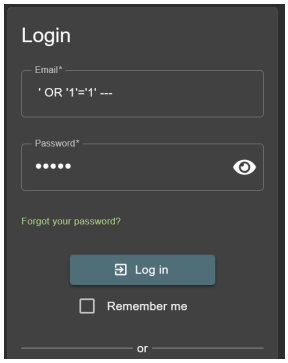
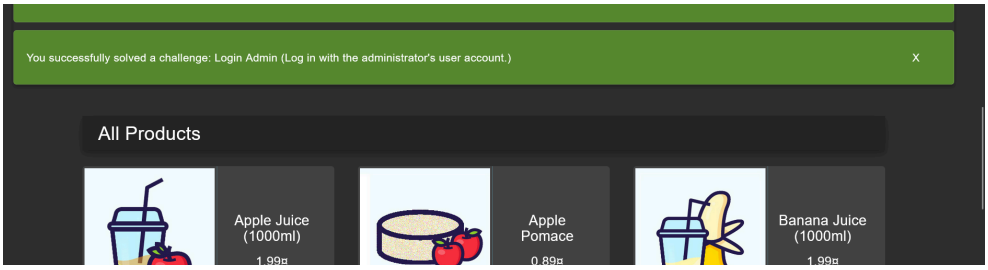
Rapport
Atelier 3:
A03 – Injection

SABRI OMAIMA

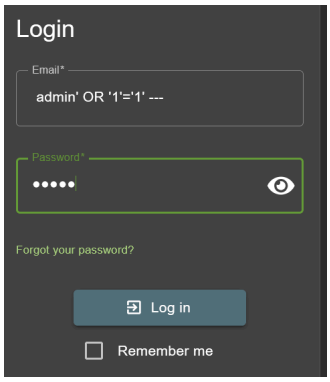
1.Injection SQL dans le formulaire de login

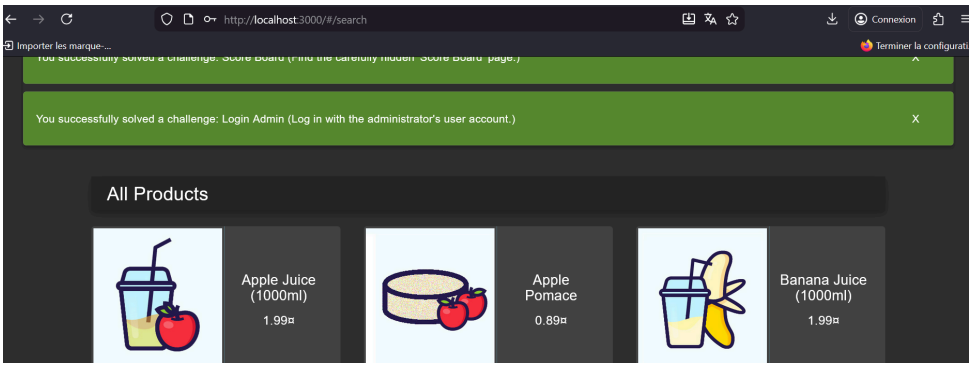
1.Ouvrir Juice Shop → page Login.
2.Saisir dans le champ Email la charge suivante : ' OR '1'='1' --- et dans Password n'importe
3.Soumettre le formulaire.
4.Observer que la connexion a réussi (ou vérifier le token dans le LocalStorage / cookie).





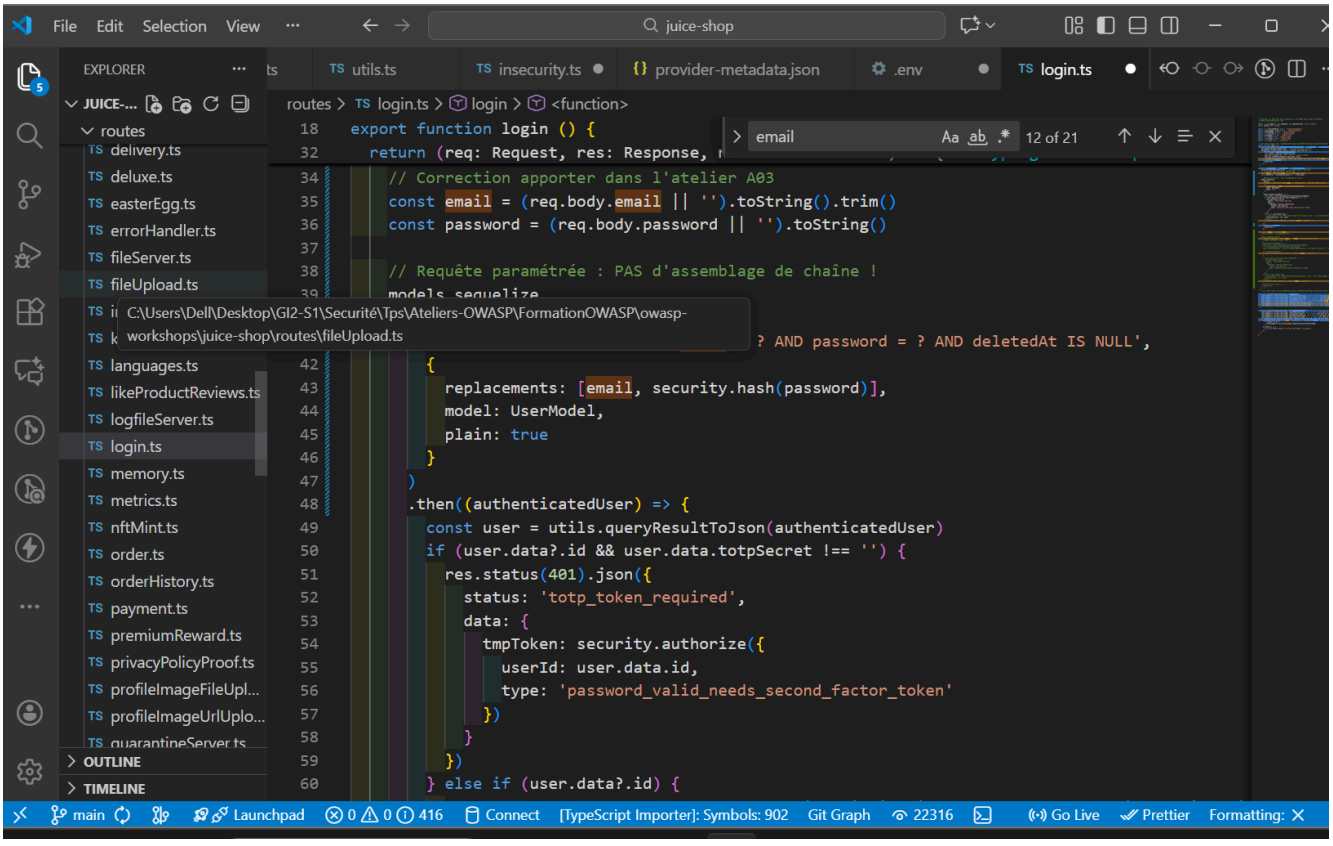
5.Refaites le test avec : admin' OR '1'='1' ---





Correction (serveur) — approche recommandée

1) Utiliser des requêtes paramétrées :



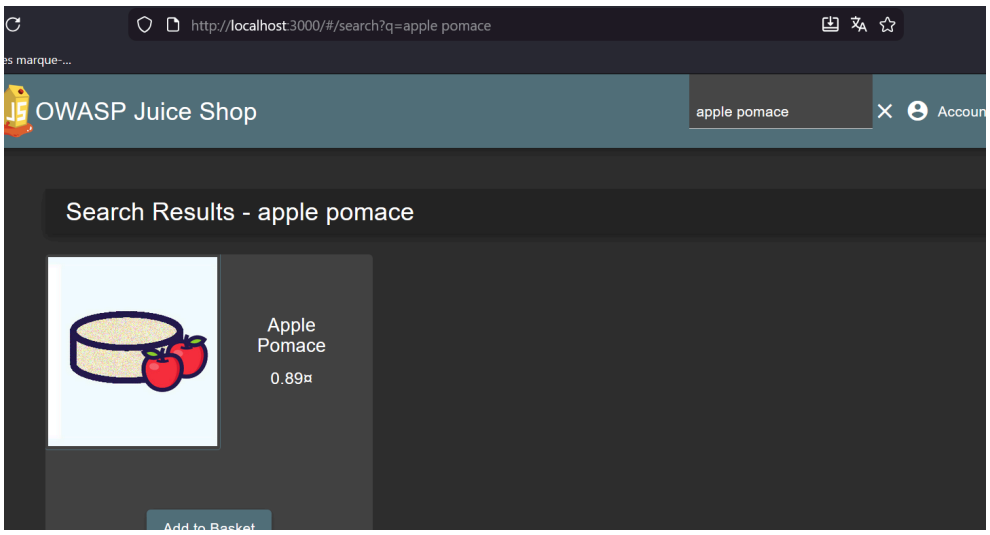
2) On peut aussi utiliser un ORM :
Un ORM (Object Relational Mapper) sert à communiquer avec la base de données sans écrire les requêtes SQL soi-même.
➡️ L'ORM génère automatiquement des requêtes sécurisées
➡️ Impossible de faire une injection SQL
➡️ Tu manipules des objets, pas du SQL

Résumé de la vulnérabilité : Injection SQL de contournement d'authentification
Cette attaque consiste à injecter du code SQL dans un champ de formulaire (comme Email ou Password), afin de modifier la requête exécutée par le serveur.
Lorsque l'utilisateur malveillant entre une charge comme : " OR '1'='1' ---"
voici ce qui se passe :

- C'est une injection SQL de contournement d'authentification : l'attaquant vise directement la page de login.
- L'utilisateur injecte un morceau de SQL dans le formulaire, qui est ajouté tel quel dans la requête du backend.
- La requête originale est modifiée, car les valeurs ne sont pas protégées.
- OR '1'='1' crée une condition toujours vraie, ce qui rend la vérification du mot de passe inutile.
- --- désactive le reste de la requête en le transformant en commentaire.
- Résultat : le système connecte n'importe qui sans connaître le mot de passe, car la condition finale renvoie toujours un utilisateur valide.

2.Injection SQL via paramètre de recherche (URL / query string)

1.Ouvrir Juice Shop.
2.Dans le champ de recherche, cherchez quelques choses. Puis inspectez dans les URL : rest/products/search




```
    })
    console.log(`SELECT * FROM Products WHERE ((name LIKE '%${criteria}%' OR description LIKE
  }
}
```

```
SELECT * FROM Products WHERE ((name LIKE '%test%')) UNION SELECT * FROM users --- OR description LIKE '%%') AN
D deletedAt IS NULL) ORDER BY name
```

Correction :

Correction frontend :

```
// correction apporter dans atelier A03
  search (criteria: string) {

const encoded = encodeURIComponent(criteria || '');
return this.http.get(`${this.hostServer}/rest/products/search?q=${encoded}`).pipe(map((resp

}
```

Correction Backend :

```
// Correction apporter de l'atelier A03      You, 42 minutes ago • Uncom
models.sequelize.query(
`SELECT * FROM Products
WHERE (
  (name LIKE '%' || :criteria || '%'
  OR description LIKE '%' || :criteria || '%')
  AND deletedAt IS NULL
)
ORDER BY name`,
{
  replacements: { criteria },
  type: QueryTypes.SELECT
}
```

1. La vulnérabilité (en bref, sans code)

Le moteur de recherche du site était vulnérable à une injection SQL de type UNION.
Cela signifie que la saisie de l'utilisateur était directement intégrée dans la requête SQL sans être filtrée.
Un attaquant pouvait donc entrer du texte spécial dans la recherche pour :

- modifier la requête SQL originale,
- ajouter une seconde requête via UNION SELECT,
- récupérer les données d'autres tables, notamment les emails et mots de passe des utilisateurs,
- contourner les contrôles prévus.

En résumé :
➡️ l'utilisateur contrôlait une partie de la requête SQL
➡️ et pouvait faire exécuter n'importe quel SQL, ce qui compromet entièrement la base de données.

2. Pourquoi cela fonctionnait

- Le backend prenait le texte entré dans la recherche tel quel,
- Il le collait directement dans la requête SQL,
- Donc tout texte malveillant était interprété par le serveur comme une commande SQL légitime.

Ce type d'attaque est précisément ce qu'on appelle :
➡️ une injection SQL en concaténation de chaînes.

3. Correction appliquée (frontend)

Au niveau du frontend, une amélioration a été apportée en encodant la valeur saisie dans la barre de recherche avant de l'envoyer au serveur.
L'encodage consiste à transformer les caractères spéciaux (comme ',), %, ou ;) en une forme inoffensive pour éviter qu'ils ne perturbent l'URL.
Concrètement :

- la valeur saisie par l'utilisateur est encodée,
- les caractères problématiques sont convertis en leur équivalent sécurisé,
- l'URL envoyée au backend ne contient jamais de caractères bruts pouvant casser la structure de la requête côté serveur.

Cependant, il est important de préciser que :

- cet encodage améliore la robustesse du frontend,
- mais il ne constitue pas une protection complète contre les attaques,
- car un attaquant peut toujours contourner le navigateur et envoyer directement une requête malveillante au serveur.

4.Correction appliquée (backend)

La correction apportée consiste à paramétrer la requête, c'est-à-dire à séparer :

- la requête SQL
- les valeurs fournies par l'utilisateur

Avec cette méthode :

- le texte de l'utilisateur n'est plus interprété comme une commande SQL,
- il est traité comme une simple valeur,
- toutes les tentatives d'injection sont automatiquement neutralisées.

En bref :
➡️ On n'insère plus jamais la saisie dans une requête SQL comme du texte brut.
➡️ Les paramètres garantissent qu'aucune injection est possible.