Rapport de Patch

Sécurisation du Backend

Projet : IPSSI_PATCH

Auteur : GAVI Holali David

Date : 11 décembre 2025

IPSSI - Mastère Cybersécurité et Cloud Computing

Table des matières

1 Introduction

Le backend du projet `IPSSI_PATCH` contenait plusieurs vulnérabilités critiques mettant en danger :

— L'intégrité de la base de données
— La confidentialité des utilisateurs
— La disponibilité du service
— La sécurité des navigateurs clients

L'objectif de ce patch est de corriger ces failles sans complexifier inutilement le code, et d'expliquer clairement :

— La vulnérabilité
— L'impact
— La correction
— Pourquoi cette correction est efficace

Les correctifs appliqués concernent principalement :

— Injection SQL
— XSS stockée
— Stockage de mots de passe en clair
— Validation insuffisante des entrées
— Amélioration structurelle via middlewares
— Sanitation des inputs

2 Vue d'ensemble des Corrections Apportées

Le tableau ci-dessous présente une synthèse des vulnérabilités identifiées et des correctifs appliqués.

Vulnérabilité	**Risque**	**Patch appliqué**
Injection SQL (`/user`)	Haute	Requêtes paramétrées + validation ID
Exécution SQL arbitraire (`/query`)	Critique	Suppression totale
XSS stockée	Haute	Sanitization backend
Mots de passe en clair	Critique	Hachage bcrypt
Absence de validation	Haute	Middlewares `validateUserId` et `sanitizeComment`
CORS ouvert	Moyen	Restriction aux origines autorisées

Table 1 – Récapitulatif des vulnérabilités et correctifs

3 Patch 1 – Injection SQL

3.1 Vulnérabilité

Avant patch, la route `/user` exécutait directement le SQL envoyé par le client :

```
db.all(req.body)
```

Impact :
Un utilisateur pouvait envoyer :

```
DROP TABLE users;
SELECT * FROM sqlite_master;
UPDATE users SET password='hacked';
```

Résultat : **Perte totale du contrôle de la base.**

3.2 Correction appliquée

La route `/user` utilise désormais :
— Un middleware de validation
— Une requête paramétrée
— Un retour propre
Nouveau code patché :

```
app.post('/user', validateUserId, (req, res) => {
  const { id } = req.body;

  db.get(
    "SELECT id, name FROM users WHERE id = ?",
    [id],
    (err, row) => {
      if (err) {
        console.error("SQL Error:", err.message);
        return res.status(500).json({ error: err.message });
      }

      res.json(row ? [row] : []);
    }
  );
});
```

Pourquoi ce correctif ?
— Les requêtes paramétrées empêchent toute injection SQL
— L'ID est contrôlé et nettoyé avant d'atteindre la base
— Le backend ne dépend plus du contenu envoyé par l'utilisateur

4 Patch 2 – XSS Stockée (Commentaires)

4.1 Vulnérabilité

La route `/comment` acceptait du HTML brut :

```
const { content } = req.body;
```

Un attaquant pouvait envoyer :

```
<script>alert("Hacked")</script>
```

Ce script s'exécutait chez tous les visiteurs → **XSS stockée.**

4.2 Correction : Sanitization backend

Ajout d'un middleware dédié :

```
function sanitizeComment(req, res, next) {
  let { content } = req.body;

  if (!content || content.length === 0) {
    return res.status(400).json({
      error: "Comment cannot be empty"
    });
  }

  // Protection XSS (sanitization minimale)
  content = content
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;");

  req.body.content = content;
  next();
}
```

Réutilisé dans la route :

```
app.post('/comment', sanitizeComment, (req, res) => {
  const { content } = req.body;

  db.run(
    `INSERT INTO comments (content) VALUES (?)`,
    [content],
    (err) => {
      if (err) {
        return res.status(500).json({ error: err.message });
      }
      res.json({ success: true });
    }
  );
});
```

Pourquoi ?

— Empêche l'exécution de code HTML/JS dans les commentaires

— Protège tous les utilisateurs contre les attaques XSS persistantes
— Simple, efficace, et ne change pas la logique de l'application

5 Patch 3 – Hachage des Mots de Passe

5.1 Vulnérabilité

Les mots de passe étaient stockés en clair :

```
const password = u.login.password;
```

Impact :
Si la base fuit :
— Tous les mots de passe sont immédiatement utilisables
— Responsabilité légale (CNIL/RGPD)

5.2 Correction : bcrypt.hash()

Ajout de la dépendance :

```
const bcrypt = require('bcrypt');
```

Code modifié dans `insertRandomUsers()` :

```
const hashedPassword = await bcrypt.hash(password, 10);

db.run(
  `INSERT INTO users (name, password) VALUES (?, ?)`,
  [fullName, hashedPassword],
  (err) => {
    if (err) console.error(err.message);
  }
);
```

Pourquoi ?
— Empêche l'accès au mot de passe même si la base est compromise
— Force brute très difficile voire impossible
— Norme de sécurité OWASP

6 Patch 4 – Middlewares de Validation

Deux nouveaux middlewares ont été créés :

6.1 validateUserId

```
function validateUserId(req, res, next) {
  const { id } = req.body;

  if (!id || isNaN(id)) {
    return res.status(400).json({ error: "Invalid ID" });
  }
```

```
7
8   next();
9 }
```

But :

— Empêcher l'envoi d'un ID vide, NULL, SQL-like ou non numérique
— Réduire la surface d'attaque

6.2 sanitizeComment

```
1  function sanitizeComment(req, res, next) {
2    let { content } = req.body;
3
4    if (!content || content.length === 0) {
5      return res.status(400).json({
6        error: "Comment cannot be empty"
7      });
8    }
9
10   content = content
11     .replace(/&/g, "&amp;")
12     .replace(/</g, "&lt;")
13     .replace(/>/g, "&gt;")
14     .replace(/"/g, "&quot;");
15
16   req.body.content = content;
17   next();
18 }
```

But :

— Protéger l'application contre les XSS
— Centraliser la sécurité

7 Patch CORS – Restriction

Avant :

```
1 app.use(cors());
```

Après :

```
1 app.use(cors({
2   origin: "http://localhost:3000"
3 }));
```

Pourquoi ?

— Empêche qu'un autre site consomme ton API à ton insu
— Réduit certains risques de CSRF-like

8 Code Backend Final Patché

Les éléments importants modifiés/ajoutés :

```
const bcrypt = require('bcrypt');

app.use(express.json());
app.use(cors({ origin: "http://localhost:3000" }));

function validateUserId(req, res, next) { ... }
function sanitizeComment(req, res, next) { ... }

const hashedPassword = await bcrypt.hash(password, 10);

app.post('/user', validateUserId, ...)

app.post('/comment', sanitizeComment, ...)
```

9 Conclusion

Les correctifs apportés ont permis de transformer un backend vulnérable en backend robuste et conforme aux bonnes pratiques OWASP.

9.1 Les risques majeurs corrigés :

Injection SQL
Exécution SQL arbitraire
XSS stockée
Stockage de mots de passe en clair
Absence de validation
CORS trop permissif

Le backend est maintenant prêt pour être intégré dans un environnement plus sécurisé et pour un rendu académique ou professionnel.

10 Annexes – Middlewares complets

10.1 validateUserId

```
function validateUserId(req, res, next) {
  const { id } = req.body;

  if (!id || isNaN(id)) {
    return res.status(400).json({ error: "Invalid ID" });
  }

  next();
}
```

10.2 sanitizeComment

```
function sanitizeComment(req, res, next) {
  let { content } = req.body;

  if (!content || content.length === 0) {
    return res.status(400).json({
      error: "Comment cannot be empty"
    });
  }

  content = content
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;");

  req.body.content = content;
  next();
}
```

11 Références

- OWASP Top 10 : https://owasp.org/www-project-top-ten/
- OWASP SQL Injection Prevention : https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- OWASP XSS Prevention : https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
- bcrypt Documentation : https://www.npmjs.com/package/bcrypt
- CORS Configuration : https://expressjs.com/en/resources/middleware/cors.html