

STI - Projet Partie 2

Rapport

Léo Cortès

Steve Henriquet

7 janvier 2019



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Hes·SO
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Contents

1	Introduction	5
2	Description du système	6
2.1	Identification des biens	6
2.2	Périmètre de sécurisation	6
2.3	Agents menaçants	7
2.4	Vecteurs d'attaque	7
2.5	Impacts techniques	7
2.6	Impacts business	7
3	Vulnérabilités, scénarios d'attaque et contre-mesures	8
3.1	XSS	8
3.2	Motivation	8
3.2.1	Exploitation : démo basique	8
3.2.2	Exploitation : vol de cookies	9
3.2.3	Contre-mesures	10
3.3	Injection SQL	11
3.4	Motivation	11
3.4.1	Exploitation : se connecter sans mot de passe	12
3.4.2	Exploitation : changer le mot de passe de n'importe quel utilisateur	13
3.4.3	Contre-mesures	14
3.5	Mauvaise destruction des session	14
3.6	Motivation	14
3.6.1	Exploitation	15
3.6.2	Contre-mesures	16
3.7	Accès au page sans login	17
3.8	Motivation	17
3.8.1	Exploitation	17
3.8.2	Contre-mesures	18
3.8.3	Spamming	19
3.8.4	Contre-mesure	19
3.9	Droits d'administrateurs retirés au logout	20
3.9.1	Contre-mesures	20
3.10	Problèmes non résolubles	21

List of Figures

1	Message malicieux	8
2	Boîte de réception	8
3	Exécution XSS	9
4	Message malveillant voleur de cookier	9
5	La faille n'est pas visible pour la victime	10
6	Le cookie volé	10
7	Exemple d'assainisation	10
8	Résultat de <i>SQLMap</i> sur la page de login	11
9	Se connecter comme <i>user.test</i> sans connaître son mot de passe	13
10	Avant d'utiliser des prepared statements il y avait juste de simples queries	14
11	Exemple de prepared statement	14
12	Mauvaise destruction de la session	15
13	Accès réservé aux admins	16
14	Mauvaise destruction de la session	16
15	Accès à la page	17
16	Requête Post réussi	18
17	Message reçu	18
18	Protection à ajouter aux différentes pages	18
19	Protection anti-spam : HTML et JS	19
20	Protection anti-spam : CSS	20
21	Protection anti-spam : PHP	20
22	Vérification des actions admin	21

Introduction

Ce projet est la continuité d'une première partie qui a été réalisé plus tôt dans le semestre. Il s'agit de réaliser une analyse de menaces de notre application, de trouver les vulnérabilités qui subsistent et de sécuriser l'application.

Description du système

Le cahier des charges est le même que pour la première partie. L'application est une simple webapp permettant d'échanger des messages entre utilisateurs. Les utilisateurs doivent posséder un compte pour pouvoir échanger des messages. Un utilisateur peut être administrateur. Si c'est le cas, il aura accès à des fonctionnalités supplémentaires lui permettant d'éditer les informations d'autres utilisateurs.

Identification des biens

L'application est une messagerie en ligne basique. Pour pouvoir envoyer ou recevoir un message, un utilisateur a besoin d'un compte. Les comptes sont stockés dans une base de données et sont des biens à protéger. On y retrouve notamment des adresses mails et des mots de passes.

Le second type de biens sont les messages en eux-même. Ceux-ci doivent rester confidentiels car ils ne concernent que deux utilisateurs.

Périmètre de sécurisation

Comme nous avons accès au code de notre application, nous avons une idée des vulnérabilités que nous pouvons y trouver. Dans un premier temps, les entrées utilisateur ne sont pas assainies et sont envoyées telle quelle. Cela laisse place à de nombreuses failles comme des injections SQL ou des XSS. Les entrées utilisateur doivent donc être vérifiées et assainies.

De plus, les accès à la DB n'utilisent pas de query pré-préparées. Cela pourrait donc donner lieu à des injections SQL supplémentaires. Il faudra modifier le code de l'application pour sécuriser les communications à la base de données.

Les accès sont, en principe, vérifiés. Un utilisateur ne peut accéder qu'aux mails qu'il a envoyé ou reçu et pas ceux d'autres utilisateurs. Les accès administrateurs sont réservés aux administrateurs. Finalement, il n'est pas possible d'accéder à des fichiers stockés sur le serveur. Si un utilisateur essaie d'accéder à une page non prévue, il sera redirigé (c'est au moins ce que nous pensions, nous verrons plus tard que ce n'était pas tout à fait le cas).

L'application utilise du HTTP simple et aucun chiffrement. Des données pourraient donc être sniffées et si une fuite de données a lieu, toutes les données pourraient être accessibles en clair. Idéalement, l'application devrait

utiliser du HTTPS, chiffrer les données et hasher les mots de passes mais nous ne le ferons pas ici.

Agents menaçants

En principe, notre application devraient être accessibles depuis n'importe qui sur Internet. Les agents menaçants peuvent donc être n'importe qui. Notre application n'est pas révolutionnaire et bien moins efficace que la concurrence. Des attaquants concurrents seraient donc peu probables. Les principaux agents menaçants seraient surtout nous (les élèves du cours STI) ou n'importe quelle personne souhaitant s'amuser.

Vecteurs d'attaque

Tout se passe sur le web. Les utilisateurs peuvent (et doivent) manipuler des entrées pour envoyer des messages ou mettre à jour leurs profils. Des entrées malveillantes pourraient exploiter des vulnérabilités au sein de notre application comme celles citées plus haut.

Impacts techniques

Si des messages sont accédés par des personnes non autorisées, cela peut entraîner une perte de confidentialité.

Un accès non autorisé pourrait donner le droit à un attaquant de modifier, voire supprimer de l'information. Une perte d'intégrité peut donc en découler.

Une perte de disponibilité est également possible si la DB est altérée par exemple.

Impacts business

L'impact business serait faible. En effet, notre application n'est pas conçue dans un but lucratif et ne sera probablement pas maintenue par la suite. Si une telle application était mise en production dans le but d'offrir un service sérieux, cela pourrait être problématique car des clients pourraient stocker des informations personnelles voire confidentielles dans leurs messages. Une fuite de données ou des vulnérabilité pourrait entraîner une perte de confiance complète des utilisateurs.

Vulnérabilités, scénarios d'attaque et contre-mesures

XSS

Motivation

Ces failles pourraient permettre d'exécuter du code sur un client. Cela pourrait permettre d'usurper une session. L'idéal serait de viser un administrateur. Ainsi, en volant sa session, on pourrait élever nos privilèges.

Exploitation : démo basique

Le programme est vulnérable aux attaques XSS. Les balises HTML sont correctement interprétées, ainsi que les balises de script. C'est problématique car un attaquant pourrait envoyer un message malicieux à un administrateur. Lorsque l'admin se connecte, le script pourrait récupérer ses cookies et l'attaquant pourrait les rejouer et devenir donc administrateur.

Voici le message envoyé par un attaquant quelconque :

Création utilisateur | Modification utilisateur | Changer de mot de passe | Nouveau message | Déconnexion | Nom d'utilisateur : c.l

Boîte de réception

Destinataire:

Sujet:

Message:

Figure 1: Message malicieux

Voici le résultat dans la boîte de réception de la victime.

c.l

Test

2019-01-07 13:07:48

Figure 2: Boîte de réception

Lorsque la victime l'ouvre, voici le script est correctement exécuté.



Figure 3: Exécution XSS

Exploitation : vol de cookies

Essayons de récupérer les cookies de l'administrateur *steve.henriquet*. Pour cela, un attaquant peut simplement écrire le message suivant :

Figure 4: Message malveillant voleur de cookie

Le script contenu dans le message va simplement envoyer une requête contenant les cookies courant sur une serveur externe (ici *localhost:3000*). L'administrateur *steve.henriquet* ne se doute de rien en ouvrant son message.

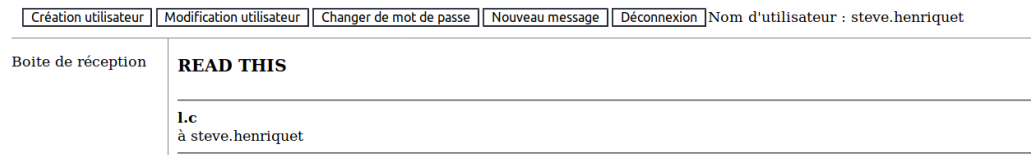


Figure 5: La faille n'est pas visible pour la victime

Finalement, si on se connecte sur notre serveur externe, on peut constater qu'un nouveau cookier contenant un id de session est apparu.

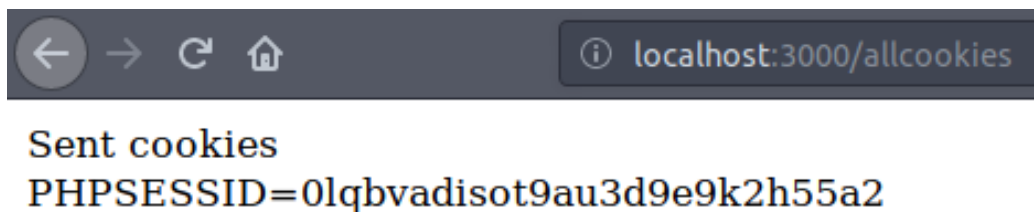


Figure 6: Le cookie volé

Ce cookie peut tout simplement être rejoué et l'attaquant peut endosser l'identité de l'administrateur *steve.henriquet* et donc augmenter ses privilèges.

Contre-mesures

Les entrées ont du être "sanitisées". Nous avons utilisé le filtre *FILTER_SANITIZE_STRING* dans les divers inputs accessibles.

```
$pass = filter_var ( $_POST["pass"], filter: FILTER_SANITIZE_STRING);
$passCheck = filter_var ( $_POST["passCheck"], filter: FILTER_SANITIZE_STRING);
```

Figure 7: Exemple d'assainisation

Injection SQL

D'après l'outil *SQLMap*, on peut déjà sortir quelques informations depuis la page de login, dont les tables de notre DB.

```
root@kali:~/Documents# sqlmap -r postLogin.txt --tables

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program.

[*] starting at 09:28:19

[09:28:19] [INFO] parsing HTTP request from 'postLogin.txt'
[09:28:19] [INFO] resuming back-end DBMS 'sqlite' module will
[09:28:19] [INFO] testing connection to the target URL
[09:28:19] [INFO] heuristics detected web page charset 'ascii'
sqlmap resumed the following injection point(s) from stored session:
Parameter: login (POST)
Type: boolean-based blind
Title: AND boolean-based blind - WHERE or HAVING clause
Payload: login=c.l' AND 7955=7955 AND 'jDbf'='jDbf&password=pass&type=login
Type: AND/OR time-based blind
Title: SQLite > 2.0 AND time-based blind (heavy query)
Payload: login=c.l' AND 2709=LIKE('ABCDEFG',UPPER(HEX(RANDOMBLOB(500000000/2)))) AND 'Trnr'='Trnr&password=pass&type=login
Type: UNION query
Title: Generic UNION query (NULL) - 4 columns
Payload: login=c.l' UNION ALL SELECT 88,88,88,'qvzzq' || 'UGCtFTa0aewYrdlGnzhtiEDBkRqPKjcqwRIqRjb' || 'qbvzq' -- ouZz&password=pass&type=login

[09:28:19] [INFO] the back-end DBMS is SQLite
web server operating system: Linux Ubuntu
web application technology: Nginx, PHP 5.5.9
back-end DBMS: SQLite
[09:28:19] [INFO] fetching tables for database: 'SQLite_masterdb'
Database: SQLite_masterdb
[3 tables]
+-----+
| Message |
| Messages |
| Personne |
+-----+

[09:28:19] [INFO] fetched data logged to text files under '/root/.sqlmap/output/172.17.0.2'

[*] shutting down at 09:28:19

root@kali:~/Documents#
```

Figure 8: Résultat de *SQLMap* sur la page de login

Motivation

En manipulant les certaines requêtes, il serait possible de gagner un accès non autorisé à l'application. Si l'on peut modifier les données contenues en base de données comme bon nous semble, il serait possible d'altérer des messages et des informations de comptes. Si l'on peut modifier les accès d'un compte ou simplement les afficher, il est possible d'augmenter ses privilèges.

Exploitation : se connecter sans mot de passe

Dans notre page de login, en essayant d'injecter du code simple, nous avons remarqué que le login se faisait tout de même. D'après notre code, nous avons découvert une injection SQL permettant de bypasser le login et de se connecter en tant que n'importe quel utilisateur sans connaître son mot de passe. La requête effectuée au login est la suivante :

```
1      SELECT Actif, User_id, Admin, Username FROM
      Personne WHERE Username='$name' AND Pass=
      '$password'
```

Puis le code suivant permet de vérifier si les credentials sont valides :

```
function checkLogin($arr) {
    $bool = true;
    $arrToSend = array();
    foreach ($arr as $row) {
        // Permet de savoir si l'utilisateur courant est actif ou
        non
        if (empty($row['Actif'])) {
            array_push($arrToSend, "error", "Compte_inactif!" . "
            <br/>");
            echo json_encode($arrToSend);
            $bool = false;
        }
        else {
            // Permet de setter les variables de session utile
            pour toute la connection
            $_SESSION["user_id"] = $row['User_id'];
            $_SESSION["username"] = $row['Username'];
            if (empty($row['Admin'])) {
                $_SESSION["admin"] = 0;
            }
            else {
                $_SESSION["admin"] = 1;
            }
            array_push($arrToSend, "body", manageLayout());
            echo json_encode($arrToSend);
            $bool = false;
        }
    }
    return $bool;
}
```

En écoutant la requête de la sorte :

```
1 SELECT Actif, User_id, Admin, Username FROM
   Personne WHERE Username=' $name ';
```

On peut se connecter sans avoir besoin de password ! Pour se faire, il suffit d'ajouter les deux caractères ' ; à la fin de son login et entrer un mode de passe random pour pas que le champs soit vide.

Login

Password

Figure 9: Se connecter comme *user.test* sans connaître son mot de passe

Exploitation : changer le mot de passe de n'importe quel utilisateur

Voici le code PHP associé à la modification d'un mot de passe :

```
else{
    $dbPass = '';
    $result = $file_db->query("SELECT_Pass_FROM_Personne_
        WHERE_User_id_=' " . $user_id . " '");
    foreach ($result as $row) {
        $dbPass = $row['Pass'];
    }
    if ($oldPass == $dbPass){
        $result = $file_db->query("UPDATE_Personne_SET_Pass_=
            ' " . $newPass . " ' WHERE_User_id_=' " . $user_id
            . " '");
        echo "Mot_de_passe_mis_a_jour_avec_succes";
    }
    else {
        echo "Mot_de_passe_errone!";
    }
}
```

Avec une injection SQL, peut modifier la deuxième requête afin de modifier le password d'un autre utilisateur. Si le nouveau password que l'on rentre est le payload suivant : *newpass' WHERE Username='victim.user' -*, alors l'utilisateur *victim.user* aura son mot de passe modifié en *newPass*. La partie *WHERE User_id ='' . \$user_id . ''*; de la requête originale n'est pas exécutée car commentée par *-*.

Contre-mesures

Tout comme pour les failles XSS, les entrées ont du être assainies. De plus, les query simples utilisées dans le code PHP ont été passées en prepared statements.

```
$result = $file_db->query( statement: "SELECT Username FROM Personne WHERE Username=" . $username . "'");
foreach ($result as $row) {
    $usernameDb = $row['Username'];
}
```

Figure 10: Avant d'utiliser des prepared statements il y avait juste de simples queries

```
$sth = $file_db->prepare( statement: "SELECT Username FROM Personne WHERE Username=:username");
$sth->execute(array(':username' => $username));
$result = $sth->fetchAll( fetch_style: PDO::FETCH_ASSOC);

foreach ($result as $row) {
    $usernameDb = $row['Username'];
}
```

Figure 11: Exemple de prepared statement

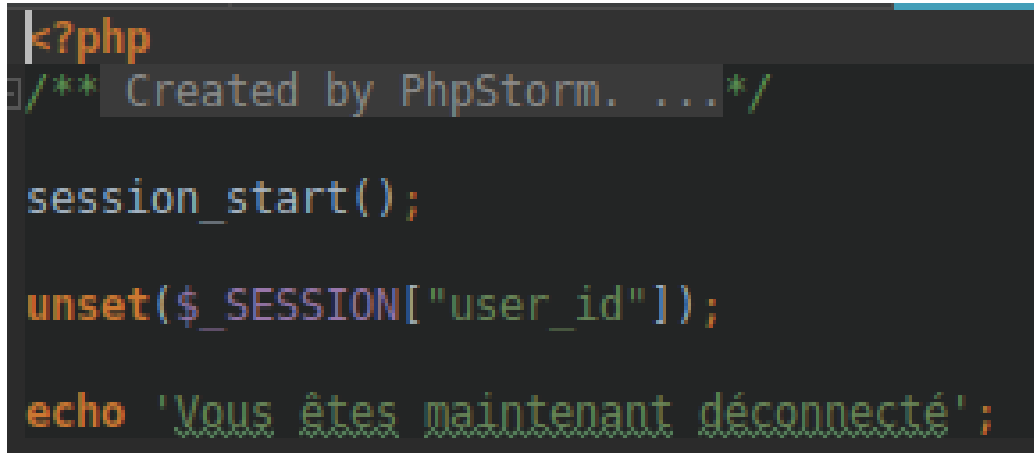
Mauvaise destruction des session

Motivation

Cette faille est une erreur d'implémentation. La motivation qu'un attaquant aurait à l'exploiter (pour autant qu'il en connaisse l'existence) serait simplement d'obtenir des informations qu'il n'est pas autorisé d'avoir.

Exploitation

Les sessions sont mal détruites. Potentiellement, après le login d'un admin, un utilisateur pourrait se retrouver à avoir accès à des informations réservées aux administrateurs, comme les layouts de modification des comptes.

A screenshot of a code editor with a dark background. The code is written in PHP and shows a function for destroying a session. The code includes a comment from PhpStorm, a call to session_start(), an unset operation on the user_id session variable, and an echo statement displaying a French message.

```
<?php
/** Created by PhpStorm. ... */

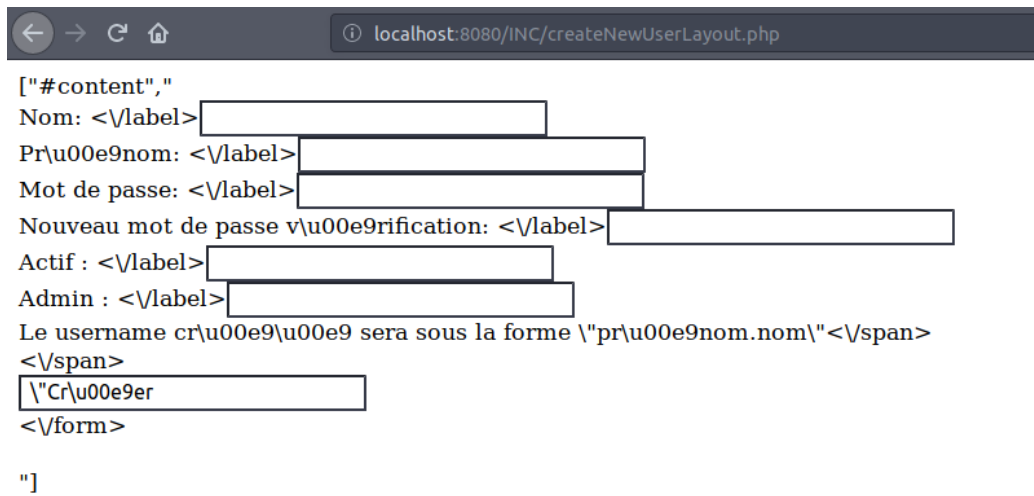
session_start();

unset($_SESSION["user_id"]);

echo 'Vous êtes maintenant déconnecté';
```

Figure 12: Mauvaise destruction de la session

Un administrateur était loggé puis, s'est déconnecté. Ensuite, un utilisateur essaie d'accéder à la page <http://localhost:8080/INC/createNewUserLayout.php>. On voit que des informations s'affichent alors qu'elles ne devaient pas !



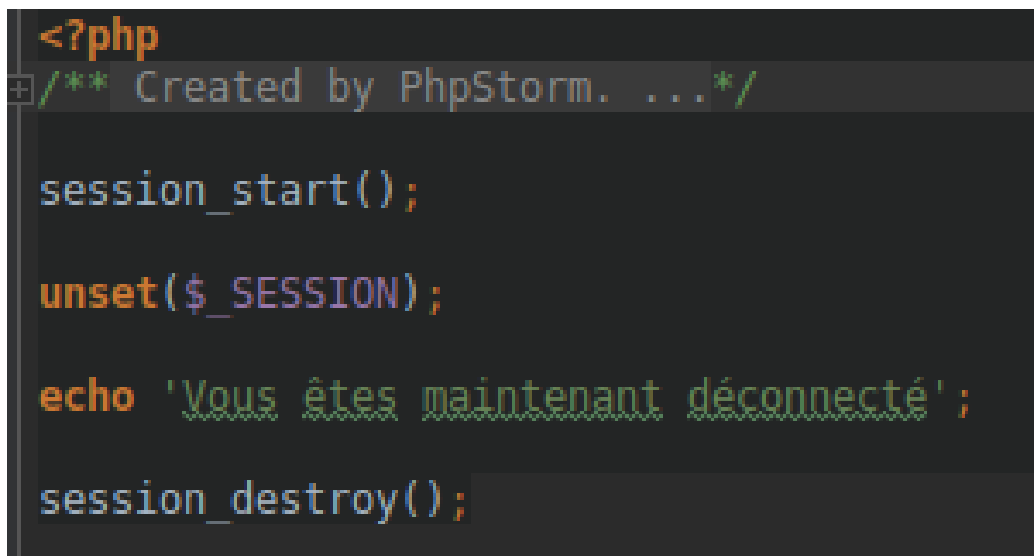
A screenshot of a web browser window showing a user creation form. The address bar indicates the URL is `localhost:8080/INC/createNewUserLayout.php`. The form contains several input fields with labels: "Nom:", "Prénom:", "Mot de passe:", "Nouveau mot de passe", "Actif:", and "Admin:". Below these fields, there is a line of text: "Le username cr\u00e9er sera sous la forme 'pr\u00e9nom.nom'", followed by a text input field containing the value "Cr\u00e9er". The form is enclosed in a container with a "content" attribute.

```
[ "#content", "  
  Nom: </label> <input type='text' />  
  Pr\u00e9nom: </label> <input type='text' />  
  Mot de passe: </label> <input type='password' />  
  Nouveau mot de passe v\u00e9rification: </label> <input type='password' />  
  Actif : </label> <input type='checkbox' />  
  Admin : </label> <input type='checkbox' />  
  Le username cr\u00e9er sera sous la forme 'pr\u00e9nom.nom'</span>  
  </span>  
  <input type='text' value='Cr\u00e9er' />  
</form>  
"]
```

Figure 13: Accès réservé aux admins

Contre-mesures

Désormais, les sessions sont bien détruites.



A screenshot of a code editor showing PHP code for session destruction. The code includes a comment from PhpStorm, followed by `session_start();`, `unset($_SESSION);`, `echo 'Vous êtes maintenant déconnecté';`, and `session_destroy();`.

```
<?php  
/** Created by PhpStorm. ... */  
  
session_start();  
  
unset($_SESSION);  
  
echo 'Vous êtes maintenant déconnecté';  
  
session_destroy();
```

Figure 14: Mauvaise destruction de la session

Accès au page sans login

Motivation

Un attaquant pourrait avoir accès à des fonctionnalités de l'application sans avoir à se connecter.

Exploitation

Il est possible d'effectuer des actions sur notre application sans passer par l'interface. En effectuant des requêtes via un terminal ou d'autres outils (tels que *Postman* par exemple), on pourrait accéder à des fonctionnalités sans avoir à se connecter. Voici comment envoyer un mail par exemple.

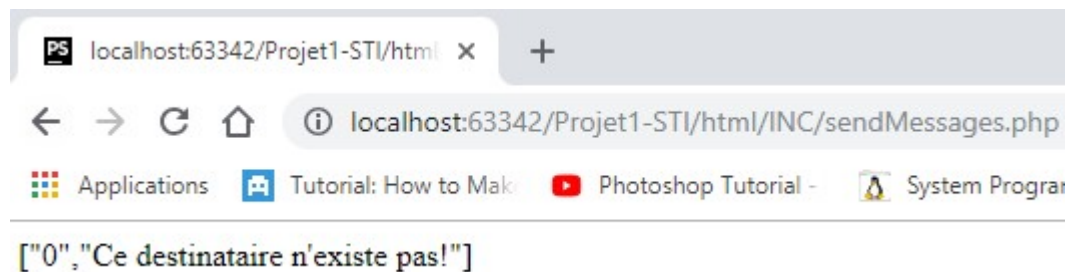


Figure 15: Accès à la page

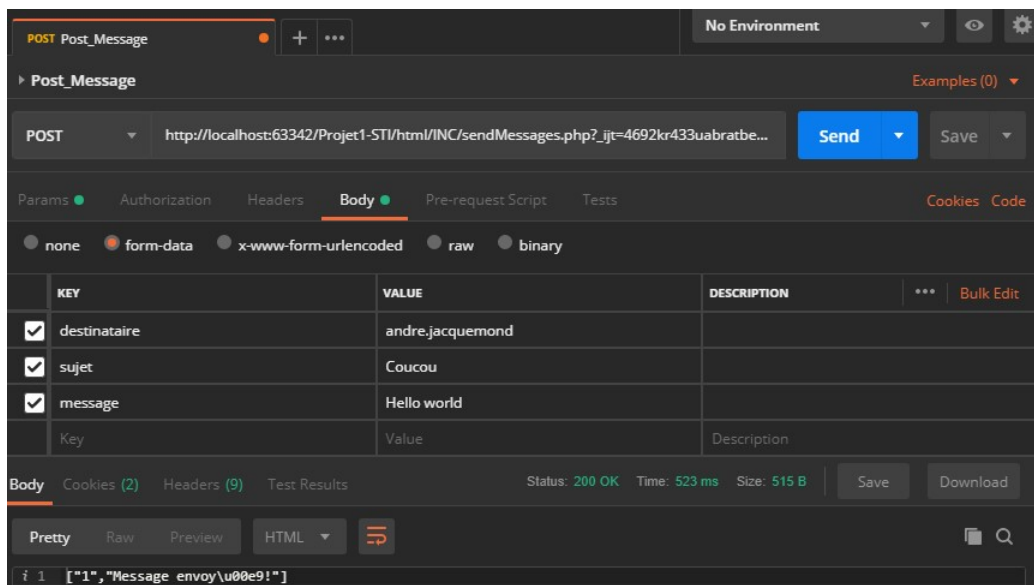


Figure 16: Requête Post réussi



Figure 17: Message reçu

Contre-mesures

Vérification de l'existence de la variable de session `$_SESSION["user_id"]`.

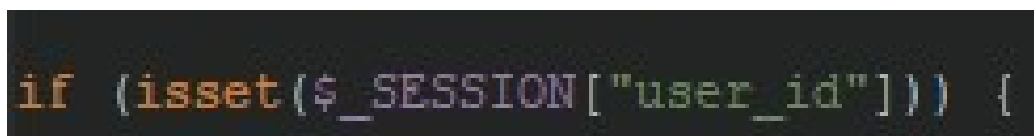


Figure 18: Protection à ajouter aux différentes pages

L'ajout de cette vérification s'effectue sur tous les fichiers à l'exception de index.php, login.php et des fichiers effectuant déjà la vérification par rapport au fait d'être administrateur

Spamming

Des auteurs malveillants pourraient programmer des bots qui enverraient des messages de manière abusives. Cela pourrait nuire aux performances de l'application et mener à un déni de service si l'application est vraiment surchargée.

Contre-mesure

Un input caché est rajouté. Les bots ne font pas attention aux styles. Si un bot arrive sur la page et remplit l'input caché, alors le bot sera détecté et démasqué.

```
.='<form action="javascript:login()">'
.'<label for="login">Login</label>'
.'<input type="text" id="login" name="login" placeholder="email" value="" required><br><br>'
.'<input type="text" id="login_spam" name="login_spam">'
.'<label for="password">Password</label>'
.'<input type="password" id="password" name="password" placeholder="password" value="" required><br><br>'
.'<input type="submit" value="submit">'
.'</form>'
.'<div class="msgError" id="msgErrorId"></div>'
.'<script>$("#login_spam").hide()/script>';
```

Figure 19: Protection anti-spam : HTML et JS

```

#login_spam, #message_spam {
    display:none;
}

```

Figure 20: Protection anti-spam : CSS

```

//in your php ignore any submissions that include this field
if(!empty($_POST['login_spam'])) {
    $arrToSend = array();
    array_push( &array: $arrToSend, var: "#msgErrorId", _: "die spamming bot") ;
    die(json_encode($arrToSend));
}

```

Figure 21: Protection anti-spam : PHP

Droits d'administrateurs retirés au logout

Si un enlève les droits d'administrateurs à un admin, la modification est appliquée qu'au moment où il se log la prochaine fois. Potentiellement, un administrateur connecté auquel on retire ses droits pourrait toujours entreprendre des actions réservées aux admins, ce qui n'est pas souhaitable.

Contre-mesures

Désormais chaque action commise par un administrateur est vérifiée. Avant d'être appliquée on vérifie que le user courant est effectivement administrateur dans la DB.

```

$isAdmin;
$userIdSession = $_SESSION["user_id"];

$stmt = $file_db->prepare( statement: "SELECT Admin FROM Personne WHERE User_id =:userId");
$stmt->execute(array(':userId' => $userIdSession));
$result = $stmt->fetchAll( fetch_style: PDO::FETCH_ASSOC);
foreach ($result as $row) {
    $isAdmin = $row['Admin'];
    $_SESSION['admin'] = $row['Admin'];
}

if($isAdmin) {

```

Figure 22: Vérification des actions admin

Problèmes non résolubles

- PHP 5.6
- Gestion des cookies de session

Conclusion

Cette deuxième partie de projet était très intéressante. La partie recherche de vulnérabilités et amusante à faire mais il est possible que d'autres failles subsistent encore. C'était un bon moyen de manipuler de nouveaux outils et aussi de manipuler des mécanismes fréquemment utilisé dans les applications web. La partie sécurisation nous montre qu'il est important de prévoir la sécurité dès le départ.