Rapport
Atelier 4:
A04 - Insecure Design

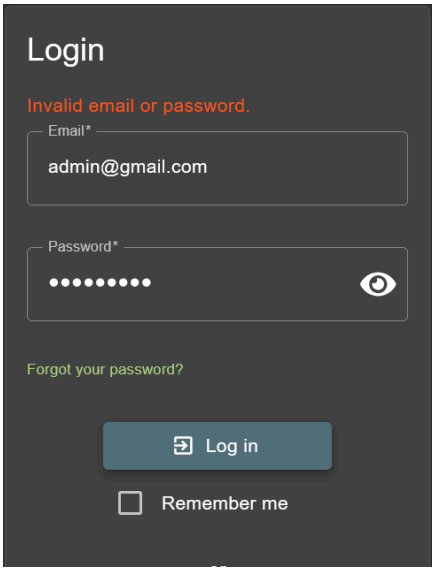
SABRI OMAIMA

1.Flux d'authentification (partie design)

Reproduction :·

- **Observer le formulaire login de Juice Shop.**
- **Essayer de :**

1. **Soumettre un mot de passe incorrect plusieurs fois.**
2. **Modifier les champs via la console (F12) pour changer l'email ou le mot de passe.**



J'ai utilisé la console JavaScript (F12 → Console) pour modifier les champs du formulaire dans le console :

```
>> document.getElementById("email").value = "omaima@gmail.com"
← "omaima@gmail.com"
>> document.querySelector('input[type="password"]').value = "12345";
← "12345"
```

Problème :

- **Le design initial du flux login ne limite pas le nombre de tentatives.**
- **Aucun mécanisme anti-brute-force (captcha, lockout).**
- **Même les comptes admin peuvent être ciblés facilement si un attaquant devine l'URL du login.**

Prévention

1. **Limiter le nombre de tentatives côté serveur (ex : 5 tentatives avant lockout temporaire).**

Ne pas bloquer les comptes de manière permanente.

Solution : Utiliser express-rate-limit

✓**Etapes :**

Installer le module (dans un vrai projet) :

```
PS C:\Users\Dell\Desktop\GI2-S1\Securité\Tps\Ateliers-OWASP\FormationOWASP\owasp-workshops\juice-shop> npm install e
xpress-rate-limit
>>

added 8 packages, removed 3 packages, changed 10 packages, and audited 2165 packages in 2m

239 packages are looking for funding
  run `npm fund` for details

45 vulnerabilities (1 low, 18 moderate, 19 high, 7 critical)

To address all issues possible (including breaking changes), run:
  npm audit fix --force

Some issues need review, and may require choosing
a different dependency.
```

Ajouter ce middleware dans Juice Shop :

```
middlewares > TS loginLimiter.ts > ...
1   import { rateLimit } from 'express-rate-limit'
2
3   // Limiteur par IP - protection générale
4   export const loginLimiter = rateLimit({
5     windowMs: 10 * 60 * 1000, // 10 minutes
6     max: 5, // 5 tentatives max par compte
7     message: 'Trop de tentatives , Veuiller ressayer dans 10 minutes.  ',
8     standardHeaders: true,
9     legacyHeaders: false,
10    skipSuccessfulRequests: false
11  })
12
```

Puis dans server.ts

```
import { loginLimiter } from './middlewares/loginLimiter'

  app.post('/rest/user/login', loginLimiter, login)
```

✓ Correction appliquée :
Mise en place d'une protection anti-brute-force sur la route /rest/user/login
✓ Explication :

- **Ajout d'un middleware Express qui limite la fréquence des tentatives.**
- **Après 5 tentatives ratées, l'IP ou l'utilisateur est bloqué temporairement.**
- **Aucune suppression ou blocage définitif du compte.**
- **Très efficace contre les attaques automatisées.**

C'est le Le rate limiting (ou limitation de fréquence) qui est un mécanisme de sécurité essentiel dans les applications web.
Le module express-rate-limit est une librairie Node.js pour Express qui sert à limiter le nombre de requêtes qu'un client peut faire dans un temps donné.

2. Ajouter captcha ou challenge anti-bot pour éviter les scripts automatisés.

pas besoin d'ajouter de nouveaux fichiers pour le captcha dans OWASP Juice Shop, car le projet contient déjà tout ce qu'il faut pour gérer les captchas.
Donc il suffit juste de connecter la vérification du captcha avant ou pendant la route /rest/user/login

```
  // Correction A04 using captcha verification middleware      Yo
app.post('/rest/user/login', verifyCaptcha(), login())
```

3. Surveiller et alerter sur les tentatives multiples depuis la même IP ou compte. (Pensez au Honey Pot)

Honey Pot
C'est une technique anti-bot très intelligente :

- **Tu mets un champ ou un endpoint invisible pour les humains (via CSS ou JS, par exemple un input display:none).**
- **Un vrai utilisateur ne remplira jamais ce champ.**
- **Un bot, qui remplit tous les champs automatiquement, le remplira → tu sais que c'est un bot.**
- **Tu peux alors bloquer la requête ou alerter.**

Exemple de champ Honey Pot dans un formulaire HTML :
<input type="text" name="website" style="display:none">

- **Si website est rempli → bot détecté.**
- **Si vide → humain normal → passe.**

Dans login.component.html on ajoute ce champ :

```
  <input type="text" name="hiddenTrap" style="display:none">
```

On cree le fichier : /server/middlewares/securityMonitor.ts

```ts
import type { Request, Response, NextFunction } from 'express'

// Dictionnaires pour enregistrer les tentatives
const ipAttempts: Record<string, number> = {}
const userAttempts: Record<string, number> = {}

export function honeyPot (req: Request, res: Response, next: NextFunction) {
  if (req.body.hiddenTrap) {
    console.log('🪤 HoneyPot triggered ! Suspicious bot detected : ', req.ip)
    return res.status(400).json({ message: 'Bot détecté !' })
  }
  next()
}

export function monitorAttempts (req: Request, res: Response, next: NextFunction) {
  // ✓ Récupérer l'IP sans risque de undefined
  const ip = (req.ip ?? req.~~connection~~?.remoteAddress ?? 'unknown-ip')

  // ❌ Tu utilisais "attempts" → il n'existe pas → erreur TS2552
  // ✓ Correction : utiliser la bonne variable : ipAttempts
  ipAttempts[ip] = (ipAttempts[ip] || 0) + 1

  if (ipAttempts[ip] > 10) {
    console.log(`⚠️ Suspicious IP flood: ${ip}`)
    return res.status(429).json({ message: 'Trop de tentatives depuis cette IP.' })
  }

  // ✓ Comptage par email (tentatives bruteforce)
  const email = req.body.email

  if (email) {
    userAttempts[email] = (userAttempts[email] || 0) + 1

    if (userAttempts[email] > 5) {
      console.log(`🚨 Possible bruteforce on account: ${email} from IP: ${ip}`)
    }
  }

  next()
}
```

on l'import dans sever.ts

```ts
// import { loginLimiter } from './middlewares/loginLimiter'
import { honeyPot, monitorAttempts } from './middlewares/securityMonitor'
```

```ts
app.post(
  '/rest/user/login',
  honeyPot, // bloque les bots
  monitorAttempts, // surveille IP et email      You, 5 hours ago • Unc
  login() // ton login existant
)
```

4. Penser dès la conception à l'abus potentiel des flux d'authentification (mindset hacker).

1. Énumération de comptes (Account Enumeration)
Scénario d'attaque :
- L'attaquant essaie plusieurs emails pour voir si le message d'erreur change :
  - "Email incorrect" → Le compte n'existe pas
  - "Mot de passe incorrect" → Le compte existe !

Mindset hacker :
- "Je peux découvrir quels emails sont enregistrés sans connaître les mots de passe."

Prévention dès le design :
- Utiliser un message d'erreur générique : "Email ou mot de passe incorrect"
- Retourner le même temps de réponse pour éviter le timing attack

2. Attaque par force brute (Brute Force)
Scénario d'attaque :
- L'attaquant teste automatiquement des milliers de mots de passe sur un compte connu (ex: admin@juice-sh.op).

Mindset hacker :
- "Si je lance 10 000 requêtes, vais-je être bloqué ?"

Prévention dès le design :
- Rate limiting côté serveur (5 tentatives max par IP/compte)
- Délai exponentiel après chaque échec (1s, 5s, 30s...)
- Captcha après 3 tentatives échouées
- Lockout temporaire (15 min après 5 échecs)

3. Credential Stuffing
Scénario d'attaque :
- L'attaquant utilise des millions de combinaisons email/mot de passe volées d'autres sites (leaks).

Mindset hacker :
- "Les utilisateurs réutilisent souvent leurs mots de passe. Je vais tester des listes de credentials volées."

Prévention dès le design :
- Vérifier les mots de passe contre des bases de données de fuites (ex: HaveIBeenPwned API)
- Exiger l'authentification multi-facteurs (MFA)
- Détecter les logins depuis des localisations inhabituelles

4. Session Hijacking / Fixation
Scénario d'attaque :
- L'attaquant vole ou force un token de session pour accéder au compte sans mot de passe.

Mindset hacker :
- "Si je vole le cookie de session, puis-je me connecter sans credentials ?"

Prévention dès le design :
- Régénérer le session ID après login
- Utiliser des cookies avec flags HttpOnly, Secure, SameSite
- Expiration courte des tokens (15-30 min)
- Détecter les sessions multiples simultanées

5. Password Reset Abuse
Scénario d'attaque :
- L'attaquant exploite le flux de réinitialisation de mot de passe :
  - Tokens prévisibles ou réutilisables
  - Pas d'expiration des liens
  - Réponse différente si l'email existe ou non

Mindset hacker :
- "Puis-je réinitialiser le mot de passe d'un autre utilisateur ?"

Prévention dès le design :
- Tokens aléatoires cryptographiquement sûrs
- Expiration rapide (15-30 min)
- Invalider le token après utilisation
- Message générique même si l'email n'existe pas

5. Valider les entrées côté serveur, même si la validation existe côté client.

Pourquoi la validation côté client NE SUFFIT PAS ?
Le client est sous contrôle total de l'attaquant
Un attaquant peut facilement contourner la validation côté client
Mais l'attaquant peut :

1. Désactiver JavaScript dans le navigateur
2. Modifier le code via DevTools (F12)
3. Envoyer des requêtes directement avec curl, Postman, Burp Suite :

en contournant complètement toute validation côté client.