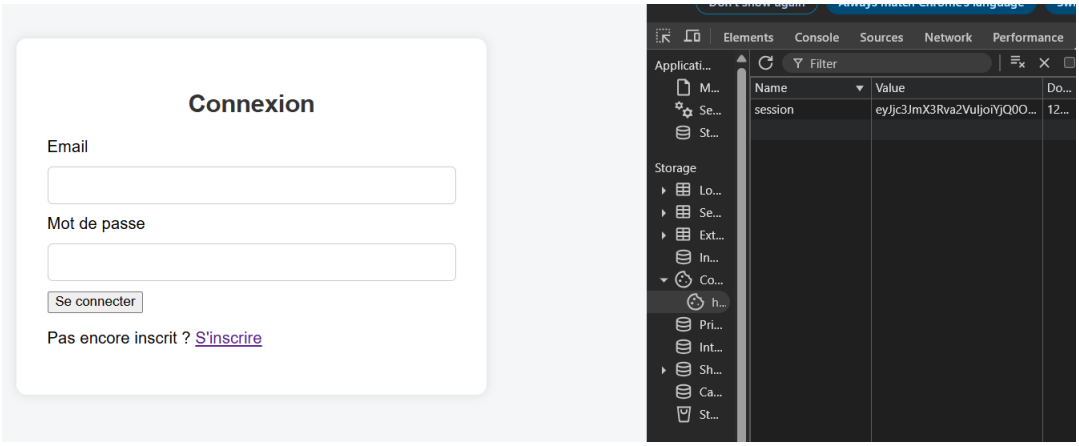
Une tentative d'accès direct à une page protégée de l'application (http://127.0.0.1:5000/dashboard) a été réalisée depuis une nouvelle session de navigation sans être authentifié. L'accès a été automatiquement refusé et une redirection vers la page de connexion a été observée. Ce test confirme que l'application vérifie correctement la présence d'une session valide avant d'autoriser l'accès aux ressources sensibles.



Test 3 - Expiration de session :

Le cookie de session a été supprimé manuellement depuis le navigateur afin de simuler une perte ou un vol de session. Après suppression, une tentative d'accès à une page protégée a été effectuée. L'utilisateur a été automatiquement déconnecté et redirigé vers la page de connexion, ce qui démontre que la session est correctement invalidée en l'absence du cookie.





Preuve dans le script app.py :

```
13  # Protection contre le vol de cookies
14  app.config["SESSION_COOKIE_HTTPONLY"] = True
15  app.config["PERMANENT_SESSION_LIFETIME"] = timedelta(minutes=30)
16  app.secret_key = "camilletouret"
17
```

On remarque bien qu'une protection contre les vols de cookies à bien été mise en place.

```
@app.route("/dashboard")
def dashboard():
    if "user_id" not in session:
        return redirect("/login")
```

Les cookies de session sont configurés avec HttpOnly et la vérification de session bloque tout accès aux pages protégées sans authentification.

Recommandations de Sécurité

Même si l'application est déjà protégée contre SQL Injection, XSS, CSRF et possède une gestion sécurisée des sessions, les recommandations suivantes peuvent renforcer la sécurité :

- **Renforcer le hashage des mots de passe** : utiliser bcrypt ou Argon2 pour protéger davantage les mots de passe.
- **Déployer l'application en HTTPS** : sécuriser toutes les communications et protéger les cookies et tokens.
- **Activer des headers HTTP de sécurité** : Content-Security-Policy, X-Frame-Options et X-Content-Type-Options pour limiter les attaques XSS et clickjacking.
- **Ajouter la Multi-Factor Authentication (MFA)** : renforcer l'accès aux comptes sensibles, notamment les comptes administrateurs.