HE2B ESI

SECG4 - Sécurité

Projet de chat sécurisé

Auteurs :
Dylan Bricar (54027)
Jeremie Seshie (54627)

Professeur :
Moussa WAHID

Groupe :
D122

Année académique :
2020 - 2021

Table des matières

Introduction

Le but est de créer un chat dans lequel les utilisateurs peuvent communiquer de manière privée et où les informations sont sécurisées. Ce projet comprend l'inscription, la connexion des utilisateurs ainsi qu'une gestion des contacts.

Informations complémentaires

De nombreux logiciels existent afin de créer du code en ligne, dans ce projet le choix s'est porté sur un framework extrêmement connu : " Laravel ".
Ce dernier jouit d'une très grande communauté remplie de développeur. De ce fait, plusieurs optimisations sont déjà présentes et facilitent l'intégration des sécurités. De plus si une faille est trouvée elle est très rapidement corrigée et il suffira de se tenir au courant pour avoir une base de site solide.
Ce dernier se base sur PHP et requiert donc un serveur ainsi qu'une base de données pour la persistance de donnée. De plus, afin de correctement modifier un projet Laravel, il est nécessaire d'avoir installé Composer et NodeJS.

Informations sur la sécurité

- Utilisation d'un compte propre au chat dans la BDD. Cela permet de se protéger s'il y a une faille au niveau du chat, en effet les modifications/pertes de données faites avec cet utilisateur n'impacteront pas le reste de l'application.
- Le compte associé au chat n'a accès qu'aux requêtes SELECT, UPDATE, DELETE et INSERT afin de ne pas pouvoir causer des dégâts importants.
- La version de débogage est désactivée dans le fichier « .env » ce qui protège davantage les failles si des erreurs apparaissent (rejoins les améliorations citées ci-dessous au niveau du serveur).
- Les mots de passe sont cryptés à l'aide de la fonction " password_hash() " (compatible bcrypt). A l'heure actuelle, c'est ce qui est défini comme la référence de cryptage pour les mots de passe sur le web. Chaque mot de passe est crypté différemment même si les mots sont identiques. Il y a un salt et avec l'options « rounds » à 12, il y a volontairement une boucle d'hachage de 12 fois. C'est un algorithme qui est volontairement plus lent afin d'impacter fortement la « rainbow attack » entre autres.
- Les messages sont cryptés avant l'insertion dans le serveur et décryptés au client à l'aide de plusieurs HASH directement dans SQL, cela permet une prévention en cas de leak de la base de données et aucun utilisateur n'a accès à cette clé de chiffrement/déchiffrement (tout se passe du côté serveur).
- Etant donné le choix d'un framework récent dont les mises à jour sont effectuées et récentes, toutes les failles découvertes à l'heure actuelle sont corrigées. Il est nécessaire de se tenir à jour des avancées de Laravel afin d'être toujours à jour d'un point de vue sécurité.
- La liste des failles les plus courantes sont vérifiées et sécurisées (OWASP) : Injection, authentification interrompue, exposition des données sensibles, entités externes

XML, contrôle d'accès interrompu, configuration incorrecte de la sécurité, Cross-Site Scripting XSS, désérialisation non sécurisée, utilisation de composants avec des vulnérabilités connues, journalisation et surveillance insuffisantes.

- Après 5 tentatives de connexion / inscription, l'utilisateur est bloqué pour 1 minute. Cela permet d'éviter les requêtes trop rapides et le brute force.
- Les ID des utilisateurs sont des UUID, cela permet d'éviter qu'un utilisateur modifie l'URL pour tenter de connaître l'ID d'un autre destinataire (même si des vérifications sont effectuées).
- Un captcha est proposé à l'inscription et à la connexion afin d'éviter les inscriptions et tentatives de connexion par des robots.
- Tous les formulaires possèdent un tag spécifique à Laravel : @CSRF afin de protéger contre les failles CSRF qui permettent à un utilisateur de faire exécuter une action qui n'était pas prévue par l'utilisateur de base.
- Les mots de passe indiqués doivent avoir au moins 8 caractères et respecter le format (majuscule, minuscule, caractère spécial et nombre) afin de complexifier les attaques de récupération de mot de passe.
- Les actions des utilisateurs sont enregistrées dans les dossiers de Laravel : « /storage/logs » et sont configurables dans le serveur (voir plus bas). Il est important de pouvoir voir toutes les actions d'un utilisateur pour détecter une tentative d'intrusion ou en cas de problème comprendre d'où cela provient afin de régler rapidement le souci.
- Aucun risque d'injection SQL car toutes les requêtes sont préparées automatiquement lors de l'utilisation des méthodes de l'interface DB.
- Aucun risque de faille XSS puisque les utilisateurs ne peuvent pas entrer de code et que tous les codes sont échappés automatiquement lors de l'affichage (mais voir configuration du serveur plus bas).
- Tous les formulaires sont vérifiés du côté serveur également afin de bloquer un utilisateur qui modifie les données avec des informations auquel il n'a pas accès afin de créer une action non autorisée.
- Les API sont sécurisées et des vérifications d'authenticités sont effectuées par rapport aux connexions. Un utilisateur A ne pourra pas accéder aux messages d'un utilisateur B même en connaissant le lien vers lequel pointe l'API.
- Le contenu du code chargé par Javascript dans les messages est inséré à travers la méthode « *text* » afin d'empêcher l'insertion d'HTML du côté client.

Même si les configurations liées au code sont terminées, plusieurs configurations sont à faire pour optimiser grandement la sécurité du site internet afin qu'il soit viable en production :

- Après déploiement sur un nom de domaine, il faudrait un certificat SSL afin de sécuriser le transfert d'informations entre le client et le serveur.
- Avec le certificat SSL, tous les cookies pourraient passer en HTTPS-ONLY et seront plus sécurisés que les cookies simples.
- Avec un nom de domaine, il serait possible de lier Cloudflare afin de limiter les attaques DdoS, de bot, un captcha etc.
- Installer sur le serveur Fail2ban pour traiter les attaques réseau.

- Il faudrait ajouter des règles iptables dans le fichier « **installation/firewall.sh** » sur le serveur afin de limiter le ping, les attaques DdoS et bannir temporairement ceux qui effectuent trop de requêtes.
- Il faudrait bloquer les requêtes trop nombreuses d'un même utilisateur (évite le brute force ou une surcharge du serveur) avec une configuration du côté PHP + MySQL + Firewall.
- Il faudrait configurer PHP pour n'afficher aucune erreur au client (afficher les erreurs permet de mieux comprendre l'architecture du code) ainsi que de n'autoriser que les cookies provenant du serveur courant.
- Il faudrait activer les logs du serveur (NGINX/APACHE), PHP ainsi que MySQL pour voir toutes les requêtes effectuées et déceler s'il y a un problème ou des tentatives d'intrusion.
- Il faudrait ajouter dans la configuration du serveur (NGINX dans ce cas) des règles bloquant les failles XSS, sécurisant les COOKIES et validant l'authentification du serveur (voir fichier « **installation/xss.txt** »).

Conclusion générale

Les données sont sécurisées chez le client comme du côté serveur, cependant pour sécuriser la totalité du système il est nécessaire de passer aussi par une sécurisation du côté serveur.

Le framework est une aide très importante, mais il est nécessaire de régulièrement se tenir au courant des mises à jour puisqu'une faille découverte sur un framework peut très rapidement être exportée sur les autres sites qui utilisent ce même framework.

Il est impossible d'être totalement sécurisé de toutes les failles (DdoS par exemple) mais il est possible de limiter leur impacte et d'augmenter considérablement la difficulté des hackers (mettre un système d'anti-brute force avec iptables ou Cloudflare augmente énormément le temps de calcul nécessaire pour tester tous les mots de passe).

Ce rapport a été écrit par Dylan BRICAR et Jeremie SESHIE dans le cadre du cours de laboratoire de sécurité donné par la Haute École Bruxelles-Brabant - HE2B ESI et enseigné par Moussa WAHID.