Marion BAILLEUX

PARTIE 1

Question 1 - Injection SQL

1. Une injection SQL est le fait qu'un attaquant puisse insérer du code SQL dans un input utilisateur, qui viendrait se greffer dans une requête SQL mal sécurisée pour accéder à des données sensibles et/ou exécuter des commandes dangereuses.

```
const user = req.body.email;
const passWd = req.body.passWd;
const query = `SELECT * FROM users WHERE email = '${email}' AND password = '${passWd}'`;
```

2. En tapant ' OR '1'='1 dans l'input email, un attaquant pourrait se connecter sans identifiants valides, possiblement comme le premier utilisateur dans la base (souvent admin). Cela fonctionne car la condition a condition `email = '' OR '1'='1'` est toujours vraie.

```
SELECT * FROM users WHERE email = '' OR '1'='1' AND password = 'anything';
```

3. Une méthode pour s'en protéger serait d'utiliser des requêtes préparées, avec un ORM par exemple. Les données utilisateur ne sont pas insérées directement dans la chaîne SQL mais envoyées séparément, ce qui empêche l'interpréteur SQL de traiter les données comme du code.

```
const { email, passWd } = req.body;

const user = await prisma.user.findFirst({
  where: {
    email: email,
    password: passWd, // En théorie il faudrait hasher le mdp
  },
});
```

Question 2 - XSS

1. XSS réfléchi : le code malveillant est injecté via une URL ou un champ de formulaire, puis immédiatement renvoyé dans la réponse sans être stocké côté serveur.

   XSS stocké : le code est injecté puis enregistré (ex. : base de données) et s'exécute chaque fois que la page concernée est affichée.

   BUT : Voler des cookies, effectuer des redirections frauduleuses, ou espionner des saisies de clavier.

2.

```
<input type="text" id="email" value="<script>alert('XSS')</script>">
```

3. Pour s'en protéger, il faudrait filtrer et échapper les entrées utilisateur et activer une politique de sécurité CSP

Question 3 - CSRF

1. Le principe de cette attaque est de transmettre à un utilisateur authentifié une requête HTTP falsifiée qui pointe sur une action interne au site, afin qu'il l'exécute sans en avoir conscience et en utilisant ses propres droits.
   - L'utilisateur est connecté à un site web (par exemple : sa banque).
   - L'attaquant lui envoie un lien piégé (par e-mail, message ou site web).
   - L'utilisateur clique sur ce lien.
   - Ce lien déclenche une requête vers le site légitime (ex : virement d'argent).
   - Comme l'utilisateur est connecté, le site exécute la requête avec ses droits (via les cookies de session envoyés automatiquement).

2. Token CSRF : un jeton secret généré par le serveur et vérifié à chaque requête pour s'assurer qu'elle vient bien du site légitime.

   Attribut SameSite : une règle côté navigateur qui limite l'envoi des cookies aux requêtes provenant du même site, bloquant ainsi les requêtes cross-site non désirées.

   Le token agit côté serveur, SameSite côté client.

3. Deux méthodes de protection concrètes sont :
   - Utiliser un token CSRF :
     Générer un token unique côté serveur, l'inclure dans les formulaires et vérifier sa présence à chaque requête sensible.

   - Attribut SameSite sur les cookies :
     Configurer les cookies de session avec SameSite=Lax ou SameSite=Strict pour empêcher leur envoi sur des requêtes cross-site.

Question 4 - Sécurité des sessions et accès

1. Un cookie httpOnly est cookie qui n'est accessible que par requête HTML et pas via JavaScript. C'est important car les cookies ne peuvent donc pas être récupérés via du code JavaScript.
2. L'attaque IDOR est le fait de ne pas vérifier si un utilisateur est autorisé à accéder à une ressource spécifique, permettant à n'importe quel utilisateur de consulter ou modifier les données d'un autre utilisateur.

Pour s'en protéger :

- Vérifier les permissions avant d'accéder à une ressource
- Utiliser des tokens d'authentification (ex : JWT) pour garantir que seul l'utilisateur légitime accède à ses données
- Ne pas exposer d'identifiants sensibles issus de la BDD dans l'URL

Question 5 - Bonnes pratiques générales

1. Utiliser des tokens JWT pour l'auth : Le token étant signé, cela permet de sécuriser l'accès aux informations d'un utilisateur.
2. Faire des requêtes préparées : Afin d'éviter les injections SQL et l'accès à des données sensibles
3. Toujours échapper et filtrer les entrées utilisateurs : Cela permet d'éviter différents types d'attaques comme les attaques XSS ou les injections SQL

PARTIE 2

1. Identifier les vulnérabilités :
   - Injection SQL dans la route login

```
const user = users.find(u => u.username == username && u.password == password);
```

   - Faille XSS : les messages des utilisateurs sont stockés puis affichés sans validation ni échappement. Les messages malveillants sont stockés dans le tableau [messages] et affichés à tous les visiteurs (XSS stocké).

```
const { message } = req.body;
messages.push(message);
```

   - Faille IDOR dans la route /dashboard. Alice peut faire la requête GET /dashboard?id=2 et voir le profil de Bob

```
const userId = parseInt(req.query.id || req.session.user.id);
const user = users.find(u => u.id === userId)
```

   - Absence de CSRF sur /edit-profile. Alice pourrait exécuter du code malveillant sans s'en rendre compte. Le navigateur inclut automatiquement le cookie de session d'Alice et le serveur modifie le profil d'Alice sans qu'elle le sache

```
app.post('/edit-profile', (req, res) => {
  user.username = req.body.username;
```

- Informations sensibles dans la session : le mdp est stocké en session

```
req.session.user = user;
```

- Cookie de session non sécurisé