Rapport des vulnérabilités — The Broken API

Ce document rassemble les failles identifiées dans l'API fournie, leur impact, un exemple d'exploitation et des corrections définitives recommandées.

> *Fichiers analysés :* `server.js`, `package.json`

Résumé exécutif

L'API contient plusieurs vulnérabilités majeures et faciles à exploiter : secrets codés en dur, stockage de mots de passe en clair, injections SQL, contrôle d'accès insuffisant, vulnérabilités XSS et fuite d'informations via les messages d'erreur. Les corrections proposées visent à rendre l'application adaptée à un environnement de production.

Faille n°1 : Secrets codés en dur

- Localisation : `server.js` (const `ADMIN_TOKEN` près du haut du fichier)
- Description : Le token d'administration est stocké en clair dans le code source.
- Exploitation : Un attaquant ayant accès au repo (ou à un dump) récupère le token et peut effectuer des actions privilégiées (ex : suppression d'utilisateurs via `/api/delete-user`).
- Impact : Compromission complète des opérations d'administration (élevation de privilèges, suppression/modification des comptes).
- Correction proposée :
  - Retirer toute valeur secrète du code source. Utiliser des variables d'environnement (fichier `.env` pour le dev, gestionnaire de secrets en prod).
  - Exemple (utiliser `dotenv`) :

```
# .env (ne pas commit)
ADMIN_TOKEN=un_token_long_et_securise
```

Dans le code :

```
require('dotenv').config();
const ADMIN_TOKEN = process.env.ADMIN_TOKEN;
```

- Notes : en production, utiliser un gestionnaire de secrets (Vault, AWS Secrets Manager...) et des tokens avec rotation.

Faille n°2 : Mots de passe en clair dans la base

- Localisation : `server.js` (INSERT initial des users avec mots de passe en clair)
- Description : Les mots de passe sont stockés en texte clair dans la table `users`.
- Exploitation : Base de données compromise -> tous les mots de passe utilisateurs divulgués.
- Impact : Vol d'identifiants, usurpation d'identité, risques étendus si les utilisateurs réutilisent leurs mots de passe.
- Correction proposée :
  - Ne jamais stocker de mot de passe en clair. Utiliser un algorithme de hachage sécurisé (bcrypt, argon2) avec sel.

- Exemple (bcrypt) :

```
const bcrypt = require('bcrypt');
const hashed = await bcrypt.hash(plainPassword, 12);
// stocker hashed dans la DB
```

- Notes : prévoir un processus de migration pour les comptes existants (forcer reset password si nécessaire).

Faille n°3 : Injection SQL

- Localisation : `server.js`
  - `/api/user` : construction de la requête `SELECT ... WHERE username = '${username}'` (concaténation) — ligne d'assemblage de requête
  - `/api/delete-user` : `DELETE FROM users WHERE id = ${id}` (concaténation)
- Description : Les paramètres utilisateur sont insérés directement dans les requêtes SQL — vecteur d'injection SQL.
- Exploitation : En fournissant des payloads malveillants, un attaquant peut exécuter des requêtes arbitraires (exfiltration de données, suppression de lignes, élévation de privilèges).
- Exemple d'exploitation (GET) :

```
GET /api/user?username=admin' OR '1'='1
```

Ce qui peut transformer la requête en un SELECT retournant plusieurs lignes.

- Correction proposée :
  - Utiliser systématiquement des requêtes préparées/paramétrées. Pour sqlite3 :

```
const sql = 'SELECT id, username, role FROM users WHERE username = ?';
db.get(sql, [username], (err, row) => { ... });
```

Et pour DELETE :

```
db.run('DELETE FROM users WHERE id = ?', [id], function(err) { ... });
```

- Notes : valider et normaliser les entrées (type, range) avant usage.

Faille n°4 : Contrôle d'accès trop faible / gestion du token

- Localisation : `server.js`, endpoint `/api/delete-user`
- Description : Contrôle basé sur la simple égalité d'une valeur d'en-tête. Pas de format standard (ex : `Bearer`), pas d'expiration, pas de journalisation, pas d'authentification pour d'autres endpoints.
- Exploitation : Token volé -> actions admin ; pas de revocation/expiration -> persistance d'accès.
- Impact : Compromission administrative, suppression/modification de données sensibles
- Correction proposée :
  - Utiliser une solution d'authentification robuste (JWT avec signature et expiration, ou sessions serveur + cookies sécurisés) et RBAC (roles clairement définis).
  - Exemple minimal : exiger header `Authorization: Bearer <token>` et vérifier le token côté serveur. Pour production, vérifier signature du JWT et rôles.

- Ajouter journalisation (who, when) pour toutes les actions sensibles et mécanisme de rotation/révocation des clés.

Faille n°5 : Cross-Site Scripting (XSS) via affichage HTML dynamique

- Localisation : `server.js`, endpoint `/api/welcome` qui fait `res.send(`

  # Bienvenue..., ${name} !

  `);
- Description : Le paramètre `name` est injecté dans une réponse HTML sans échappement -> XSS réfléchi possible.
- Exploitation : En fournissant `?name=<script>...</script>`, un navigateur affichant cette page exécute le script.
- Impact : Exécution de JS côté client, vol de cookies, usurpation de session, redirection malveillante.
- Correction proposée :
  - Éviter d'envoyer de l'HTML directement depuis des paramètres non-sanitized. Préférer JSON pour une API.
  - Si HTML nécessaire, échapper correctement les données (utiliser des librairies de templating sécurisées ou fonctions d'échappement).
  - Exemple : renvoyer JSON : `res.json({ message: 'Bienvenue ...', name: escapedName })`.

Faille n°6 : Fuite d'information via messages d'erreur

- Localisation : `server.js`, endpoint `/api/debug` qui lance `throw new Error("Base de données inaccessible sur 192.168.1.50:5432")` et d'autres endroits qui renvoient `err.message` au client
- Description : Le serveur renvoie des messages d'erreur détaillés contenant des informations réseau et d'infrastructure.
- Exploitation : Un attaquant peut collecter des informations sensibles (adresses IP, ports, messages internes) pour préparer des attaques ciblées.
- Impact : Réduction de la confidentialité de l'architecture, facilitation de la reconnaissance.
- Correction proposée :
  - Ne jamais renvoyer les détails internes d'erreur au client. Logguez les détails côté serveur (fichier log ou système central) et renvoyez un message générique au client.
  - Exemple :

```
console.error(err);
res.status(500).json({ error: 'Erreur interne' });
```

Faille n°7 : Absence de headers de sécurité & hardening HTTP

- Localisation : configuration globale (aucune utilisation de `helmet` ou CSP)
- Description : Pas de headers comme Content-Security-Policy, X-Content-Type-Options, X-Frame-Options, etc.
- Impact : Facilite XSS, clickjacking, sniffing MIME.
- Correction proposée :

- Installer et configurer `helmet` (ou ajouter manuellement les headers). Exemple : `app.use(require('helmet')())`.
- Définir CSP adapté si des ressources tierces sont chargées.

Faille n°8 : Absence de validation d'entrée

- Localisation : endpoints qui acceptent `req.query` / `req.body` sans validation
- Description : Paramètres non validés (types, longueur, format) permettant injections, débordements ou données invalides.
- Exploitation : injection SQL, XSS, Out-of-bounds selon le contexte.
- Impact : comportement imprévu, sécurité compromise.
- Correction proposée :
  - Utiliser un schéma de validation (Joi, express-validator, zod) pour valider toutes les entrées avant usage.

Faille n°9 : Mécanismes d'authentification et transport non sécurisés

- Localisation : général
- Description : Le code ne force pas HTTPS, pas d'indication de sécurisation des cookies, pas de rate-limiting, pas de protection brute-force.
- Impact : interception de tokens/mots de passe, attaques par force brute, abuse de ressources.
- Correction proposée :
  - Forcer HTTPS en production (reverse proxy, HSTS). Utiliser cookies sécurisés si sessions.
  - Mettre en place rate-limiting (express-rate-limit) et mécanismes anti-brute-force.

Faille n°10 : Usage d'une base en mémoire pour la production

- Localisation : `server.js` (sqlite `:memory:`)
- Description : DB en mémoire perd les données au redémarrage et n'est pas adaptée pour multi-process.
- Impact : perte de données, problèmes de scalabilité.
- Correction proposée :
  - Utiliser une BD persistante (fichier SQLite sur disque ou un serveur DB dédié) avec backup et permissions restreintes.

Corrections définitives recommandées (plan d'action)

1. Sortir tous les secrets du code (variables d'environnement, gestionnaire de secrets).
2. Mettre en place hachage des mots de passe (bcrypt/argon2) et migration des comptes existants.
3. Refactorer toutes les requêtes SQL pour utiliser des requêtes préparées/paramétrées.
4. Implémenter une authentification robuste (JWT signé ou sessions serveur) et RBAC pour endpoints sensibles.
5. Remplacer les réponses HTML non-sanitized par des réponses JSON, ou utiliser un moteur de template sécurisé et échapper correctement.
6. Centraliser la gestion des erreurs : logger les erreurs internes, renvoyer des messages génériques au client.
7. Ajouter `helmet`, rate-limiting, CSP et autres headers de sécurité.
8. Ajouter validation d'entrées (Joi/express-validator) et vérifications strictes des types/longueurs.
9. Forcer HTTPS en production, configurer HSTS et sécuriser les cookies.

10. Mettre en place une procédure de tests (unitaires + tests d'intégration) pour vérifier les principales protections.

Exemple minimal de plan d'implémentation (priorités)

- Priorité haute (à déployer rapidement) : sortir les secrets, hachage des mots de passe, requêtes préparées, protection des endpoints admin, suppression des fuites d'erreur.
- Priorité moyenne : `helmet`, validation d'entrée, rate-limiting, logs d'audit.
- Priorité basse : migration DB vers solution persistante, rotation & gestion des secrets en prod.
