

HEIG-VD

SÉCURITÉ DES TECHNOLOGIES INTERNET

RAPPORT D'ÉTUDE DE MENACES

Messagerie électronique sécurisée

Auteurs :

Yosra HARBAOUI

Luana MARTELLI

Professeurs :

Abraham RUBINSTEIN

Yohan MARTINI

13 janvier 2018

heig-vd

Table des matières

1	Introduction	2
2	Environnement	2
3	Description du système	2
3.1	Prérequis	2
3.2	Sécurité	3
3.3	Data Flow Diagrams	4
3.4	Identification des biens	5
3.5	Définition du périmètre de sécurisation	5
4	Scénarios d’attaques	6
4.1	Identification des sources de menaces	6
4.2	Éléments du système attaqué	6
4.3	Motivations	7
4.4	STRIDE	7
4.5	Scénarios d’attaque et contre-mesures	8
4.6	Dans le futur	18
5	Difficultés rencontrées	19
6	Conclusion	19

1 Introduction

Dans le premier projet, il a été question de développer une application de messagerie. Le cahier des charges consistait à mettre en place une interface web permettant à un client (utilisateur ou administrateur) de pouvoir se connecter à sa messagerie. Il pouvait envoyer des messages, en recevoir, et éditer son profil. Si le client était administrateur, alors il pouvait aussi avoir accès à la liste des utilisateurs.

Dans le cadre de ce deuxième projet, nous devons sécuriser cette application. L'aspect sécuritaire a été volontairement mis de côté lors de l'implémentation du programme. Ici, il s'agit d'identifier les menaces, de décrire des scénarios et finalement d'implémenter des contre-mesures, afin de rendre notre application utilisable et sécurisée.

2 Environnement

L'application côté serveur est codée en PHP version 5.4. La base de donnée utilise la technologie SQLite. Cette application a été testée sur une machine virtuelle CentOS 7.1.

3 Description du système

3.1 Prérequis

Selon le cahier des charges du premier projet, quelques prérequis étaient à mettre en place :

- sans authentification, seule la page de *login* est accessible,
- un mécanisme d'authentification simple (nom d'utilisateur et mot de passe) a été mis en place,
- un utilisateur peut être administrateur ou collaborateur,
- un compte peut être actif ou inactif; s'il est inactif, alors l'utilisateur n'a pas accès à son compte,
- un collaborateur peut changer son mot de passe et consulter ses messages; il peut en créer ou en supprimer,
- un administrateur a les mêmes fonctionnalités qu'un collaborateur; en plus, il a accès à la liste des utilisateurs et peut modifier l'état des comptes.

3.2 Sécurité

Un aspect sécuritaire a été pris en compte durant le développement de la phase une. Afin d'assurer un bon fonctionnement de l'application, les pages qui nécessitent une authentification ne sont accessibles qu'aux personnes authentifiées, les autres sont redirigés sur la page de *login*. De plus, un simple utilisateur authentifié n'a pas accès aux pages d'administration.

Enfin, les accès à la base de données étaient déjà sécurisés, car ils utilisaient déjà les fonctions approuvées par la documentation. Cependant, les injections SQL sont une des menaces les plus connues dans les attaques d'application web. C'est pourquoi nous lui avons quand même consacré un scénario.

3.3 Data Flow Diagrams

Notre application est conçue comme suit : un utilisateur ou un administrateur a accès à des pages, selon ses accréditations. Toutes les requêtes se font via l'interface web. Dans l'exemple ci-dessous sur la Figure 3.3, on suppose que les deux sont loggués correctement.

Toutes les demandes sont effectuées sous forme de fonctions. Toutes ces fonctions sont recensées dans des modèles (deux exactement : user et message). Ce sont ces modèles qui effectuent les requêtes auprès de la base de données.

La base de données renvoie les informations correspondantes et l'application web les affiche.

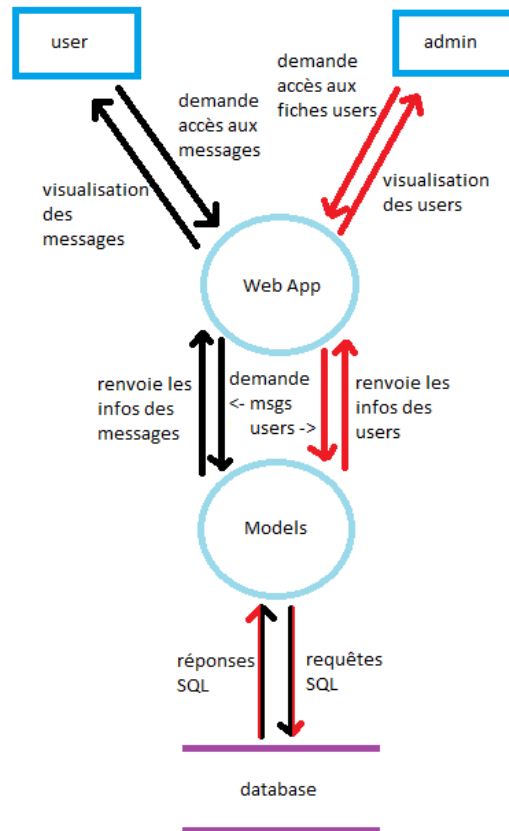


FIGURE 1 – Data Flow Diagram de l'application

3.4 Identification des biens

Dans notre application, les données sensibles qu'il est nécessaire de protéger sont les informations utilisateurs (nom et mot de passe), ainsi que les messages échangés. Un attaquant peut vouloir voler ses informations pour diverses raisons, explicitées plus bas dans ce rapport. L'accès à la base de données doit donc être protégée. Le serveur, qui héberge l'application, est aussi un bien sensible.

3.5 Définition du périmètre de sécurisation

Le scénario le plus plausible dans le cadre de cette application est un attaquant, ayant accès au service de messagerie, tente de trouver des failles afin de faire dysfonctionner le système selon ses désirs. Il s'agit donc avant tout de sécuriser le côté client, notamment de contrôler attentivement les formulaires auxquels un utilisateur a accès. Il est questions ici d'attaques actives, où l'attaquant essaye de prendre le contrôle de l'application.

On peut aussi imaginer que l'attaquant effectue une écoute passive. Il faut donc aussi penser à sécuriser la transmission de données.

Un des risques encourus par toutes applications est une attaque de type déni de service. Il est en effet possible pour un attaquant d'inonder le serveur de requêtes afin de le rendre inapte. Il existe diverses possibilités pour s'en prémunir, ou en tout cas dissuader l'attaquant, comme par exemple bloquer les adresses IP qui font trop de requêtes dans un court laps de temps. Cependant, un attaquant peut contourner cette mesure en utilisant des proxys, ou simplement plusieurs machines. La question du déni de service étant vaste et complexe et l'application n'étant pour le moment hébergée que sur un ordinateur en *localhost*, nous ne l'avons pas pris en compte dans le cadre de ce projet.

4 Scénarios d’attaques

4.1 Identification des sources de menaces

- **Pirate.** Le but de sa manœuvre serait simplement de détruire l’application, par challenge ou parce qu’une source externe l’aurait payé pour. Il pourrait aussi vouloir récupérer les informations contenues dans la base de données, par exemple les mots de passe, car la plupart des utilisateurs utilisent le même mot de passe pour plusieurs applications.
- **Utilisateur frustré ou malveillant.** Si l’utilisateur est suffisamment qualifié pour hacker l’application lui-même, il ne va pas avoir recours aux services d’un pirate. Ses motivations sont diverses : son compte a été désactivé et il souhaite le récupérer, il tente d’élèver ses privilèges pour devenir administrateur. Finalement, il peut aussi vouloir lire des messages qui ne lui étaient pas destinés.
- **Programme malveillant.** La différence entre le pirate et le programme est que, afin d’empêcher le pirate d’agir, on va essayer de le forcer à s’authentifier afin de restreindre ses droits en conséquence. Ainsi, il ne sera pas en mesure d’accéder à l’application. Le programme, lui, ne passe pas par la case authentification. Il s’agit donc, ici, de contrôler les injections de code ou les requêtes suspicieuses et de bloquer les programmes non-reconnus par l’application.
- **Menace physique.** On entend ici une catastrophe naturelle, un sinistre, comme un incendie ou une inondation qui endommagerait le matériel qui contient l’application.
À noter que ce type de menaces n’est pas pris en compte dans le présent rapport. Nous avons effectué des sauvegardes régulières de l’application. S’il devait se produire une catastrophe et que l’application n’est plus accessible, merci de nous contacter afin que nous vous fournissions le code source.

4.2 Éléments du système attaqué

Les principaux points faibles de l’application sont tous les champs utilisateurs. En effet, s’ils ne sont pas protégés, un attaquant peut les utiliser

pour obtenir des informations, en ajouter, les modifier ou les supprimer. Ils sont ce qui relie l'utilisateur à la base de données. C'est pourquoi il faut leur amener une attention particulière.

Un attaquant peut aussi écouter passivement le réseau et acquérir des informations. Les communications entre le serveur et le client se faisant via un réseau, il est donc aussi ciblé.

Le serveur peut aussi se faire physiquement attaqué. Il faut donc mettre en place un système de backups réguliers. Cette éventualité ne rentre pas dans le cadre du projet, mais c'est un élément à ne pas négliger pour une amélioration future.

Finalement, un attaquant pourrait effectuer du *social engineering* sur un utilisateur afin d'obtenir des informations. Cette attaque ne rentre pas non plus dans le cadre du projet, mais dans un cas réel, il faudrait sensibiliser les utilisateurs à ce fait.

4.3 Motivations

Selon les sources de menaces identifiées ci-dessus et dans le cadre de cette application, les motivations d'un attaquant peuvent être variées : la volonté d'élever ses privilèges afin de pouvoir modifier des paramètres de comptes, l'envie d'accéder à des messages qui ne lui étaient pas destinés.

4.4 STRIDE

STRIDE est un acronyme pour différents niveaux de menaces. Il s'agit d'un modèle de classification qui permet de se rappeler des différentes menaces que l'on peut rencontrer dans une application. Les catégories sont les suivantes : *Spoofing* (vol d'identité), *Tampering* (sabotage), *Repudiation*, *Information disclosure* (leak de données), *Denial of service*, *Elevation of privileges*.

4.5 Scénarios d’attaque et contre-mesures

Pour les scénarios qui suivent, nous imaginerons que Jean-Kévin est un administrateur de notre application. Tous les scénarios d’attaque décrits ci-dessous sont aussi valables pour un utilisateur lambda. Cependant, il est toujours plus intéressant de viser la session d’un administrateur visant ainsi plus de privilèges. Mallory sera l’attaquante qui souhaite prendre le contrôle de l’application.

Il n’y a pas d’ordre particulier dans les scénarios.

Scénario 1 - *Mots de passe faible*

Attaque

Dans ce premier scénario, nous partons du principe que Mallory a réussi à obtenir une liste de noms d’utilisateur de l’application de messagerie. Grâce à cela, elle va pouvoir tenter de *brute-force* les mots de passe des comptes et ainsi pouvoir dérober des informations ou usurper des identités. Dans l’application de base, il n’y a aucune mesure particulière mise en place pour contrôler la force des mots de passes utilisée. Il y a donc de fortes chances que les utilisateurs aient utilisé des mots de passe dits “faibles”.

Classification : SI(E)

En partant du principe que l’attaque réussisse et que Mallory obtienne des mots de passe, elle peut non seulement voler des identités (S), mais aussi obtenir l’accès à des informations sensibles (I), comme par exemple des messages. Pire encore, si on part du principe que beaucoup d’utilisateurs utilisent le même mot de passe pour tous leurs comptes, Mallory y aura également accès. Finalement, il est possible que Mallory arrive à hacker un compte administrateur, ce qui lui donnerait les pleins pouvoirs (E).

Contre-mesures

Afin de contre-carrer les plans de Mallory, Jean-Kévin a décidé d’agir. Tout d’abord, lorsqu’une connexion échoue, l’utilisateur n’a pas d’informations sur le paramètre erroné. Il reçoit simplement :

Wrong credentials!

Ainsi, Mallory ne peut pas établir une liste d’utilisateurs.

De plus, l’URL ne divulgue pas d’informations sur un compte. Tout est stocké dans la session. C’est pourquoi, il est impossible de *brute-force* des URL contenant des noms d’utilisateurs.

Finalement, l'entrée d'un nouveau mot de passe choisi par l'utilisateur est problématique. En effet, afin de contrer les mots de passe “faibles”, une possibilité est d'obliger l'utilisateur à entrer un nombre minimal de caractères, des chiffres, des lettres, ainsi que des caractères spéciaux. Cependant, ce genre de pratique pousse (trop) souvent l'utilisateur à ne pas retenir ce mot de passe et à le noter sous le clavier.

Dans un cas comme dans l'autre, il y a un risque non-négligeable que le mot de passe soit “faible” (que ce soit dans sa construction ou dans son maintien). Jean-Kévin étant conscient des risques, il a décidé que, dans le cadre de cette application, il est plus probable qu'un hacker obtienne une liste des noms d'utilisateurs et tente de *brute-force* leur mot de passe. Il a donc choisi d'obliger l'utilisateur à choisir un mot de passe “fort”. Le risque que l'utilisateur l'écrive derrière son clavier ne rentre pas dans le périmètre de sécurisation du projet. Un mot de passe doit contenir au moins huit caractères, dont au moins une majuscule, un chiffre et un caractère spécial.

```
function check_password($password) {
    $uppercase = preg_match('/[A-Z]/', $password);
    $lowercase = preg_match('/[a-z]/', $password);
    $number     = preg_match('/\d/', $password);
    $special    = preg_match('/[^a-zA-Z\d]/', $password);
    $length     = 8;

    if($uppercase == 1
        and $lowercase == 1
        and $number == 1
        and $special == 1
        and strlen($password) >= $length) {
        return true;
    }
    return false;
}
```

Scénario 2 - *Hachage de mots de passe*

Attaque

Jean-Kévin le sait, un mot de passe, c'est important. Surtout si un utilisateur l'utilise pour plusieurs applications. C'est pourquoi Jean-Kévin a stocké le haché du mot de passe dans la base de données. Ainsi, si la base de données est compromise, les mots de passe ne sont pas dévoilés. Cependant, afin d'éviter une attaque sur les mots de passe à l'aide de dictionnaires de hachés, comme les *rainbow tables*, il faut ajouter un sel et le stocker.

Classification : I

Le mot de passe est une des informations les plus sensibles d'une application. Les voler et les craquer est évidemment une fuite d'informations confidentielles (I).

Contre-mesures

La version de PHP utilisée pour ce projet est 5.4. La fonction `crypt()` utilisée génère des hachés MD5 avec un sel choisi selon l'implémentation de la fonction. Il n'est donc pas nécessaire d'en ajouter un. Ce sel est cependant décrit dans la documentation comme "faible". Cependant, plutôt que de générer un sel aléatoire et le stocker, il serait plus simple de mettre à jour la version de PHP à au moins 5.5, car cette version contient `password_hash()` qui génère des hachés et des sels "forts" ¹.

Note : Le mot de passe de base du premier compte (admin/admin) est très faible et très simple. Il est donc fortement conseillé, une fois la mise en place de l'application et de la base de données, de modifier ce mot de passe.

Scénario 3 - *Divulgateur d'information à travers l'URL*

Attaque

Dans l'état actuel de l'application, les URL existantes qui ne doivent pas être accessibles renvoient un 200 (car la page a été trouvée), mais est complètement blanche. En revanche, celles qui n'existent pas renvoient un 404. Afin d'obtenir des informations dans le but d'attaquer une application, Mallory peut tenter d'accéder à des URL diverses pour découvrir quels sont les liens qui retournent une erreur ou non.

Classification : I

Comme expliqué dans l'attaque, il est possible d'obtenir des informations sur l'arborescence de l'application (I). Ce genre d'information peut s'avérer

1. Source : <https://secure.php.net/manual/fr/function.crypt.php>

utile dans un cas de *reverse engineering* de l'application.

Contre-mesures

Jean-Kévin a fait deux changements dans le code. Tout d'abord, il a ajouté un fichier `index.php` dans chaque dossier pour éviter de révéler l'arborescence de l'application. Dans ce fichier, l'utilisateur est automatiquement redirigé sur la page de *login*.

De plus, il a modifié le fichier `httpd.conf` comme suit :

```
ErrorDocument 404 http://localhost
```

Les requêtes retournant un 404 sont donc toutes redirigées sur la page `localhost`. Ainsi, si un pirate tente d'accéder à des URL de manière pseudo-aléatoire, il n'obtiendra pas d'information en plus.

Scénario 4 - Informations sur le serveur

Attaque

Jean-Kévin, effectuant quelques tests sur son application, se rend compte que - horreur ! - la version du serveur qu'il utilise est visible en clair par tous. Cette information étant compromettante, il décide de remédier à ça. En effet, il suffit d'entrer la version du serveur sur l'Internet mondial pour y trouver toutes les failles répertoriées.

Classification : STRIDE

Selon le serveur utilisé et les failles correspondantes trouvées, l'attaque peut menacer toutes les catégories.

Contre-mesures

Jean-Kévin a modifié le fichier de configuration `/etc/httpd/conf/httpd.conf`, en y ajoutant deux lignes

```
ServerTokens Prod
ServerSignature Off
```

Afin de cacher les informations sur le serveur. **ServerToken** notifie le niveau d'information que le serveur va transmettre dans l'entête HTTP. Dans notre cas, **Prod**, fournit juste le nom du serveur, qui est l'information minimale. Par défaut, toutes les informations sont affichées. **ServerSignature**, s'il est activé, note une ligne en bas de l'application contenant le serveur et sa version. Par défaut, cette option est désactivée, mais il est plus sage et plus lisible de l'écrire ici.

Scénario 5 - *Injection SQL*

Attaque

Mallory a appris qu'un site ayant une base de données et n'étant pas protégé est vulnérable à des attaques de type injections SQL. Grâce à cette attaque, elle pourrait modifier la base de données à sa guise, supprimer ou ajouter des informations. Elle tente donc d'entrer la commande suivante dans la page de *login* :

```
' OR 1=1 //
```

Cette commande ferme le champ de texte (') et le met à **TRUE**. Tout le reste (donc le mot de passe) est commenté. Dans un cas où aucune sécurité n'est mise en place, cette instruction retourne **TRUE** et l'attaquant a donc accès aux informations de la table concernée. Mallory pourrait donc récupérer des données sensibles, comme des mots de passe, des rôles, etc...

Classification : SIE

Un accès non sécurisé à une base de données amène les pleins pouvoirs. Un attaquant peut donc modifier des comptes ou les messages (SIE).

Contre-mesures

Tout d'abord, un premier contrôle est effectué pour s'assurer que les deux champs soient bien remplis :

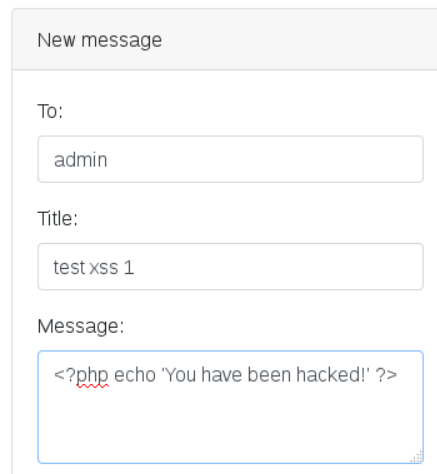
```
if (!empty ($_POST['login']))  
and !empty ($_POST['password']))
```

Ensuite, dans la fonction `authenticate_user()` du fichier `user`, Jean-Kévin a utilisé les fonctions `prepare()` et `execute()` qui permettent de parser les strings et de rendre impraticable les injections SQL. Toute information entrée par l'utilisateur est alors traitée comme une string et il est donc impossible pour Mallory de pratiquer cette attaque.

Scénario 6 - *Injection XSS*

Attaque

Dans la même idée que le scénario précédent, il est possible d'injecter du code et de l'exécuter dans les entrées utilisateurs. Dans l'application de la partie 1, il était possible de faire cela :



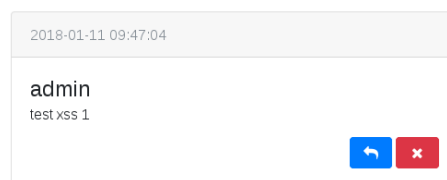
New message

To:
admin

Title:
test xss 1

Message:
<?php echo 'You have been hacked!' ?>

FIGURE 2 – Attaque XSS : Préparation



2018-01-11 09:47:04

admin
test xss 1

↩ ✕

FIGURE 3 – Attaque XSS : Résultat

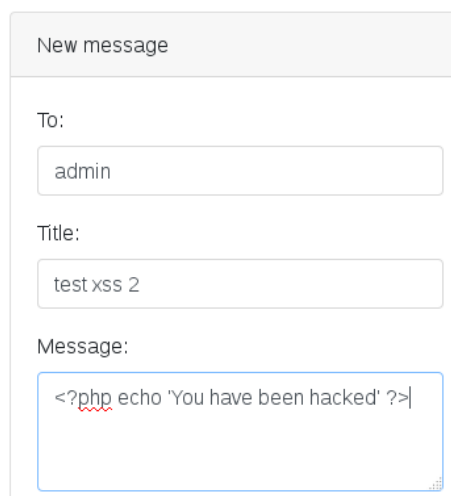
Lorsque l'on veut lire le message, le corps est vide. On voit donc que le code a été exécuté.

Classification : STIDE

Puisque dans ce scénario, Mallory exécute le code qu'elle souhaite, elle peut potentiellement menacer toutes les catégories.

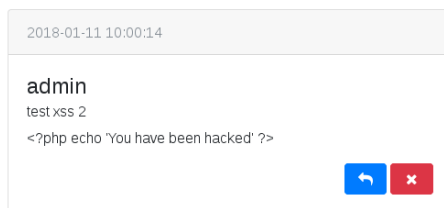
Contre-mesures

Jean-Kévin a utilisé la fonction `htmlspecialchars()` qui transforme les caractères spéciaux en caractères normaux. Ils deviennent donc non-exécutables. L'attaque mentionnée ci-dessus devient donc irréalisable :



The image shows a web form titled "New message". It has three input fields: "To:" with the value "admin", "Title:" with the value "test xss 2", and "Message:" with the value "<?php echo 'You have been hacked' ?>". The "Message:" field is highlighted with a blue border.

FIGURE 4 – Patch XSS : Préparation



The image shows an email message received on 2018-01-11 10:00:14. The sender is "admin" and the subject is "test xss 2". The body of the email contains the text "<?php echo 'You have been hacked' ?>". At the bottom right of the email, there are two buttons: a blue button with a left arrow and a red button with a close 'X'.

FIGURE 5 – Patch XSS : Résultat

Le message est maintenant visible.

Scénario 7 - *Brute force d'un compte*

Attaque

Mallory a réussi à récupérer le nom du compte de Jean-Kévin. Elle décide donc d'utiliser un outil pour *brute-force* les mots de passe et de tous les tester. Ce scénario diffère du premier car dans le cas précédent, on cherchait un compte avec un mot de passe faible. Dans ce cas, l'attaque est ciblée

sur un utilisateur, et on peut donc tester tous les mots de passe possible, y compris des mots de passe “forts” (même si du coup cela rallonge le temps de recherches).

Classification : (S)I(E)

Ici, Mallory cherche à obtenir une information sur un utilisateur particulier (I). Dans le cas où elle arriverait à ses fins, elle peut voler son identité (S) et, puisqu’il s’agit du compte de Jean-Kévin, élever ses privilèges (E).

Contre-mesures

Jean-Kévin a eu vent des plans de Mallory. Il décide alors de protéger un peu mieux la page de *login*. Au lieu de mettre un time-out, au bout d’un certain temps, afin d’obliger l’attaquant à patienter, il a l’idée de mettre en place un captcha. Dès que l’utilisateur veut se connecter, il doit remplir un captcha. Après avoir téléchargé une librairie², il ajoute dans la page *login* les quelques lignes :

Pour la partie html

```
<div class="form-group">
  <?php
    echo Securimage::getCaptchaHtml();
  ?>
</div>
```

Et pour la partie PHP (c’est-à-dire contrôle des données)

```
require_once '../captcha/securimage/securimage.php';

/* code */

$image = new Securimage();
if ($image->check($_POST['captcha_code']) == true)

/* suite du code */
```

Scénario 8A - Vol de session Partie 1

Attaque

Un vol de session est possible de plusieurs manières. Mallory peut écouter passivement le réseau et tenter d’intercepter un *ID session*. Ainsi, elle pourra

2. Source : <https://github.com/dappph/securimage>

le réinjecter et accéder aux données de Jean-Kévin. Une autre manière est d'écrire un script qui récupère cet ID et le renvoie à l'attaquant. Dans ce scénario, il s'agit de protégé avant tout l'ID de la session, afin qu'il ne soit pas récupérable facilement.

Classification : SI(E)

Pour ce scénario, le pirate souhaite avant tout endosser l'identité de l'utilisateur courant (S), afin d'accéder à des informations privées (I). Dans le cas où la victime est un administrateur, le pirate peut élever ses privilèges (E).

Contre-mesures

Dans le fichier qui vérifie la session courante, on peut ajouter ces quelques lignes, afin de s'assurer, d'une part, que les cookies ne passent que par le protocole HTTP et ne peuvent donc pas être récupéré autrement :

```
/* Prevents attacks in order to steal the session ID */
ini_set('session.cookie_httponly', 1);
/* Session ID cannot be passed through URLs */
ini_set('session.use_only_cookies', 1);
/* Unauthorize sesssion without id */
ini_set('session.use_strict_mode', 1);
```

Scénario 8B - *Vol de session Partie 2*

Attaque

Jean-Kévin n'est pas satisfait. Bien que les cookies ne transitent plus que par le protocole HTTP, les informations circulent toujours en clair. Pour un attaquant possédant un sniffer de réseau, il est donc très facile d'avoir accès aux informations sensibles. Jean-Kévin décide donc de mettre en place un certificat TLS, afin que toute son application soit en HTTPS.

Classification : SI(E)

Les menaces ciblent ici les mêmes classifications que pour le scénario A.

Contre-mesures

Afin de passer de HTTP à HTTPS, il suit à la lettre un tutoriel qu'il a trouvé en ligne³. En quelques étapes, Jean-Kévin, crée une paire de clés publique/privée, un certificat qu'il signe lui-même, et met en place une politique de sécurité dans un fichier de configuration. Il écrit aussi un script qui

3. Source : <https://www.digitalocean.com/community/tutorials/how-to-create-an-ssl-certificate-on-apache-for-centos-7>)

permet à un utilisateur d'installer ce certificat `conf/ssl/enable-https.sh`. Ouf, Jean-Kévin est à présent satisfait.

Note : Le certificat ayant été auto-signé, il est nécessaire de l'autoriser lors de la première connexion au site après l'avoir installé.

Scénario 9 - *Fonction logout*

Attaque

Jean-Kévin aime bien lire ses e-mails dans un cybercafé avant de partir au travail. Son café terminé, il se déconnecte de sa session et part, en pensant que son compte est désormais inaccessible. Erreur ! Mallory, qui l'espionnait, prend alors sa place devant l'ordinateur. Elle appuie sur le bouton "retour arrière" du navigateur web. Horreur ! Elle a désormais accès aux mails de Jean-Kévin !

Scénario alternatif : Jean-Kévin part sans en oubliant de se déconnecter - Horreur x2 ! -. Mallory arrive cinq minutes après et a accès à tous ses e-mails !

Classification : SI(E)

Dans ce cas, le pirate récupère simplement la session du dernier utilisateur, la fonction `logout()` ne faisant rien, si ce n'est revenir à la page principale. Il peut donc se faire passer pour sa victime (S), lire ses e-mails (I) ou modifier des comptes (E), s'il tombe -par malchance- sur le compte d'un administrateur.

Contre-mesures

Dans la fonction `logout()`, Jean-Kévin a utilisé les fonctions suivantes

```
session_start();  
session_unset();  
session_destroy();
```

Afin de pouvoir réutiliser à nouveau ses variables de sessions, on appelle la fonction `session_start()`. Ensuite, on utilise la fonction `session_unset()` qui libère toutes les variables de session et finalement `session_destroy()` qui détruit toutes les variables. Cela nous permet donc bien de libérer la session et de ne pas pouvoir la récupérer en un simple clic. Afin de ne pas se retrouver dans le cas du scénario alternatif, nous avons ajouté dans le fichier `check_session.php` un *time out*. Si une session reste inactive plus de cinq minutes, la session est détruite.

```
if (isset($_SESSION['LAST_ACTIVITY']))
```

```

    if((time() - $_SESSION['LAST_ACTIVITY']) > 300) {
        session_start();
        session_unset();
        session_destroy();
        header('Location: ./index.php');
    }
}
$_SESSION['LAST_ACTIVITY'] = time();

```

4.6 Dans le futur

Cette sous-section présente un scénario qui n'a pas été implémenté dans le projet. Cependant, il nous semblait important de le présenter ici, pour une implémentation future.

Scénario 10 - *PGP*

Attaque

Dans le cas où attaquant aurait accès à la base de données, il pourrait voir tous les messages en clair. De plus, l'intégrité des messages n'est pour le moment pas assurée. Un utilisateur pourrait donc écrire un message, l'envoyer, et ensuite, assurer qu'il n'a rien fait et que son compte a été piraté.

Classification : SRIE I

Ici, les données sensibles sont la cible de cette attaque. Si Mallory a un accès complet à la base de données, elle peut modifier des messages.

Contre-mesures

Il faudrait mettre en place un système qui permet à la fois d'assurer la confidentialité, l'intégrité et l'authentification des messages et de leur auteur. Un bon moyen est le système Pretty Good Privacy. Il s'agit de générer deux paires de clés qui vont permettre, respectivement, de chiffrer les données et de les signer. Ainsi, en cas de vol de base de données, elles restent protégées. De plus, leur intégrité et leur auteur sont assurés grâce à la signature.

Il existe des outils qui implémentent un service PGP pour PHP. La documentation est cependant un peu barbare. Nous réservons cette implémentation pour un déploiement futur de l'application.

5 Difficultés rencontrées

Il existe de nombreux tutoriels pour apprendre le PHP et sécuriser des applications web. L'écriture du code n'a donc pas posé trop de problèmes. De plus, dans beaucoup de cas, cela n'a demandé que quelques lignes supplémentaires. Ce qui a pris le plus de temps à été de définir avec précision le périmètre de sécurité, c'est-à-dire choisir qu'est-ce qu'il faut sécuriser et qu'est-ce qui est le plus vulnérable.

6 Conclusion

En effectuant des recherches afin de sécuriser notre application, nous avons remarqué que certaines manipulations avaient déjà été effectuées. Cela est dû au fait que lorsque nous avons fait des recherches pour implémenter telle ou telle fonction, le standard utilisé était (souvent) celui qui était le mieux sécurisé. C'est une bonne nouvelle et cela montre que même si on y prête pas trop attention, quelques mesures de sécurités sont mises en place de base.

Ce deuxième projet a été très sympa à réaliser et nous a montré le genre de réflexion à avoir lorsque l'on développe une application web.