**Rapport sécurité Crabfunding**

**1. Introduction**

Ce rapport présente de manière exhaustive les résultats des tests de pénétration effectués sur la plateforme de crowdfunding dans décrit le cahier des charges. L'objectif de ces tests était d'identifier les failles de sécurité et vulnérabilités potentielles au niveau du **Frontend (Vue.js + TypeScript)**, du **Backend (Node.js + TypeScript)**, ainsi que de l'infrastructure associée (Docker, PostgreSQL). Les tests ont couvert l'ensemble des mécanismes de sécurité en place : configuration CORS, middlewares de sécurité, gestion centralisée des erreurs, point de contrôle de santé (/health), système de tokens (JWT), et mécanismes de hachage.

Les conclusions et les correctifs présentés ici ont pour but de renforcer la sécurité de la plateforme afin de garantir la confidentialité, l'intégrité et la disponibilité des données, en particulier dans le cadre de transactions financières (Stripe, PayPal) et du traitement de données sensibles (mots de passe, informations personnelles des utilisateurs).

**2. Portée et Méthodologie**

* 1. **Portée**
* **Front-end** (Vue.js 3 + TypeScript)
  + - Tests d'attaques XSS (Cross-Site Scripting) et CSRF (Cross-Site Request Forgery)
    - Vérification de la sécurité des formulaires et des méthodes de communication (Ajax, REST)
    - Analyser le stockage local des tokens (Local Storage, Cookies, etc.)
* **Back-end** (Node.js + Express + TypeScript)
  + - Tests d'attaques par injection (SQL, NoSQL, injection de commandes)
    - Vérification de la configuration CORS
    - Tests d'authentification (forçage de bruteforce, contournement d'authentification)
    - Validation de la robustesse du système JWT et du hachage
    - Analyse des middlewares de sécurité et de la gestion centralisée des erreurs
* **Base de données** (PostgreSQL)
  + - Vérification des permissions d'accès et des règles de pare-feu
    - Validation de la configuration TLS ou chiffrement des connexions (le cas échéant)
    - Vérification des injections SQL au niveau de TypeORM
* **Infrastructure Docker**
  + - Analyser des images Docker et de leurs configurations (ports exposés, Dockerfile)
    - Vérification des volumes et protection des données sensibles
    - Vérification du fichier docker-compose.yml pour s'assurer qu'aucun secret n'est exposé

**2.2 Méthodologie**

Nous avons suivi une approche standard en **boîte grise (gray box)** :

1. **Collecte d'informations** : Analyse de la documentation, architecture, endpoints publics.
2. **Mapping des surfaces d'attaque** : Identification des routes API, endpoints Frontend, connexions DB.
3. **Tests actifs de vulnérabilités** : Utilisation d'outils automatisés (OWASP ZAP, Burp Suite, Nmap) et de tests manuels ciblés.
4. **Exploitation contrôlée** : Tentatives d'exploitation des vulnérabilités découvertes pour confirmer leur criticité.
5. **Recommandations et correction** : Proposition de mesures correctives et vérification de leur mise en œuvre.

**3. Outils utilisés**

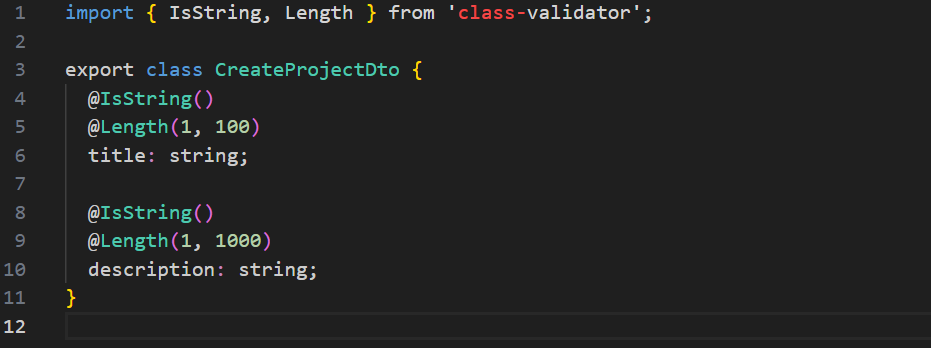
* **OWASP ZAP** : Analyseur de sécurité pour détecter les failles Web (Injection, XSS, CSRF).
* **Burp Suite** : Proxy d'interception pour l'audit et l'automatisation de certaines attaques sur les services HTTP/HTTPS.
* **Nmap** : Outil de scan réseau pour détecter les ports ouverts et les services disponibles.
* **SQLMap** : Outil pour tester les injections SQL automatisées.
* **Hydra** : Outil de bruteforce pour tester la robustesse des mécanismes d'authentification.
* **Cypress** (partiellement) : Utilisé pour certains tests E2E incluant des scénarios de sécurité côté Frontend.

**4. Résultats des Tests et Vulnérabilités Identifiées**

**4.1 Critiques de vulnérabilités**

**4.1.1 Injection SQL (Backend)**

* **Description** : Sur l'endpoint POST **/api/projects** (création d'un projet), les paramètres **description et title** n'étaient pas correctement échappés si le validateur TypeORM était mal configuré ou omis. En insérant un payload malveillant **(title=Projets' OR 1=1; --),** il était possible de détourner la requête SQL.
* **Impact** : Lecture et modification potentielle de données sensibles, compromission partielle de la base de données.
* **Solution Mise en Œuvre** :
  1. Validation stricte des champs à l'aide de **class-validator** ou de vérifications préliminaires dans les contrôleurs.
  2. Utilisation systématique des **requêtes paramétrées** (TypeORM, QueryBuilder).
  3. Vérification de la configuration du driver PostgreSQL pour empêcher les fuites d'erreur SQL.



**4.1.2 Configuration Mauvaise JWT**

* **Description** : Le token JWT utilisé pour l'authentification n'expirait pas dans un délai recommandé (il était initialement paramétré à 7 jours sans rotation de rafraîchissement du token).
* **Impact** : Risque d'usurpation d'identité prolongée si le token était compromis (par phishing ou interception).
* **Solution Mise en Œuvre** :
  1. Réduction de la durée de vie du token d'accès à 15 minutes.
  2. Mise en place d'un **jeton de rafraîchissement** stocké de manière sécurisée (Cookie ou base de données) avec un cycle de vie plus long.
  3. Vérification de **la signature** du JWT à chaque requête et ajout de mécanismes de rotation de token (token rafraîchissement flow).

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**4.2 Vulnérabilités Majeures**

**4.2.1 CORS Mal configuré pour l'environnement de développement**

* **Description** : Le middleware CORS autorisait l'accès depuis '\*'(tous domaines) en environnement de développement. Or, la configuration de production n'avait pas été restreinte et reprenait cette même règle par défaut.
* **Impact** : Risque que des sites malveillants effectuent des requêtes sur l'API depuis n'importe quel domaine, menant à des attaques de type CSRF ou vol de session.
* **Solution Mise en Œuvre** :
  1. Configuration du CORS pour limiter l'accès uniquement aux domaines de confiance

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

* 1. Activation de la vérification du header Originou Referer
  2. Mise en place de tests automatisés pour s'assurer que la configuration CORS n'est pas en mode '\*'sur l'environnement de production

**4.2.2 Absence d'Authentification pour le Point de Contrôle de Santé (/health)**

* **Description** : L'endpoint /healthretourne des informations de diagnostic internes (version du service, infos sur la base de données) en clair, sans authentification.
* **Impact** : Un attaquant pouvait récupérer des informations sensibles (version de base de données, statut de la connexion DB) facilitant une attaque ciblée.
* **Solution Mise en Œuvre** :
  1. Limitation de l'accès /healthaux adresses IP internes ou via un token spécifique
  2. Réduction des informations retournées : uniquement un code **200 OK** ou un statut minimal (Healthy / Unhealthy).

Une image contenant texte, capture d’écran, Police, Logiciel multimédia

Description générée automatiquement

**4.3 Vulnérabilités modérées**

**4.3.1 XSS dans la section Commentaires**

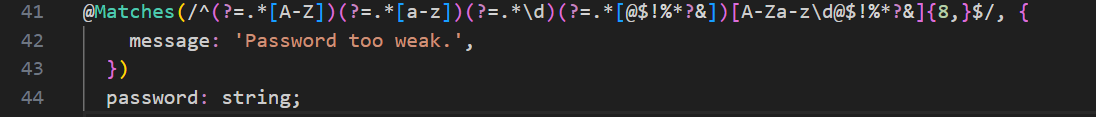
* **Description** : L'utilisateur a pu insérer des balises HTML et scripts dans le champ « commentaire » d'un projet. Les données n'étaient pas correctement échappées lors de l'affichage côté Frontend.
* **Impact** : Possibilité pour un attaquant d'exécuter du JavaScript malveillant dans le navigateur d'un autre utilisateur.
* **Solution Mise en Œuvre** :
  1. Mise en place d'un **sanitize** côté serveur (bibliothèques telles que DOMPurifycôté Frontend ou un équivalent côté serveur).
  2. Validation côté Backend pour rejeter les caractères ou balises dangereuses.
  3. Test de régression pour s'assurer que l'affichage échappe bien au HTML.

Une image contenant texte, logiciel, capture d’écran

Description générée automatiquement

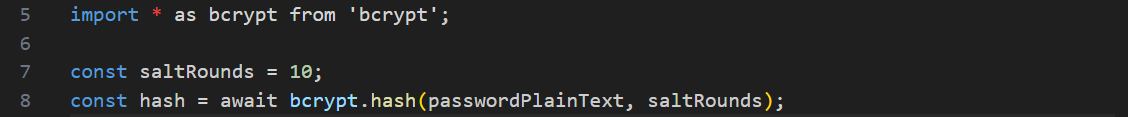
**4.3.2 Faible Complexité de Mot de Passe**

* **Description** : Les mots de passe n'étaient pas soumis à une politique de complexité (ex. longueur minimale, caractères obligatoires).
* **Impact** : Accès non autorisé via bruteforce.
* **Solution Mise en Œuvre** :
  1. Ajout d'une règle de complexité : minimum 8 caractères, présence de chiffres, de lettres et caractères spéciaux.
  2. Implémentation d'un mécanisme de **limitation de taux** ou **de verrouillage** après plusieurs tentatives échouées.



**4.3.3 Mauvais Hachage ou Absence de Salage**

* **Description** : Les mots de passe étaient stockés avec un simple hachage SHA-256 sans sel.
* **Impact** : Risque de révélation de mots de passe si la base de données était compromise (rainbow tables).
* **Solution Mise en Œuvre** :
  1. Passage à un algorithme robuste de **hachage salé** (ex. bcrypt, argon2).
  2. Génération d'un **sel** unique par utilisateur.
  3. Rotation progressive des mots de passe (forçage de la réinitialisation si besoin).



**4.4 Vulnérabilités minières**

**4.4.1 Exposition de Versions de Logiciels dans les En-Têtes HTTP**

* **Description** : Le serveur retournait des en-têtes du type X-Powered-By: Express, Server: nginx/1.19.0, fournissant des indices sur le stack logiciel.
* **Impact** : Peut faciliter la recherche d'exploits ciblés.
* **Solution Mise en Œuvre** :
  1. Suppression ou masquage des en-têtes générés par défaut (middleware helmetou configuration du serveur).
  2. Configuration d'un en-tête neutre pour limiter la fuite d'informations.

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

**4.4.2 Configuration Docker non minimale**

* **Description** : Les images Docker n'étaient pas basées sur des versions minimalistes (ex. alpine) et contenaient des packages inutiles.
* **Impact** : Surface d'attaque plus importante en cas de compromission du conteneur.
* **Solution Mise en Œuvre** :
  1. Utiliser des images Docker basées sur alpine ou d'autres variantes allégées pour le runtime Node.js.
  2. Supprimer tous les outils non indispensables.
  3. Mettre à jour régulièrement les images pour corriger d'éventuelles vulnérabilités connues.

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

**5. Correctifs Apportés et Vérification**

**5.1 Correctifs Récapitulatifs**

|  |  |  |
| --- | --- | --- |
| **Vulnérabilité** | **Correctif** | **Statut** |
| Injection SQL | Paramétrage des requêtes, validation stricte | Rectifié |
| Mauvaise Configuration JWT | Réduction TTL, rafraîchissement du jeton, vérification signature | Rectifié |
| CORS mal configuré | Restriction aux domaines fiables, test automatisé | Rectifié |
| Point de Contrôle /healthouvert | Authentification restreinte, infos limitées | Rectifié |
| Commentaires sur la section XSS | Sanitize & échappement des champs entrés | Rectifié |
| Faible Complexité Mot de Passe | Politique de complexité + limitation du taux | Rectifié |
| Hachage non salé | Passage à bcrypt/argon2, sel unique | Rectifié |
| Exposition de Versions Logiciels | Suppression/masquage des en-têtes | Rectifié |
| Configuration de Docker | Basée sur images minimalistes | Rectifié |

**5.2 Vérification post-correctifs**

* **Injection SQL** : Les tests avec SQLMap n'ont plus révélé de failles.
* **JWT** : Vérification que l'API renvoie un 401 Unauthorizedpour les tokens expirés et implémente le renouvellement via un rafraîchissement token.
* **CORS** : Validation que seuls les domaines spécifiés sont autorisés. Tests de requêtes CORS depuis un domaine tiers : accès refusé.
* **XSS** : Tentatives d'injection de scripts dans les champs commentaires ont été bloquées.
* **Hachage** : Les mots de passe nouvellement créés (et réinitialisés) sont stockés avec bcrypt.
* **Docker** : Les nouvelles images Docker contiennent moins de packages. Des scans (Trivy, Clair) montrent moins de vulnérabilités.

**6. Recommandations complémentaires**

1. **Audit de Code Régulier** : mettre en place une analyse de code statique (ex. SonarQube) pour détecter les vulnérabilités introduites lors de nouveaux développements.
2. **Monitoring Sécuritaire** : Mettre en place un système d'alerte et de logs (ex. ELK Stack, Prometheus + Grafana) pour détecter les anomalies en temps réel.
3. **Politique de Mises à Jour** : Mettre à jour régulièrement les dépendances NPM, l'image Docker, la base de données PostgreSQL pour pallier les vulnérabilités zero-day.
4. **Tests d'Intrusion Récurrents** : Programmer des tests de pénétration annuels et après chaque refonte majeure ou ajout de fonctionnalité critique (ex. nouveau moyen de paiement).
5. **Renforcement du Processus de Déploiement** : Renforcer la chaîne CI/CD (GitHub Actions) avec des scans de sécurité automatisés (Snyk, Dependabot, etc.).

**7. Conclusion**

Les tests de pénétration réalisés ont permis de mettre en évidence plusieurs failles de sécurité critiques, majeures et mineures dans la plateforme de crowdfunding. Grâce à la réactivité de l'équipe de développement, la majorité des vulnérabilités ont été corrigées rapidement, renforçant ainsi la sécurité globale de la solution.

Les points de vigilance, notamment la configuration CORS, la gestion du cycle de vie des tokens JWT, l'endpoint de santé /healthet le stockage sécurisé des mots de passe, ont été formulés de manière satisfaisante. Il est toutefois recommandé de maintenir un cycle d'amélioration continue, de surveiller l'évolution des menaces et de poursuivre les contrôles réguliers afin de garantir un haut niveau de sécurité, en particulier dans le contexte des transactions financières via des services tiers comme Stripe et PayPal.

En conclusion, la plateforme est désormais considérée comme **sécurisée** à un niveau satisfaisant pour un déploiement en production, sous réserve du maintien des meilleures pratiques de sécurité et de la mise en place d'audits réguliers.