Rendu 8 - Semaine 10 - 6 Mai

Groupe 4 - Clément Miesse, Ambroise Mostin et Cyril Wastchenko → **Lead Clément**

Sécurité

Quels sont les biens à protéger?

- Le serveur
- La base de données
- Le code

Quelles sont les vulnérabilités et les menaces de ces biens?

Serveur → Accès non autorisé, données vulnérables
Base de données → Injection SQL, copie non autorisée de données sensibles, abus/élévation de privilège
Code → Cross-Site Scripting (XSS), Buffer Overflow et Broken Authentication

Pour chaque menace, quels sont les risques associés?

Risques :

1. Accès non autorisé au serveur / accès aux données sur le serveur
2. Copie de données de la base de données
3. Accès aux données de la BDD suite à un abus/une élévation de privilèges
4. Injection SQL dans la BDD
5. Cross-Site Scripting
6. Buffer Overflow
7. Broken Authentication

Contre-mesures (cfr ci-dessous) :

1. Authentification par clés RSA, désactivation de la connexion via ROOT
2. Connexion à la DB via un seul utilisateur, par mot de passe
3. /
4. /
5. /
6. Testing → détection d'anomalies dans le code
7. Mots de passe Hashés

Documenter les risques résiduels

- Accès à la base de données via Root par mot de passe - **à améliorer**
- Abus / élévation de droits dans la BDD → **Non géré**
- Injection SQL → **Non géré**
- Cross-site scripting → **Non géré**

Assurer la maintenance (plan de suivi, monitoring, stratégies de réaction/mitigation

Utilisation de PM2 → **OK**

Contres-mesures

Authentification par clés RSA

- Désactivation de la connexion sur l'utilisateur root
- Création et utilisation de clés RSA pour se connecter
- Désactivation de la connexion par mot de passe pour tous les utilisateurs

Testing

Tests unitaires

- Fichier Common.test.js (fichier de tests unitaires utilisant Jest pour tester Common.js)
- Les méthodes getUser(), getToken(), removeUserSession(), setUserSession(token, user) sont testées
- Les autres méthodes de l'api sont de simple appels à l'API et enregistre les données dans des State ou sont intestables même en hardcodant
- Suite à cela, le code coverage est très faible, 5.85% des statements, 3.51% des branches, 3.64% des fonctions et 5.39% des lignes sont testées. Seuls les méthodes de Utils/Common.js sont correctement testées

Tests d'API

Vérification sur le statut → Retour 200
Vérification sur les données → Données ponctuelles pour chaque fonction

Code coverage

- **Couverture des déclaration (statement)** - chaque déclaration du programme a-t-il été exécuté?
- **Couverture des branches** - chaque branche de chaque structure de contrôle (comme dans les instructions *if* et *case*) a-t-elle été exécutée? Par exemple, étant donné une instruction *if* , les branches true et false ont-elles été exécutées?
- **Couverture des fonctions** - chaque fonction du programme a-t-elle été appelée?
- **Couverture des lignes** - Chaque ligne exécutable du fichier source a-t-elle été exécutée?