

HEIG-VD

Modélisation de menaces

STI – Projet 2

Gabriel Roch & Cassandre Wojciechowski

22/01/2021

Table des matières

1 Introduction.....	2
2 Description du système	3
2.1 Data-Flow Diagram	3
2.2 Identification des éléments du système	4
2.3 Définition du périmètre de sécurisation	4
3 Identification des sources de menaces	4
3.1 Eléments du système attaqué	4
3.2 Motivation(s)	4
3.3 Scénarios d'attaque.....	5
3.3.1 Suppression de la base de données (injection SQL).....	5
3.3.2 Réalisation d'actions en tant qu'administrateur (XSS / injection HTML)	5
3.3.3 Réalisation d'actions en tant qu'administrateur (CSRF).....	6
3.3.4 Usurpation d'identité	7
3.3.5 Infection des systèmes des utilisateurs.....	8
3.3.6 Distributed Denial of Service	9
3.3.7 Distributed Denial of Service v.2	9
3.3.8 Bruteforce sur les mots de passe	9
3.4 STRIDE.....	10
4 Identifier les contre-mesures	11
5 Sécurisation de l'application	12
6 Conclusion	13

1 Introduction

Ce rapport contient une modélisation de menaces réalisée pour une application de messagerie Web implémentée dans le cadre du cours STI (Sécurité des Technologies Internet) lors de notre cinquième semestre d'étude à la HEIG-VD.

Le but de ce document est de déterminer les objectifs de sécurité que nous souhaitons atteindre dans l'implémentation de notre application et, par conséquent, d'identifier les menaces auxquelles l'application de messagerie Web est exposée. Pour ce faire, nous allons détailler le fonctionnement de notre application, les différentes parties qui la composent et les utilisations qui peuvent s'en faire.

Le code source analysé et modifié est disponible sur le repo Github : <https://github.com/g-roch/heig-sti-lab01>.

Pour rappel, l'application de messagerie Web que nous avons implémentée en PHP lors du premier projet STI devait comporter les fonctionnalités suivantes :

Un collaborateur aura accès aux fonctions suivantes :

- *Lecture des messages reçus : une liste, triée par date de réception, affichera les informations suivantes :*
 - *Date de réception*
 - *Expéditeur*
 - *Sujet*
 - *Bouton ou lien permettant la réponse au message*
 - *Bouton ou lien permettant la suppression du message*
 - *Bouton ou lien permettant d'ouvrir les détails du message*
 - *Devra permettre l'affichage des mêmes informations/options que ci-dessus, avec le corps du message en plus*
- *Ecrire un nouveau message : rédaction d'un nouveau message à l'attention d'un autre utilisateur. Les informations suivantes devront être fournies :*
 - *Destinataire (unique)*
 - *Sujet*
 - *Corps du message*
- *Changement du mot de passe : afin de pouvoir modifier son propre mot de passe*

Un administrateur aura accès aux fonctions suivantes :

- *Doit avoir les mêmes fonctionnalités qu'un Collaborateur, en plus des suivantes*
- *Ajout / Modification / Suppression d'un utilisateur : un utilisateur est représenté par :*
 - *Un login (non modifiable)*
 - *Un mot de passe (modifiable)*
 - *Une validité (booléenne, modifiable), actif ou inactif*
 - *Un rôle (modifiable)*

Après avoir spécifié les menaces que nous nous attendons à rencontrer, nous allons estimer l'exposition de notre application à celles-ci en imaginant plusieurs scénarios d'attaque.

Ce rapport a pour but final de réduire les risques encourus par notre système en trouvant des solutions sécurisées et en mettant en place des contre-mesures.

2 Description du système

Notre système (application de messagerie Web) a plusieurs objectifs : tout d'abord elle permet aux utilisateurs inscrits dans l'application de s'échanger des messages qui doivent rester privés. Notre système doit également rester disponible ~99% du temps. Nous comptons donc sur sa disponibilité et sa confidentialité pour garder une bonne réputation auprès de nos utilisateurs et futurs utilisateurs.

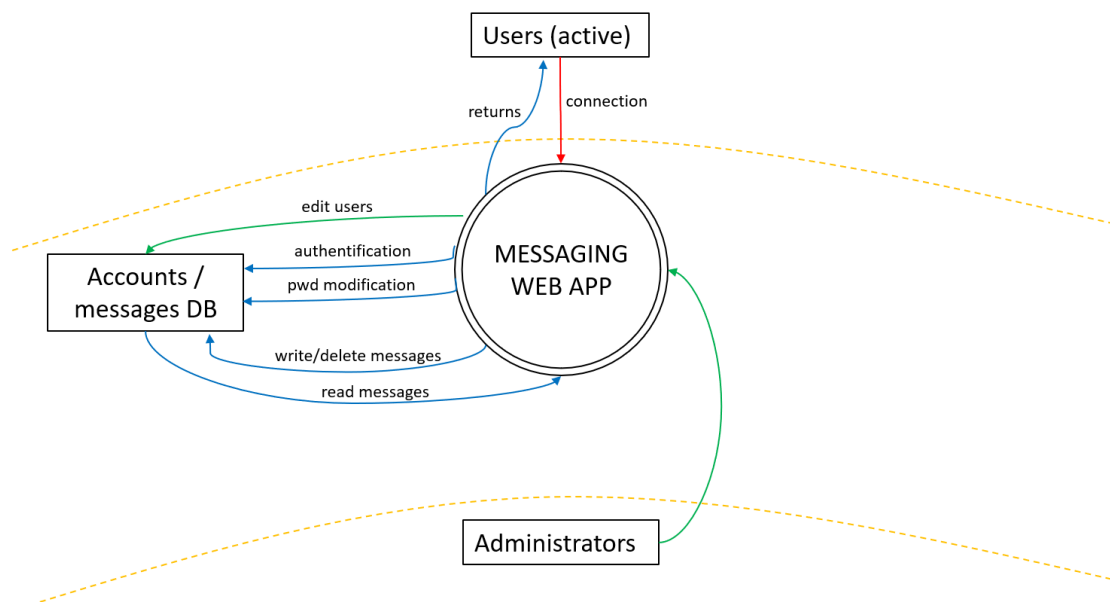
Nous posons les hypothèses de sécurité suivantes : les administrateurs actifs sur notre application sont de confiance et n'entreprennent pas d'action visant à nuire à nos utilisateurs ou à notre système ; le serveur Web sur lequel l'application est hébergée est de confiance et ne possède pas de faille exploitable.

Nous posons les exigences de sécurité suivantes :

- Seuls les utilisateurs inscrits et actifs peuvent se connecter et envoyer des messages (Contrôles d'accès)
- Les utilisateurs ne doivent pas accéder aux fonctions d'administrateur (Contrôles d'accès)
- Les messages et les données personnelles ne doivent être modifiés que par l'utilisateur concerné (Intégrité)
- L'application doit être disponible 99% du temps pour ses utilisateurs (Disponibilité)
- Les messages et les informations personnelles des utilisateurs doivent être protégés (Sphère privée)

Notre système comprend deux sortes d'utilisateurs : les utilisateurs normaux, qui se connectent et échangent des messages, et les administrateurs, qui ont le droit de modifier la base de données des utilisateurs en plus.

2.1 Data-Flow Diagram



Dans le diagramme ci-dessus, nous voyons que les utilisateurs doivent impérativement se connecter à l'application (flèche rouge), sans quoi aucune action ne peut être entreprise. Une fois connectés et authentifiés, les utilisateurs peuvent modifier leur mot de passe et/ou lire et écrire des messages.

L'administrateur peut effectuer toutes ces actions et il peut en plus modifier la base de données des utilisateurs.

2.2 Identification des éléments du système

Notre système compte les éléments suivants :

- Une base de données contenant les utilisateurs, leurs données personnelles et leurs messages envoyés / reçus (ce sont donc des données)
 - o Cela touche à la confidentialité / sphère privée des utilisateurs
 - o En cas de problème, la réputation de l'application est mise en jeu car les utilisateurs n'apprécieraient pas que leurs données et leurs messages soient rendus publics.
- Une application Web (c'est une infrastructure)
 - o Cela touche à l'intégrité et la disponibilité du système
 - o En cas de problème, il est possible de nuire à la disponibilité et/ou à la réputation (les utilisateurs ne voudront pas utiliser une application qui n'est pas disponible ou qui donne accès à leurs informations personnelles)

2.3 Définition du périmètre de sécurisation

Ce qu'il faut absolument sécuriser dans le cadre de ce projet, c'est donc l'application PHP ainsi que la base de données SQLite qui lui est reliée. Pour cela, nous devons sécuriser les interactions entre l'application et la base de données pour éviter de faire fuiter des informations.

3 Identification des sources de menaces

3.1 Éléments du système attaqué

Les cibles potentielles qu'un attaquant pourraient avoir sont donc la bases de données contenant les messages et les données personnelles des utilisateurs ainsi que l'application Web elle-même.

3.2 Motivation(s)

Les motivations que l'on pourrait prêter à un attaquant qui s'en prend à notre application de messagerie Web seraient :

- Lecture des messages privés d'utilisateurs de la messagerie
- Envoi de messages en usurpant l'identité d'autres utilisateurs
- Récupération des informations personnelles des utilisateurs (mots de passe, ...)
- Empêcher l'utilisation de l'application (Dos, DDos)
- S'amuser
- Prise de contrôle, pouvoir, ego

3.3 Scénarios d'attaque

3.3.1 Suppression de la base de données (injection SQL)

- Impact métier : moyen à élevé (en fonction de l'utilisation de l'outil par l'entreprise)
- Source de la menace : employé mécontent, sur le point d'être licencié, espion pour la concurrence
- Niveau de connaissance : script kiddies
- Motivation : vengeance, causer des dommages à un concurrent
- Ressource(s) ciblée(s) : base de données
- Scénario d'attaque :
 - Utilisation d'un outil tel SQLMap pour obtenir un accès à la base de données
 - Suppression de la base de données en envoyant un « DROP DATABASE »
- Contre-mesures :
 - Echapper toutes les chaînes de caractères avant l'envoi à la base de données via des requêtes préparées

Pour effectuer une attaque de ce genre, on peut donc faire une injection SQL dans l'URL, comme ci-dessous, où nous supprimons tous les messages contenus dans la base de données :



http://127.0.0.2:8080/mailbox.php?delete=2 OR 1 = 1

Pour corriger cette faille, une bonne pratique est d'utiliser des requêtes préparées comme suit (on voit toujours la ligne de code contenant la faille commentée en haut de la capture) :

```
// $statement = $pdo->query("DELETE FROM `messages` WHERE `id` = $_GET[delete]");  
$statement = $pdo->prepare( query: "DELETE FROM `messages` WHERE `id` = ?");  
$statement->execute([$_GET["delete"]]);
```

3.3.2 Réalisation d'actions en tant qu'administrateur (XSS / injection HTML)

- Impact métier : élevé (exfiltration de messages sensibles possible)
- Source de la menace : utilisateur malicieux avec quelques connaissances de hacking
- Niveau de connaissance : hacker débutant
- Motivation : expérimentation, amusement
- Ressource(s) ciblée(s) : compte d'autrui, surtout un compte administrateur
- Scénario d'attaque :
 - Envoi d'un e-mail avec une injection HTML dans le corps du message
 - Récupérer l'ensemble des messages reçus de la cible
 - S'octroyer les droits d'administration (si la cible est administrateur, l'attaquant peut obtenir ses droits, créer/modifier un utilisateur et donc, se promouvoir en tant qu'administrateur)
- Contre-mesures :
 - Echapper les caractères dans l'affichage du corps de l'e-mail

Il est possible d'envoyer un message contenant dans le corps un script HTML qui exécutera des actions que le destinataire ne contrôle pas. Pour montrer le fonctionnement de cette attaque, nous avons créé le script *hacker.js* :

```
var xhr = new XMLHttpRequest();
xhr.open("POST", '/users.php', true);

//Envoie les informations du header adaptées avec la requête
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

xhr.onreadystatechange = function() { //Appelle une fonction au changement d'état.
    if (this.readyState === XMLHttpRequest.DONE && this.status === 200) {
    // Requête finie, traitement ici.
    }
}
xhr.send("2[password]=&2[confirm-password]=&2[isadmin]=true&2[isactive]=true");
```

Nous l'avons ensuite inclus dans le corps du message comme sur la capture d'écran suivante :

To	Subject
<input type="text" value="alice"/>	<input type="text" value="123"/>
Body	
<input 127.0.0.1="" hacker.js"><="" script>"="" type="text" value="<script src="/>	

Cela nous a permis de promouvoir l'utilisateur qui a envoyé le message en administrateur. Pour corriger cela, il faut ajouter la fonction *htmlentities* pour récupérer le corps du message et échapper tous les caractères qui permettent d'écrire des instructions en HTML.

```
<div id="body---><?/= htmlentities($message['id']) ?><!--" class="sr-only">--><?/= ($message['body']); ?><!--
<div id="body-<?= htmlentities($message['id']) ?>" class="sr-only"><?= htmlentities($message['body']); ?></div>
```

3.3.3 Réalisation d'actions en tant qu'administrateur (CSRF)

- Impact métier : moyen (modification de la base de données)
- Source de la menace : attaquant externe avec des connaissances et quelques moyens
- Niveau de connaissance : hacker expérimenté
- Motivation : récupération de données
- Ressource(s) ciblée(s) : comptes d'utilisateur et/ou administrateur, base de données
- Scénario d'attaque :
 - Transmission d'un lien vers un site web malicieux à un utilisateur de l'application de messagerie
 - Espoir que la personne clique sur le lien dans une fenêtre du navigateur dans laquelle l'application est déjà ouverte (ou qu'au moins une personne le fasse)
 - Suppression/ajout d'un utilisateur, suppression de message(s)
 - Possibilité de faire une injection SQL lors de la suppression d'un utilisateur/message
 - Si l'utilisateur fait un clic sur le site malicieux, toutes les autres actions sont disponibles (modification du mot de passe, envoi d'un message, création/modification d'un utilisateur)
- Contre-mesures :
 - Mettre un *token* CSRF sur toutes les actions réalisables

Pour effectuer cette attaque-là, nous avons créé une page *hacker.html* contenant un bouton cliquable redirigeant sur une autre page en effectuant des actions « POST ». Le code ci-dessous permet de modifier le statut de l'utilisateur et de le promouvoir comme administrateur :

```
<form method=post action="//127.0.0.1:8080/users.php">
  <input type=hidden name="2[password]" value="" />
  <input type=hidden name="2[confirm-password]" value="" />
  <input type=hidden name="2[isadmin]" value="true" />
  <input type=hidden name="2[isactive]" value="true" />
  <button action=submit>send</button>
</form>
```

Après que le bouton ait été utilisé, notre utilisateur est devenu administrateur. Pour corriger cette faille, nous avons ajouté un *token* CSRF dans le fichier *users.php* (première capture) qui sera vérifié lors des actions demandées par l'utilisateur (seconde capture) :

```
<?php
// New token
$_SESSION['token'] = bin2hex(openssl_random_pseudo_bytes( length: 32));
?>
<input type="hidden" name="csrf" value="<?> htmlentities($_SESSION['token']) ?>" />

// Vérification du token CSRF
if (isset($_POST['csrf'], $_SESSION['token']))
    if ($_SESSION['token'] !== $_POST['csrf']) {
        // New token
        $_SESSION['token'] = bin2hex(openssl_random_pseudo_bytes( length: 32));
    } else {

        //Traitement des modifications demandés par l'utilisateur
        foreach ($_POST as $id => $value) {
```

3.3.4 Usurpation d'identité

- Impact métier : élevé (selon l'identité de la cible)
- Source de la menace : attaquant avec des connaissances
- Niveau de connaissance : hacker expérimenté
- Motivation : récupération de données
- Ressource(s) ciblée(s) : comptes d'utilisateur et/ou administrateur, base de données
- Scénario d'attaque :
 - Création d'un compte grâce à une des attaques citées précédemment
 - Création d'un nom d'utilisateur similaire à un autre préexistant avec un caractère ne s'affichant pas (par exemple : le caractère Unicode, 'U+AD' trait d'union conditionnel)
 - Cela permet d'obtenir un compte qui n'est pas différenciable du compte légitime par les utilisateurs
- Contre-mesures :
 - Patcher les vulnérabilités utilisées plus haut pour créer un compte
 - Ajouter un contrôle manuel sur les créations de comptes
 - Créer un système avec une catégorie « Contacts » pour identifier plus rapidement l'émetteur d'un message

Sur la première capture ci-dessous, nous avons créé un deuxième utilisateur « qwuade » mais le caractère Unicode 'U+AD' ne s'affiche pas à l'écran et ne nous permet pas de distinguer les deux utilisateurs « qwe » :

Users list

#	Username	Password
1	alice	...
2	qwe	...
4	qwe	...

Ensuite, chaque utilisateur « qwe » a envoyé un message à « alice » mais nous constatons que la différence entre les noms d'utilisateurs n'est pas visible :

From	Subject	Date	Action
qwe	I am hacker	Friday, January 15th 2021 13:50:13	view reply delete
qwe	I am qwe	Friday, January 15th 2021 13:50:03	view reply delete

Etant donné que les failles citées précédemment ont été corrigées, il n'est plus possible de faire des injections SQL ou HTML ou encore des attaques CSRF, mais il est toujours possible d'usurper l'identité d'un autre utilisateur.

3.3.5 Infection des systèmes des utilisateurs

- Impact métier : très élevé (attaquant accède à la machine de la cible)
- Source de la menace : attaquant avec quelques connaissances et un compte utilisateur
- Niveau de connaissance : hacker débutant
- Motivation : prise de contrôle du poste
- Ressource(s) ciblée(s) : machine de la cible
- Scénario d'attaque :
 - Injection de code JavaScript dans le corps d'un message téléchargeant automatiquement un malware
 - Espoir que l'utilisateur lance le malware
- Contre-mesures :
 - Empêcher l'injection HTML/JavaScript
 - Dans le cas où les utilisateurs peuvent s'envoyer des liens/fichiers (ce qui n'est pas le cas actuellement), mettre en place un système de scan anti-virus

3.3.6 Distributed Denial of Service

- Impact métier : moyen à élevé (indisponibilité du système, ainsi que les autres services tournant sur le même serveur)
- Source de la menace : attaquant avec quelques connaissances et des moyens
- Niveau de connaissance : hacker débutant
- Motivation : empêcher l'utilisation du système
- Ressource(s) ciblée(s) : le système
- Scénario d'attaque :
 - L'attaquant achète un réseau de zombies et déclenche une attaque sur l'adresse du service
- Contre-mesures :
 - Mettre en place des systèmes anti-ddos
 - Eviter d'exposer le service sur Internet si ce n'est pas nécessaire

3.3.7 Distributed Denial of Service v.2

- Impact métier : moyen à élevé (indisponibilité du système, ainsi que les autres services tournant sur le même serveur)
- Source de la menace : attaquant avec quelques connaissances
- Niveau de connaissance : hacker débutant
- Motivation : empêcher l'utilisation du système
- Ressource(s) ciblée(s) : le système
- Scénario d'attaque :
 - L'attaquant envoie un message à tous les utilisateurs avec une injection HTML/JavaScript dans le corps
 - Le script injecté déclenche des requêtes en boucle sur le serveur
- Contre-mesures :
 - Empêcher l'injection HTML

3.3.8 Bruteforce sur les mots de passe

- Impact métier : moyen
- Source de la menace : attaquant, concurrent
- Niveau de connaissance : script kiddies
- Motivation : donner des ordres non-souhaités par l'entreprise
- Ressource(s) ciblée(s) : comptes utilisateurs
- Scénario d'attaque :
 - L'attaquant repère un nom d'utilisateur existant sur l'application
 - Il se renseigne sur la personne possédant ce compte (date de naissance, animaux de compagnie, enfants, ...)
 - Il essaie de se connecter au compte avec les mots de passe générés avec un dictionnaire et les informations obtenues précédemment
- Contre-mesures :
 - Limiter le nombre de tentatives de connexion erronées (IP et utilisateur)
 - Imposer un mot de passe fort pour tous les utilisateurs
 - Authentification à double facteur pour les administrateurs

Pour limiter le nombre de tentative de connexion, il faut ajouter un compteur dans la base de données ainsi qu'une date de dernier échec de connexion pour chaque utilisateur. A chaque tentative de connexion qui échoue, le compteur est incrémenté et la date est mise à jour. Quand il arrive au maximum autorisé (trois, par exemple), toutes les connexions sont refusées tant qu'on est dans les x minutes après le dernier échec.

Il est recommandé d'implémenter le même système de vérification pour les adresses IP (IPv4 et IPv6 si possible).

3.4 STRIDE

Spoofing :

- Scénario [3.3.8 Bruteforce sur les mots de passe](#)

Tampering :

- Scénario [3.3.1 Suppression de la base de données \(injection SQL\)](#)
- Scénario [3.3.5 Infection des systèmes des utilisateurs](#)
 - Les modifications se font sur l'OS de l'utilisateur infecté

Repudiation :

- Scénario [3.3.4 Usurpation d'identité](#)

Information disclosure :

- Scénario [3.3.2 Réalisation d'actions en tant qu'administrateur \(XSS / injection HTML\)](#)

Denial of service :

- Scénario [3.3.6 Distributed Denial of Service](#)
- Scénario [3.3.7 Distributed Denial of Service v.2](#)

Elevation of privileges :

- Scénario [3.3.2 Réalisation d'actions en tant qu'administrateur \(XSS / injection HTML\)](#)
- Scénario [3.3.3 Réalisation d'actions en tant qu'administrateur \(CSRF\)](#)

4 Identifier les contre-mesures

Scénario [3.3.1 Suppression de la base de données \(injection SQL\)](#)

Pour éviter les injections SQL, il faut échapper toutes les chaînes de caractères avant l'envoi à la base de données. Il faut toujours contrôler les informations entrées par l'utilisateur.

Scénario [3.3.2 Réalisation d'actions en tant qu'administrateur \(XSS / injection HTML\)](#)

Pour éviter des failles de Cross-Site Scripting (XSS) et les injections HTML, il faut échapper les chaînes de caractères dans l'affichage du corps de l'e-mail.

Scénario [3.3.3 Réalisation d'actions en tant qu'administrateur \(CSRF\)](#)

Il faut mettre un *token* CSRF sur toutes les actions réalisables.

Scénario [3.3.4 Usurpation d'identité](#)

Pour éviter une attaque de type usurpation d'identité, il faut tout d'abord patcher les vulnérabilités utilisées précédemment pour créer un compte. Il serait également possible d'ajouter un contrôle manuel sur les créations de compte, qu'un employé soit responsable de juger si l'utilisateur souhaitant créer un compte peut le faire. Il faudrait également créer un système avec une catégorie « Contacts » pour identifier plus rapidement l'émetteur d'un message et juger facilement si on le connaît ou non.

Scénario [3.3.5 Infection des systèmes des utilisateurs](#)

Pour éviter l'infection des systèmes utilisateurs, il faut d'abord empêcher l'injection HTML et JavaScript. Dans le cas où les utilisateurs peuvent s'envoyer des liens/fichiers (ce qui n'est pas le cas actuellement), il faut penser à mettre en place un système de scan anti-virus pour avertir si le mail contient un virus.

Scénario [3.3.6 Distributed Denial of Service](#)

Il existe des systèmes anti-ddos que l'on peut acheter. Si l'application est exposée sur Internet, cela vaut la peine d'investir dans un de ces systèmes. Sinon, la meilleure solution reste de ne pas exposer ce service sur Internet si cela n'est pas nécessaire.

Scénario [3.3.7 Distributed Denial of Service v.2](#)

Il faut empêcher l'injection HTML sur l'application.

Scénario [3.3.8 Bruteforce sur les mots de passe](#)

Pour éviter qu'un attaquant ne fasse du bruteforce sur un mot de passe utilisateur, il faut limiter le nombre de tentatives de connexion erronées (par IP et par utilisateur). Il est également recommandé d'imposer un mot de passe fort pour tous les utilisateurs et surtout, une authentification à double facteur pour les administrateurs.

5 Sécurisation de l'application

Pour améliorer la sécurité de l'application, les mécanismes recommandés sont notamment de valider les valeurs reçues de l'utilisateur. Il est conseillé d'utiliser des requêtes préparées pour toute interaction avec la base de données (et ainsi, éviter les injections SQL). Il faut contrôler les droits d'accès des utilisateurs pour chaque chargement de page. Il faut également limiter le nombre de requêtes effectuées par IP et utilisateur afin d'éviter l'automatisation d'envoi de requêtes. Un *token* CSRF doit être ajouté pour éviter les attaques CSRF, un exemple a d'ailleurs été implémenté dans le fichier *users.php*.

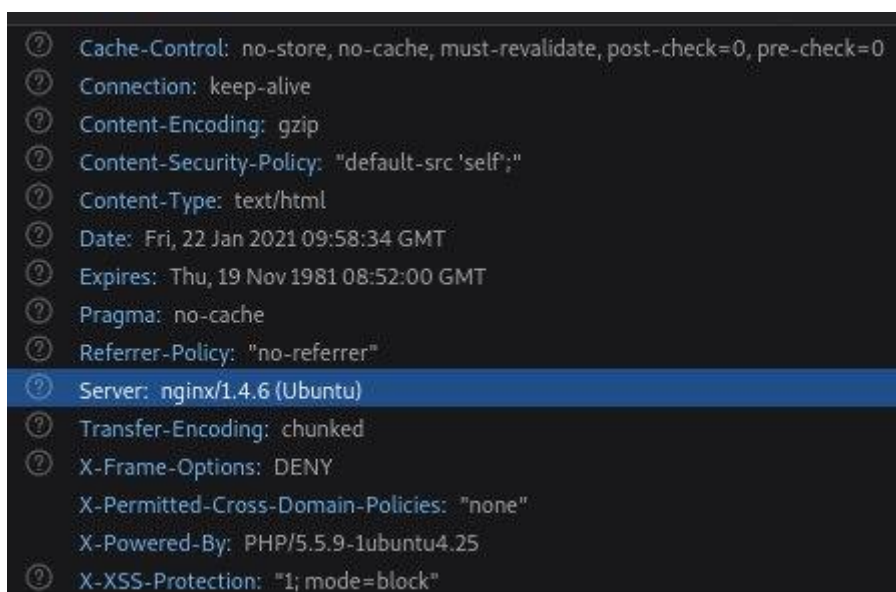
Des headers HTTP peuvent (doivent) être ajoutés pour sécuriser l'application. Certains headers permettent d'éviter par exemple les attaques XSS et le clickjacking. Dans le cadre du projet, nous avons ajouté les headers suivants :

```
header( header: 'X-Frame-Options: DENY');
header( header: 'X-XSS-Protection: 1; mode=block');
header( header: 'X-Permitted-Cross-Domain-Policies: none');
header( header: 'Referrer-Policy: no-referrer');
header( header: 'X-Content-Type-Options: nosniff');

ini_set( option: 'session.cookie_httponly', value: true);
// ↓ don't work with php 5.5.9
//ini_set('session.cookie_samesite', "Strict");
```

Ces headers peuvent être inclus dans les fichiers PHP ou directement au niveau du serveur Web. Le cookie devrait être défini en tant que *httponly* et avec le paramètre *samesite* : *Strict*. Cependant, la version actuelle de PHP ne le supporte pas : il faut donc procéder à une mise à jour de PHP.

Pour éviter des attaques ciblées sur les technologies utilisées pour l'implémentation de l'application, il faut faire en sorte que le serveur n'indique pas d'information de version :



```
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Connection: keep-alive
Content-Encoding: gzip
Content-Security-Policy: "default-src 'self';"
Content-Type: text/html
Date: Fri, 22 Jan 2021 09:58:34 GMT
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Pragma: no-cache
Referrer-Policy: "no-referrer"
Server: nginx/1.4.6 (Ubuntu)
Transfer-Encoding: chunked
X-Frame-Options: DENY
X-Permitted-Cross-Domain-Policies: "none"
X-Powered-By: PHP/5.5.9-1ubuntu4.25
X-XSS-Protection: "1; mode=block"
```

Sur la capture d'écran ci-dessus, nous constatons par exemple que le serveur est un « nginx/1.4.6 (Ubuntu) ». Cette information-là peut être cachée, ainsi que celle fournie avec le header « X-Powered-By ». Ces envois d'informations peuvent être désactivés directement sur le serveur.

Nous avons désactivé l'envoi d'information via le header « X-Powered-By » au moyen de cette ligne dans le conteneur Docker :

```
RUN echo "expose_php = off" >> /etc/php5/fpm/php.ini
```

Une bonne pratique serait d'ajouter un système de logs pour une analyse après un potentiel incident. On pourrait implémenter un contrôle qui empêcherait le lancement de l'application si le fichier *install.php* est présent, car ce fichier permet de réinitialiser la base de données.

6 Conclusion

Ce rapport de modélisation de menaces permet d'identifier des menaces, d'imaginer des scénarios d'attaque et de déterminer des contremesures pour chaque cas. Cette démarche permet d'élever le niveau global de sécurité de l'application.

Ainsi, au terme de notre réflexion, nous avons décrit le système existant et identifié des sources de menaces. Nous avons imaginé des scénarios d'attaque sur-mesure et trouvé des mécanismes pour remédier aux attaques potentielles.

Après rédaction de ce rapport, nous pouvons sécuriser notre application en intégrant nos éléments de réflexion dans l'implémentation du système. Nous avons notamment modifié notre application en échappant les caractères pour éviter certaines injections SQL et HTML, ou nous avons indiqué avec un commentaire les points d'entrée qu'il faut encore corriger. Nous avons également mis en place un *token* CSRF pour éviter les attaques par requêtes forgées.