

**STI**

Projet 2

Groupe

*Ali Miladi*

*Julien Brêchet*

Professeur

*Abraham Rubinstein*

# Table des matières

[1 Table des matières 2](#_Toc503822240)

[2 Introduction 3](#_Toc503822241)

[2.1 Objectifs 3](#_Toc503822242)

[3 Description du système 3](#_Toc503822243)

[3.1 Objectifs du système 3](#_Toc503822244)

[3.2 Data Flow Diagram 4](#_Toc503822245)

[3.3 Définition du périmètre de sécurisation 4](#_Toc503822246)

[3.4 Hypothèses de sécurité 4](#_Toc503822247)

[3.5 Exigences de sécurité 5](#_Toc503822248)

[3.6 Enumération des actifs 5](#_Toc503822249)

[3.7 Rôles des utilisateurs 5](#_Toc503822250)

[4 Identification des menaces 6](#_Toc503822251)

[4.1 Scénario d’attaque 1 6](#_Toc503822252)

[4.2 Scénario d’attaque 2 7](#_Toc503822253)

[4.3 Scénario d’attaque 3 7](#_Toc503822254)

[4.4 Scénario d’attaque 4 8](#_Toc503822255)

[4.5 Scénario d’attaque 5 8](#_Toc503822256)

[4.6 Scénario d’attaque 6 8](#_Toc503822257)

[4.7 Scénario d’attaque 7 8](#_Toc503822258)

[4.8 Scénario d'attaque 8 8](#_Toc503822259)

[4.9 STRIDE 9](#_Toc503822260)

[5 Identification des contre-mesures 9](#_Toc503822261)

[5.1 Contre-mesure 1 9](#_Toc503822262)

[5.2 Contre-mesure 2 10](#_Toc503822263)

[5.3 Contre-mesure 3 10](#_Toc503822264)

[5.4 Contre-mesure 4 11](#_Toc503822265)

[5.5 Contre-mesure 5 11](#_Toc503822266)

[5.6 Contre-mesure 6 12](#_Toc503822267)

[5.7 Contre-mesure 7 13](#_Toc503822268)

[5.8 Contre-mesure 8 13](#_Toc503822269)

[6 Conclusion 14](#_Toc503822270)

# Introduction

Dans ce rapport, nous allons présenter l’application web de messagerie que nous avons développée dans le cadre du cours de STI.

Cette application consiste en une plateforme de messagerie en ligne et a été développée en deux phases. La première consistait à implémenter simplement les fonctionnalités de base de l’application sans pour autant s’intéresser à l’aspect sécuritaire. La deuxième phase se traduisait par la sécurisation de cette application avec l’utilisation de tous les outils vus en classe au travers de présentations faites par nos camarades et nous-mêmes.

Nous allons nous pencher dans ce document sur les aspects sécuritaires revus pour cette deuxième phase du projet.

## Objectifs

L’objectif de ce projet a été de maîtriser les principes de programmation web sécurisée dans le but de développer une application web sécurisée avec PHP.

Ce second projet fait suite au premier qui nous demandait de développer une application web de messagerie en ligne mais qui n’utilise pas de protocole SMTP. Il fallait reprendre ce dernier projet et le sécuriser.

Tout au long de ce travail, nous avons fait une analyse complète des menaces qui nous a permis de nous concentrer sur les vulnérabilités et les points vitaux à sécuriser dans l’application.

Cette sécurisation ne concerne que l’application web en elle-même mais en aucun cas l’OS hôte, le réseau LAN ou la configuration du serveur PHP. Nous considérons que ces éléments ont déjà été sécurisés au préalable.

# Description du système

Notre application est une plateforme de messagerie en ligne offrant une interface simple pour les utilisateurs dans le but de la communication interne d’une organisation (entreprise, école, administration, …).

Un utilisateur enregistré dans la base de données de l’application doit pouvoir envoyer des messages à n’importe quel autre utilisateur inscrit. Il sera aussi capable de consulter tous les messages qui lui sont destinés et d’y répondre. D’autre part, il peut modifier son mot de passe pour le login.

Un utilisateur administrateur, en plus de ce que peut faire un utilisateur normal, est en mesure de gérer les comptes des autres utilisateurs (dont administrateurs) inscrits dans la base de données. Il a la possibilité de consulter la liste de tous les utilisateurs avec des détails sur leurs comptes. Il peut aussi modifier n’importe quel profil d’utilisateur, à savoir changer le mot de passe, l’identifiant, le type de profil (admin ou régulier) et activer ou désactiver le compte.

## Objectifs du système

L’objectif primaire de notre application est de permettre aux utilisateurs de communiquer aisément au sein de leur organisation.

Nous n’avons pour le moment aucun objectif financier. En effet, l’application est en l’état à but non lucratif visant seulement à banaliser la communication entre ses utilisateurs. Néanmoins, des fonctionnalités plus évoluées et des améliorations pourraient être apportées dans le futur et seraient alors inclues dans un paquet payant « Premium » en plus de l’application de base.

D’autre part, l’application aura pour but de gagner en réputation et notoriété pour tous les types d’organisations. L’objectif est de la rendre accessible, pratique, user-friendly et suffisamment sécurisée pour gagner la confiance des utilisateurs.

## Data Flow Diagram



## Définition du périmètre de sécurisation

Pour ce projet, nous allons sécuriser l’application web en elle-même. Nous allons protéger les bases de données et les données privées des utilisateurs contre les injections SQL.

Nous allons aussi protéger l’application contre les injections de code et les attaques XSS.

La validation des champs des formulaires des utilisateurs, l’imposition de la vérification du mot de passe ainsi que la mise en place d’une politique de mots de passe forts vont être introduites.

## Hypothèses de sécurité

Nous considérons que l’application devra idéalement tourner sur un serveur de l’organisation. Elle pourra aussi très bien tourner sur une station de travail sur laquelle un serveur PHP est installé.

Nous faisons dès lors des hypothèses de sécurité quant à l’infrastructure physique de l’organisation utilisatrice de l’application ainsi que le réseau local.

Nous assumons que :

* Le serveur (éventuellement workstation) sur lequel tourne l’application est de confiance. Il aura toujours les dernières mises à jour de sécurité de l’OS et de PHP (actuellement PHP 7.2).
* Les systèmes d’exploitation des membres de l’organisation (utilisateurs de l’application) sont de confiance et ayant toujours les dernières mises à jour de sécurité, antivirus actif, pare-feu, … .
* Le réseau local de l’organisation est bien protégé par des pare-feu, les dispositifs nécessaires pour un minimum de sécurité du LAN devraient être fournis.
* Les administrateurs système/réseau sont des personnes de confiance.

## Exigences de sécurité

Quelques exigences de sécurité ont été fixées pour notre application :

* Le contenu des messages doit être intègre et donc non modifiable. Un message ne doit pas être changé en cours de route.
* Un utilisateur ne doit pouvoir lire que les messages qui lui sont destinés.
* L’application n’est accessible qu’aux membres inscrits et enregistrés dans la base de données. Aucune page de l’application à part la page de login n’est accessible à n’importe quel utilisateur non authentifié. Il faut donc s’authentifier avant de pouvoir utiliser l’application.
* Un compte de type utilisateur ne peut pas accéder à certaines pages qui sont uniquement accessibles avec un compte de type administrateur.
* Les informations critiques des utilisateurs (identifiant et mot de passe) doivent être privées et protégées.
* L’application doit être disponible à 99% du temps.

## Enumération des actifs

Pour notre application, nous pouvons identifier quelques actifs ayants une haute valeur. Sans ceux-ci, l’application ne pourra pas fonctionner et sans solution de backup et de protection adaptée, nous ne pouvons pas les reconstruire.

* La base de données des utilisateurs : doit être protégée en intégrité et confidentialité. En cas de corruption des comptes utilisateur ou vol d’identités, nous perdons la confiance des clients ainsi que de la notoriété.
* La base de données des messages : doit être également protégée en intégrité et confidentialité. Dans le cas de vol de messages ou de modification de leurs contenus, la réputation de l’application serait en danger. D’autre part, dans le cas où la base de données est corrompue ou effacée, et qu’elle n’a pas été backupée, les utilisateurs perdraient à jamais leurs messages.
* L’infrastructure de l’application : le serveur (et éventuellement le load balancer dans le cas où l’on déploierait pour une grande organisation ayant besoin d’une très haute disponibilité et de redondance) doit être protégé en intégrité et doit fournir une haute disponibilité. Une coupure du serveur ou corruption de celui-ci nuirait à la réputation de l’application.

## Rôles des utilisateurs

Nous pouvons distinguer deux rôles d’utilisateurs bien distincts comme énoncé plus haut dans la description du système :

* Les utilisateurs réguliers : Ceux qui ont accès aux fonctionnalités de base. Ils peuvent envoyer/recevoir et supprimer des messages ainsi que changer de mot de passe.
* Les administrateurs : En plus des fonctionnalités de base des utilisateurs réguliers, les administrateurs ont aussi des fonctionnalités supplémentaires de gestion des utilisateurs. Ils peuvent en créer, avoir la liste de tous les utilisateurs inscrits avec les détails ainsi que modifier les détails des comptes de ces utilisateurs (nom d’utilisateur, mot de passe, type de compte, activer/désactiver un compte).

# Identification des menaces

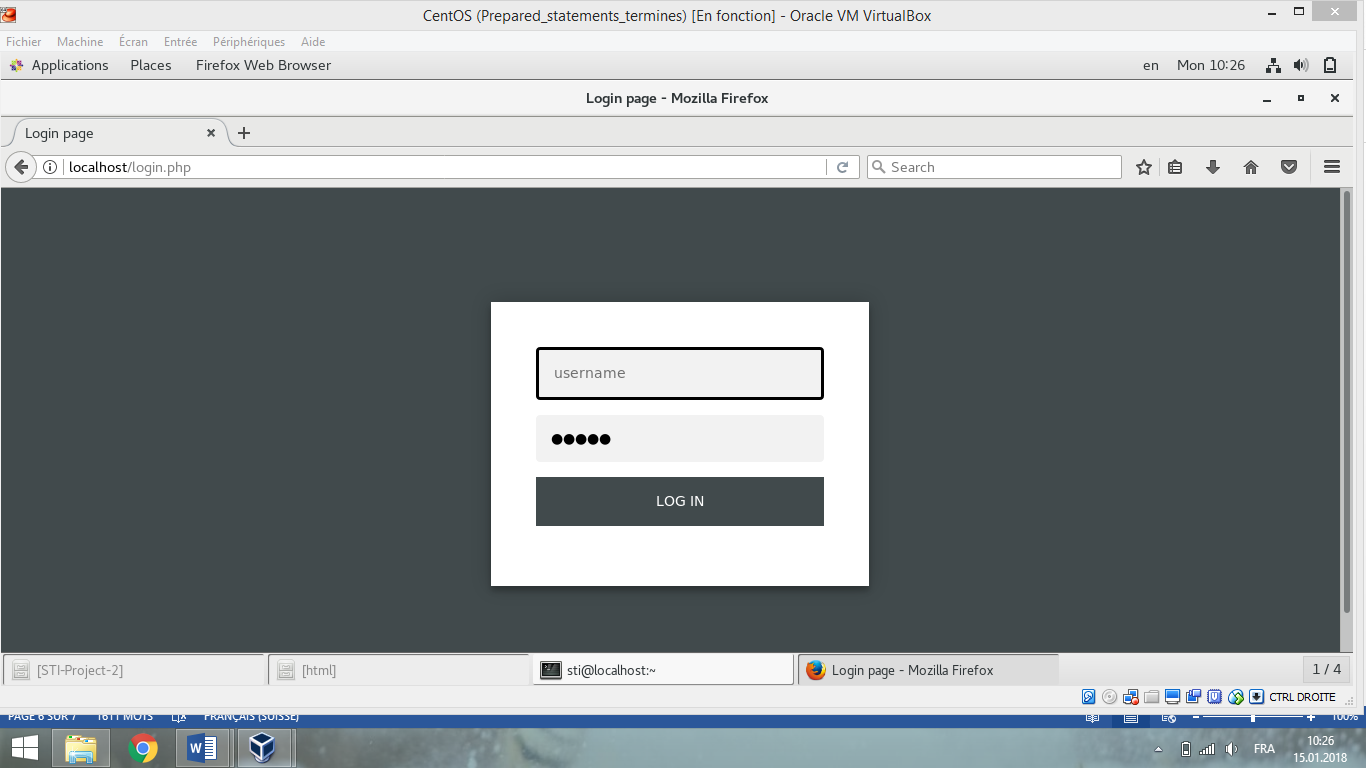
Plusieurs sources potentielles d’agression de l’application ont été identifiées :

* Hackers/script-kiddies :
  + Motivation : amusement, reconnaissance et gloire parmi les hackers
  + Cible : tous les actifs de l’application
  + Potentialité : haute
* Cybercrime :
  + Motivation : vol d’identités, informations, chantage, gain d’argent, …
  + Cible : bases de données de l’application et cookies des utilisateurs
  + Potentialité : moyenne-haute
* Concurrents :
  + Motivation : vol du code source et de la logique métier de l’application
  + Cible : code source de l’application
  + Potentialité : faible
* Utilisateurs malins :
  + Motivation : vol d’identités, vols d’informations compromettantes des autres utilisateurs et de l’application
  + Cible : tous les actifs de l’application web
  + Potentialité : faible-moyenne

Dans les points ci-dessous, nous présentons quelques scénarios d’attaques possibles.

## Scénario d’attaque 1

Notre application de messagerie utilise des requêtes SQL pour aller chercher les informations dont elle a besoin. Par exemple, lorsqu’un utilisateur se logue sur l’application, cette dernière va effectuer une requête SQL sur une base de données SQLite contenant toutes les données relatives à notre application web. Elle va effectuer une requête SQL pour récupérer les informations de l’utilisateur qui se connecte et vérifier si le username existe et que le mot de passe entré soit correct.

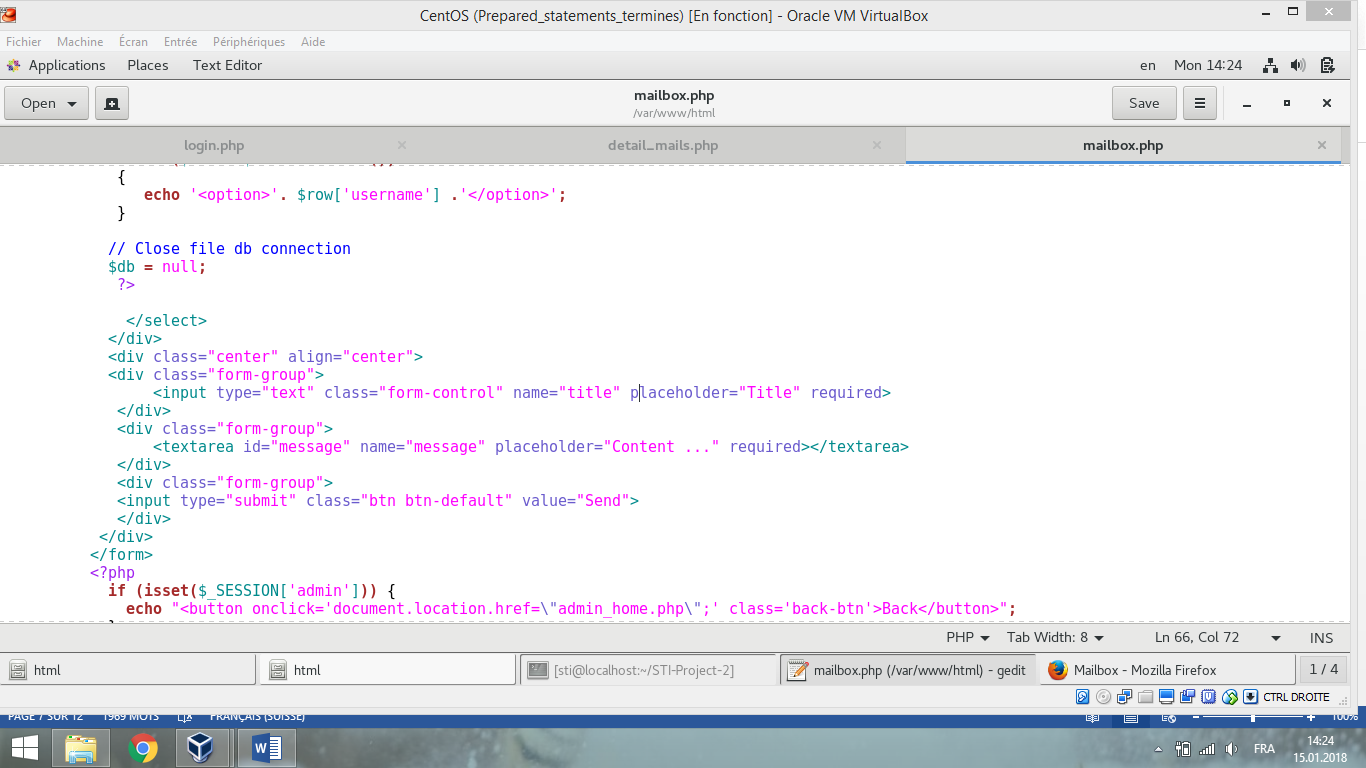


Au lieu d’entrer un nom d’utilisateur dans le champ « username », un attaquant pourrait injecter un code SQL lui retournant par exemple tous les usernames stockés dans la base de données ainsi que leurs mots de passe. L’attaquant pourrait ainsi récupérer les informations de tous les comptes de l’application et donc se connecter avec n’importe quel compte utilisateur qu’il a dérobé.

On appelle ce genre d’attaques des **injections SQL**.

## Scénario d’attaque 2

Un utilisateur vicieux décide d’inspecter le contenu HTML de l’une de nos pages web pour supprimer un attribut « required » afin de ne pas devoir forcément insérer une valeur dans un certain champ. Par exemple, il décide d’inspecter la page de création d’un nouveau message (mailbox.php) et d’enlever l’attribut *required* pour le champ du titre du message. Ainsi, il peut créer des messages sans titre. Selon la politique de notre application, chaque message doit avoir un titre, donc la cohérence des données stockées n’est plus respectée.



## Scénario d’attaque 3

S’il n’existe pas de politique pour forcer l’utilisation de mots de passe forts, certains utilisateurs vont mettre des mots de passe très simples tels que le nom de leur animal de compagnie, le nom d’un de leurs enfants ou des mots communs trouvables dans un dictionnaire de brute force.

Un attaquant muni d’un logiciel de brute force pourra ainsi plus ou moins facilement cracker ou même deviner en se renseignant sur la personne (social engineering) les mots de passes des utilisateurs négligents.

## Scénario d’attaque 4

Un attaquant arrive à pirater un compte utilisateur et insère un script dans les messages qu’il envoie aux autres utilisateurs. Chaque utilisateur ouvrant le message malicieux (qui est stocké dans la database) est infecté par le script de l’attaquant.

Un genre fréquent d’attaques de ce type est **l’attaque XSS** (Cross-Site Scripting). Elle exploite les failles permettant l’injection de code JavaScript exécutable par le navigateur.

## Scénario d’attaque 5

Un attaquant sniffe le réseau avec un outil tel que Wireshark pour tenter d’intercepter et collecter des données sensibles telles que les mots de passe entrés par les utilisateurs lorsque ceux-ci se connectent à l’application de messagerie.

## Scénario d’attaque 6

Un utilisateur tente de modifier l’ID de la variable « *message\_id* » dans l’url du navigateur afin de pouvoir accéder à des mails qui ne lui sont pas destinés.

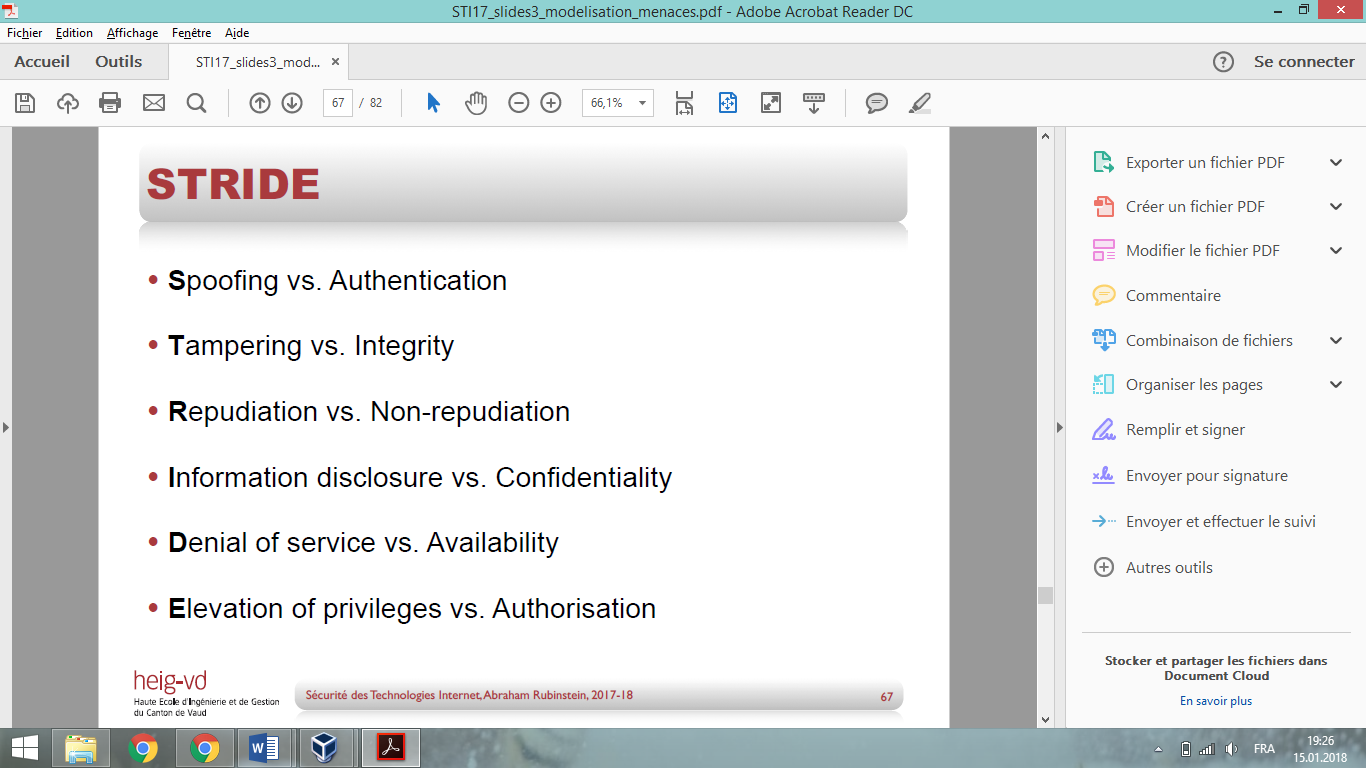
## Scénario d’attaque 7

Un administrateur décide de créer un nouveau compte pour un utilisateur avec un même username qu’un utilisateur déjà existant. Le nouvel utilisateur sachant cela, il va tenter de changer son mot de passe (de manière automatisée à l’aide d’un script) jusqu’à ce qu’il tombe sur le bon mot de passe. Il pourrait savoir que c’est le bon mot de passe car l’application lui indiquerait que la combinaison username / password existe déjà.

## Scénario d'attaque 8

Pour une quelconque raison, un script malicieux a été enregistré dans un champ de la base de données. Tous les utilisateurs allant lire ce champ vont exécuter malgré eux le script.

## STRIDE



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Composants | S | T | R | I | D | E |
| Application web | **x** | **x** | **x** | **x** | **x** | **x** |
| DB utilisateurs | **x** | **x** | **x** | **x** | **x** | **x** |
| DB messages | **x** | **x** | **x** | **x** | **x** | **x** |
| Administrateur | **x** |  | **x** |  |  |  |
| Utilisateur régulier | **x** |  | **x** |  |  | **x** |
| Agent externe | **x** |  | **x** |  |  | **x** |
| Opérations de Login | **x** | **x** |  | **x** | **x** |  |
| Opérations CRUD |  | **x** | **x** | **x** | **x** |  |

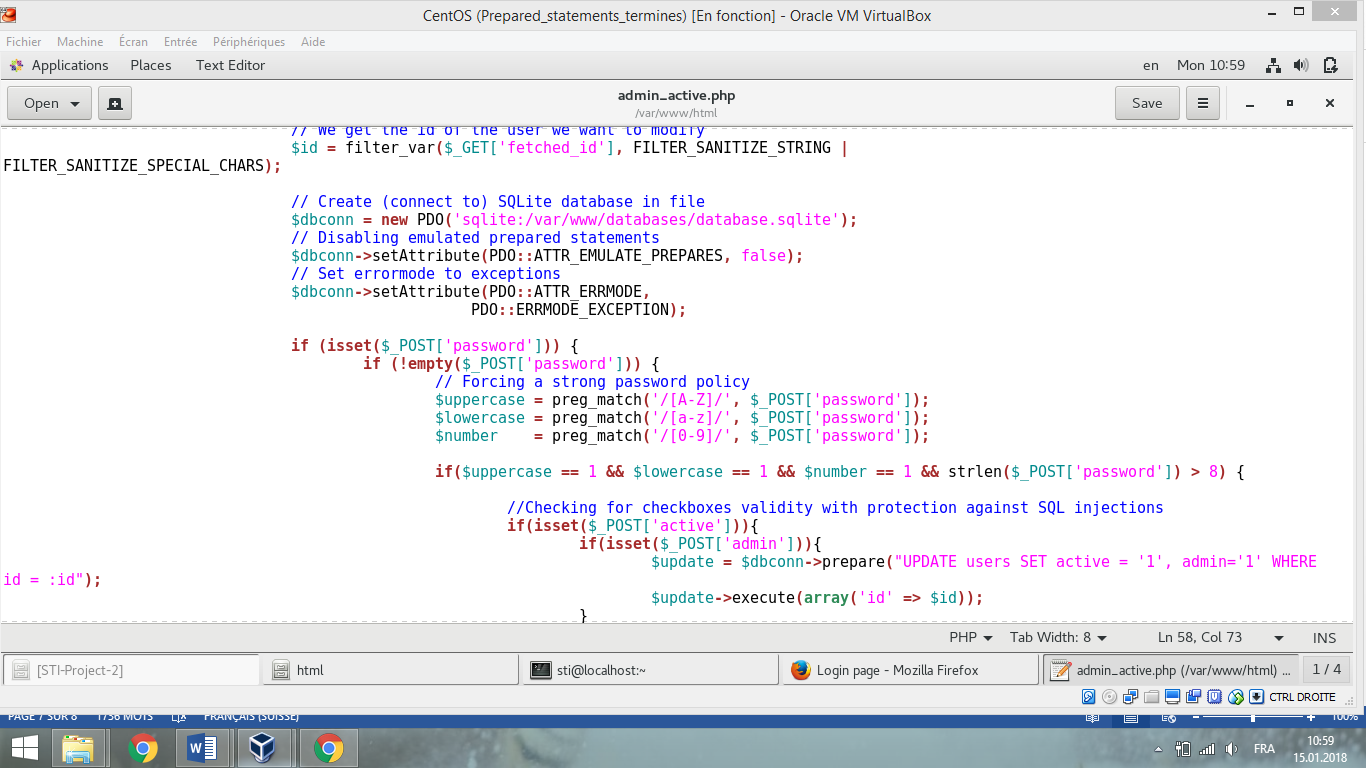
# Identification des contre-mesures

## Contre-mesure 1

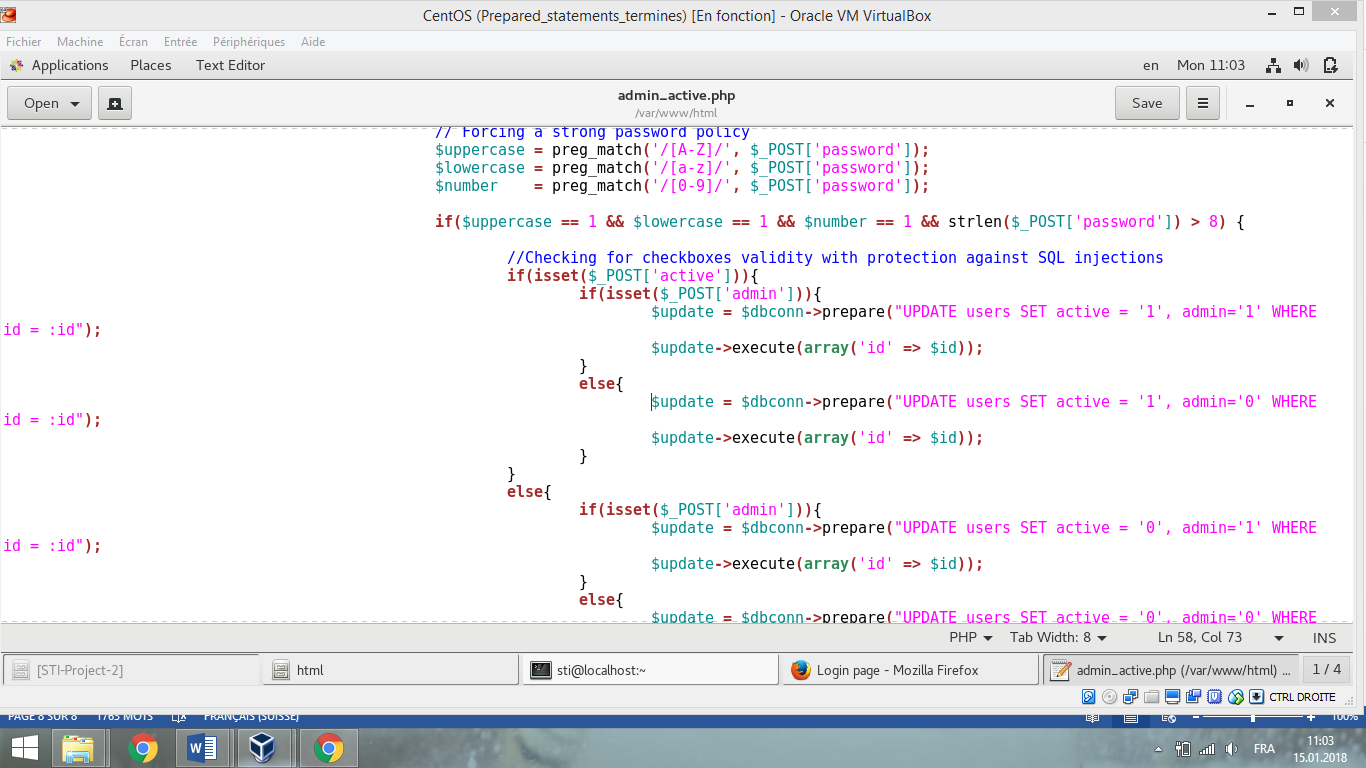
Pour se protéger des injections SQL, il faut utiliser des **« prepared statements »** (ou déclarations préparées en français).

En utilisant des prepared statements, la requête et les données sont envoyées à la base de données de façon séparée. L’attaquant ne peut plus mixer le code SQL et les données pour modifier le code à sa guise et ainsi pouvoir injecter des codes SQL malicieux.

Sur chaque page PHP où l’on doit se protéger contre des injections SQL, il faut tout d’abord désactiver les « emulated prepared statements ».



Ensuite, il faut préparer la requête (**->prepare**) pour ensuite l’exécuter en matchant les variables de la requête avec les variables PHP (**->execute(array(‘varSQL’ => $varPHP)**).

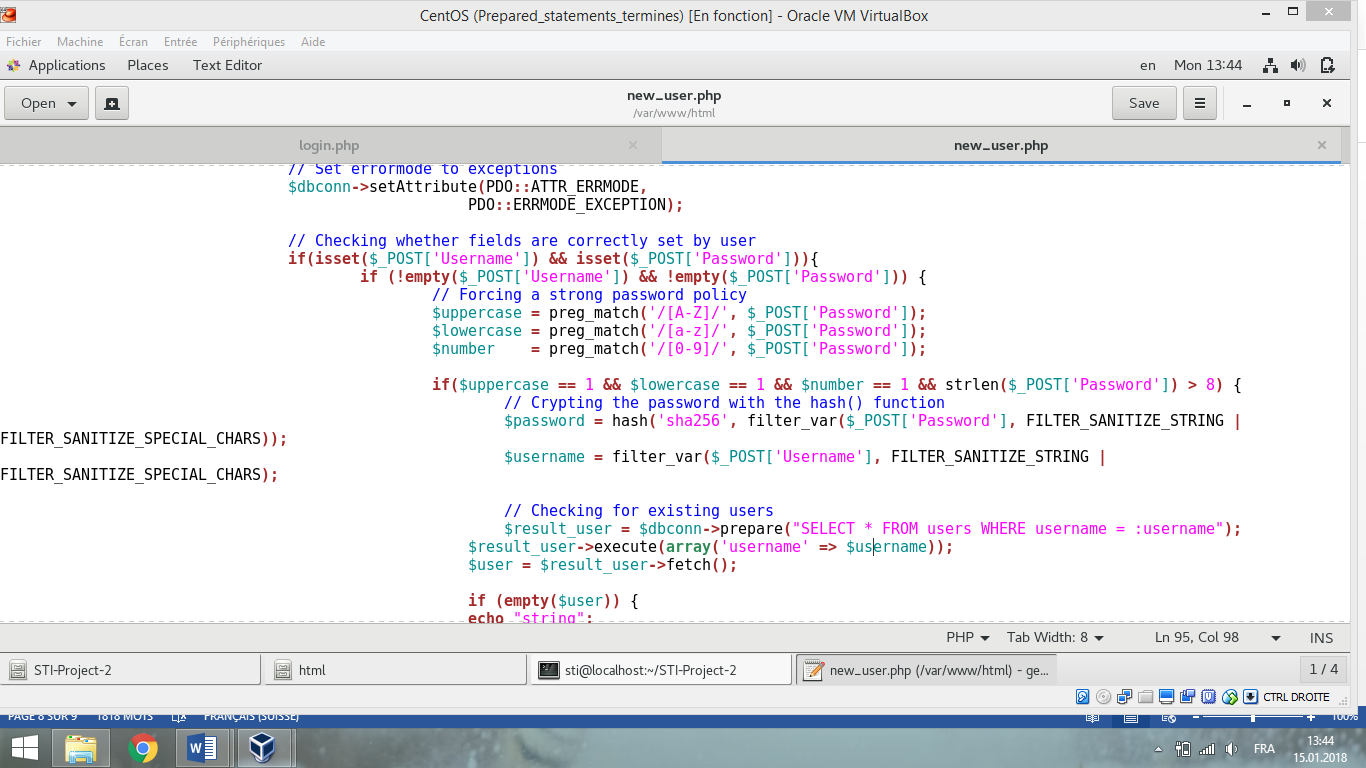


## Contre-mesure 2

Il faut empêcher les utilisateurs et les attaquants de contourner la politique des champs obligatoires, sans cela la cohérence des données stockées ne serait plus respectée.

Pour ce faire, pour empêcher de contourner les vérifications du côté client, il faut aussi effectuer les contrôles au niveau du serveur.

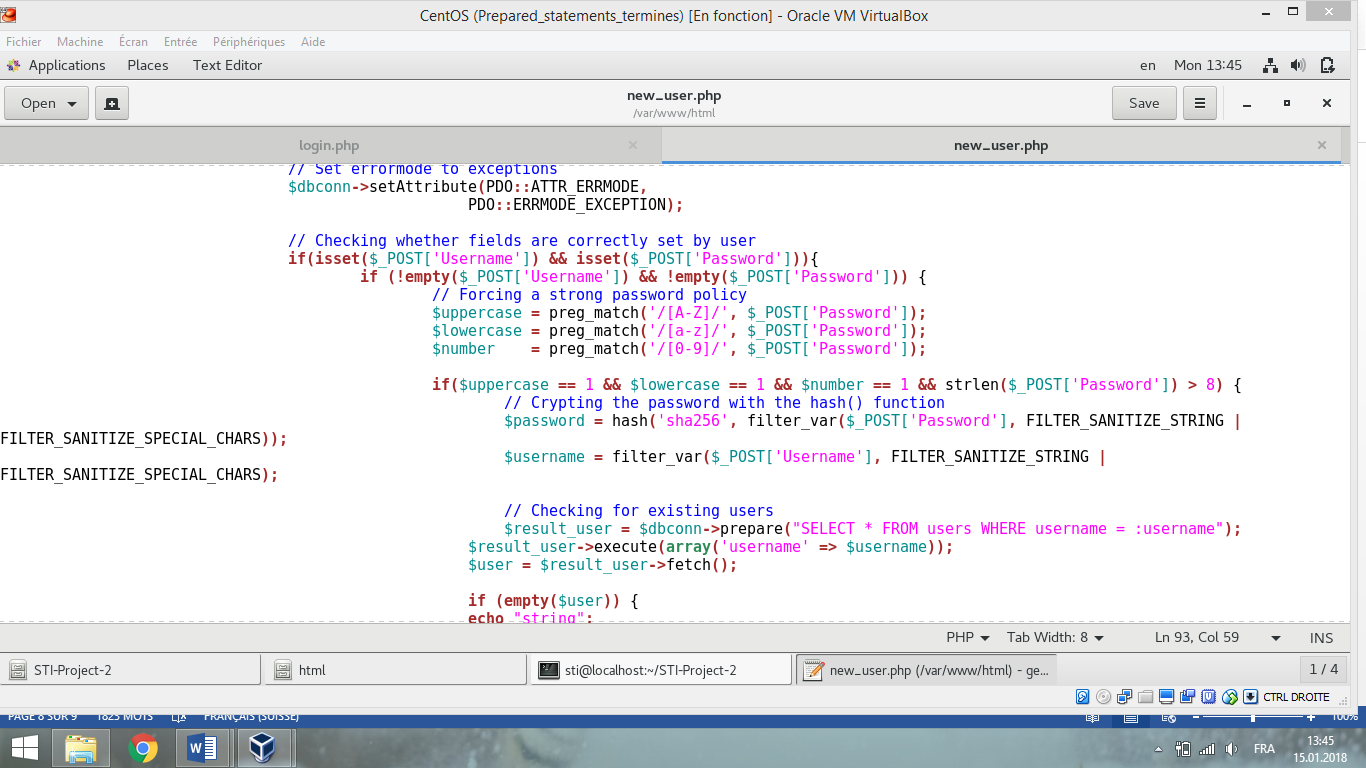
Nous nous protégeons de cela en vérifiant que les champs « required » sont bien settés (***isset***) dans la session PHP et qu’ils ne sont pas vides (***not empty***).



## Contre-mesure 3

Il faut instaurer une politique de mots de passe forts pour forcer les utilisateurs à ne pas mettre de mot de passe trop facile à deviner ou à brute forcer.

Nous avons décidé d’instaurer une politique forçant l’utilisateur à avoir un mot de passe d’au moins 8 caractères et composé d’au moins une majuscule, une minuscule et un chiffre.



Ainsi, le nombre de possibilités de mots de passe s’élève au minimum à (26 + 26 + 10)8. Ce qui équivaut à ~2.18 \* 1014 possibilités.

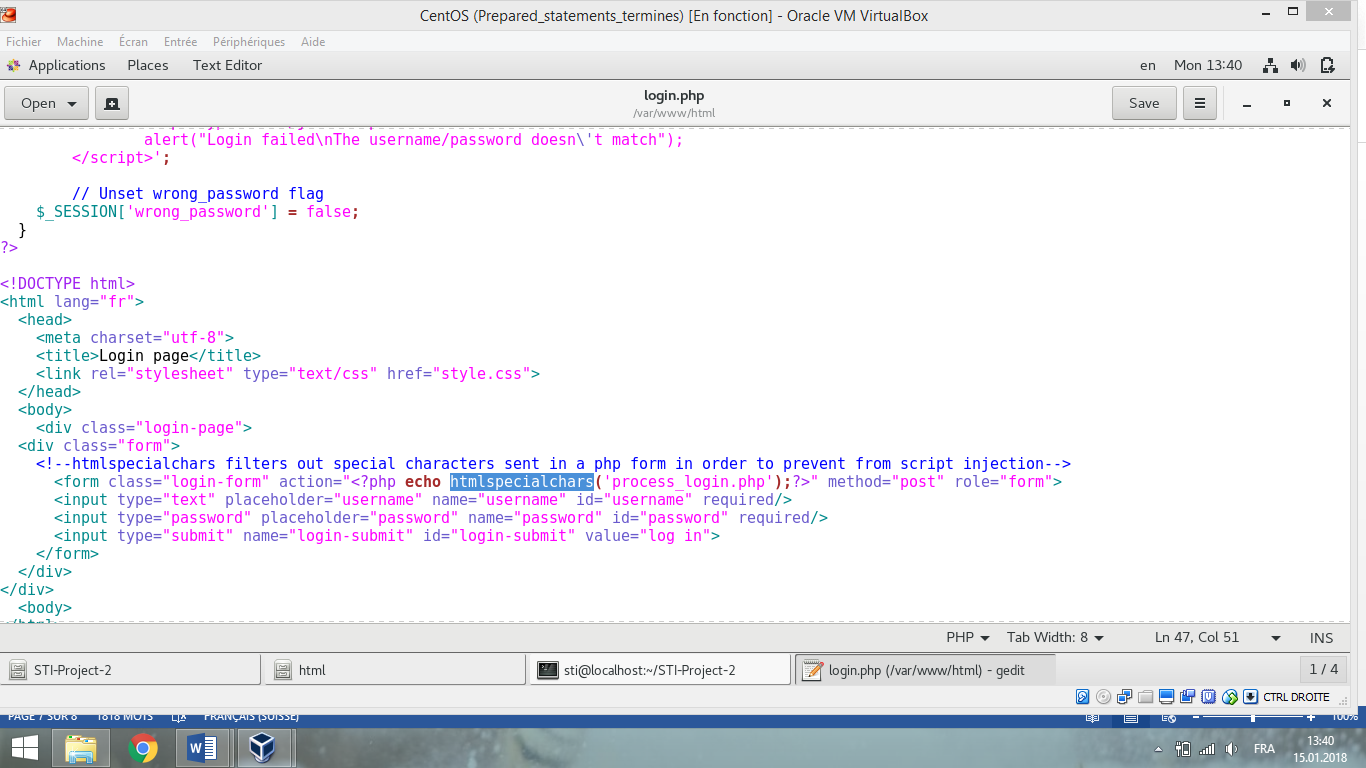
Il y a néanmoins une faiblesse dans notre politique de mots de passe forts. Nous ne vérifions pas si l’utilisateur a inséré un nom commun ou un nom propre à l’intérieur de son mot de passe. Par exemple, il est autorisé à entrer le mot de passe « JohnnyHalliday1 », « Petitchat18 » ou « Scorpion29 ».

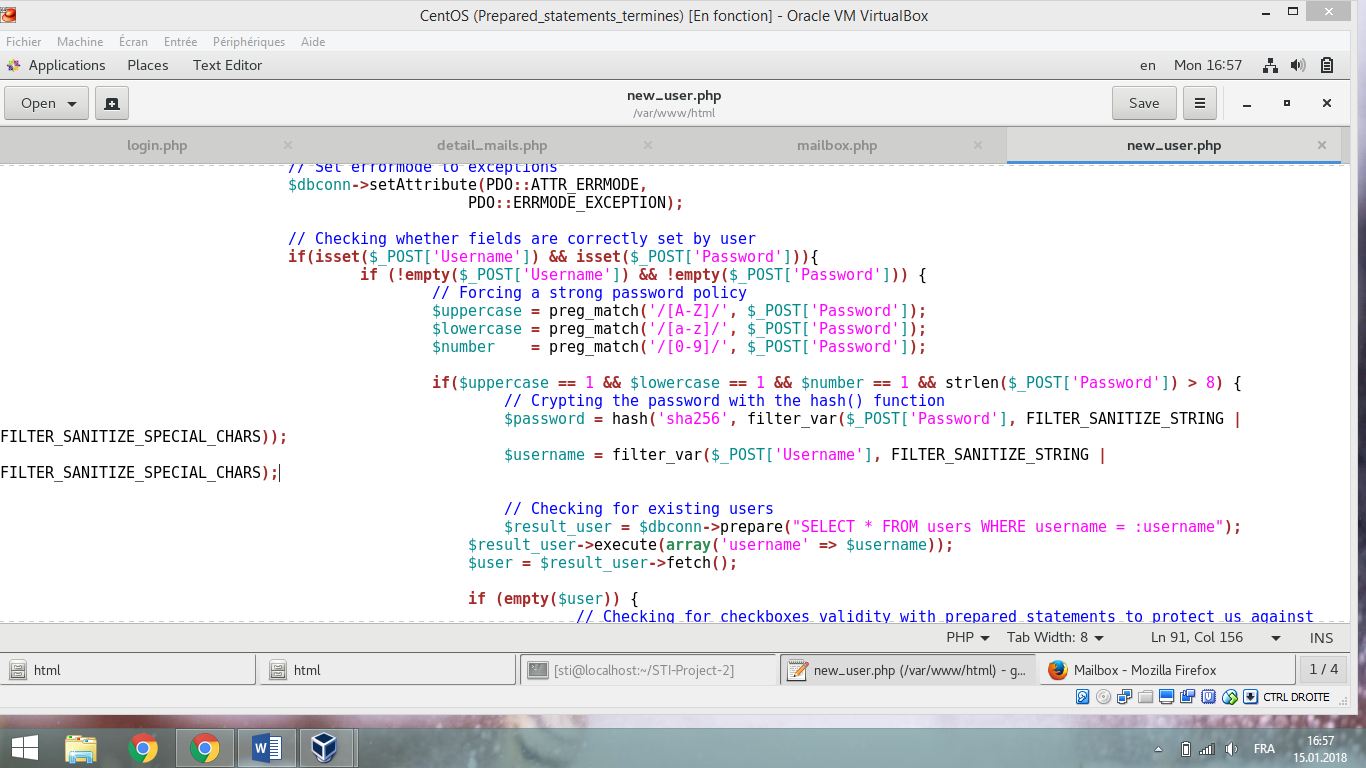
Mais le fait d’ajouter des chiffres et des majuscules obligatoires ainsi qu’une longueur minimale de 8 caractères est déjà une bonne protection. Et il ne faut pas non plus tomber dans l’excès et établir une politique de mots de passe trop contraignante comme par exemple avoir minimum 16 caractères dont 5 majuscules, 3 chiffres et 3 caractères spéciaux tels que $ ou #. Les utilisateurs perdraient facilement leurs mot de passe ou bien les noteraient sur des petits papiers qu’ils laisseraient sur leur place de travail. Une politique trop contraignante pourrait s’avérer contre-productive.

## Contre-mesure 4

Pour se protéger contre les attaques XSS et l’exécution de scripts malicieux, il faut filtrer les insertions de scripts dans les champs des formulaires.

Pour ce faire, nous utilisons la fonction ***htmlspecialchars()*** pour convertir les caractères spéciaux en entités HTML et la fonction **filter\_var(maVariable, FILTER\_SANITIZE\_STRING | FILTER\_SANITIZE\_SPECIAL\_CHARS)** pour enlever tous les tags HTML dans une chaîne de caractères.





Ainsi, si un script est inséré dans un champ de formulaire, il **sera interprété en tant que simple texte** et sera **enregistré aussi en tant que texte**. Il ne pourra donc pas être interprété et exécuté lorsque quelqu’un va lire le champ dans lequel se trouve le script.

## Contre-mesure 5

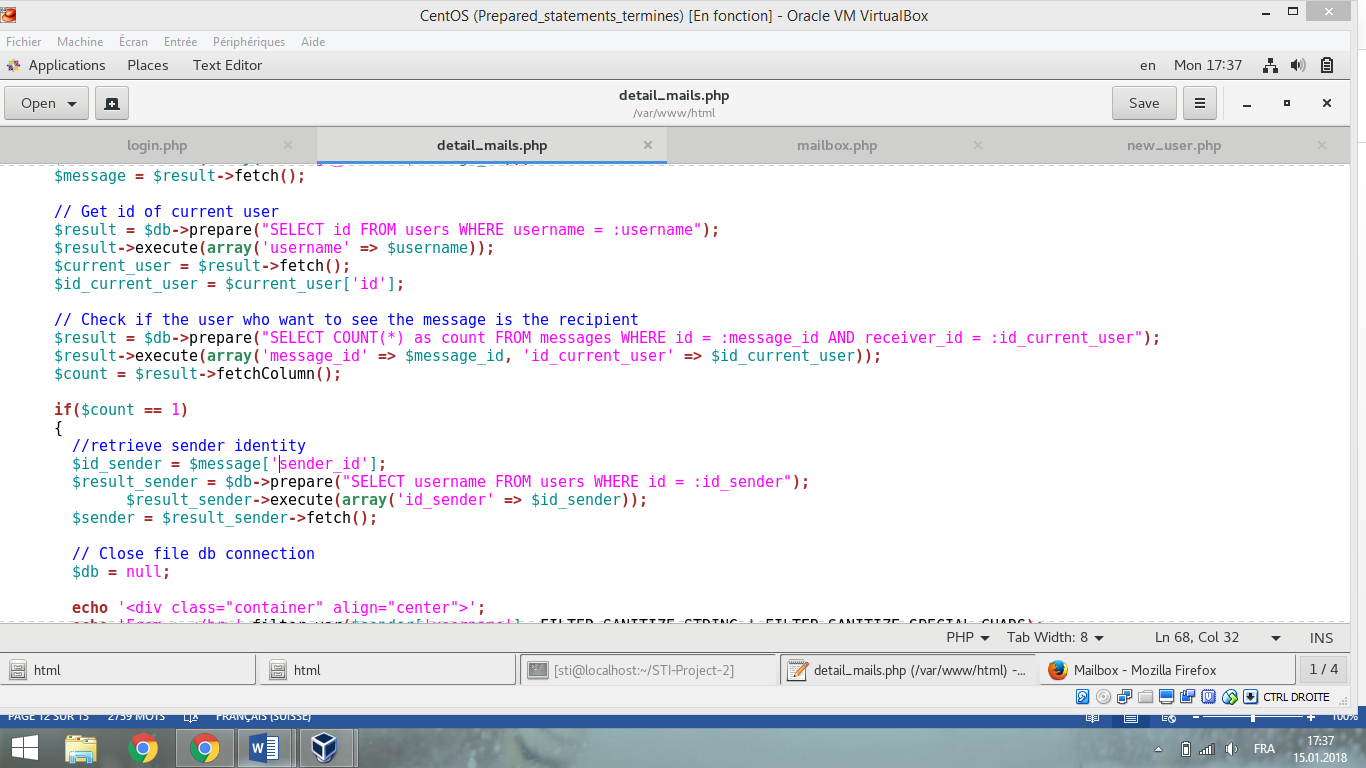
Il faut chiffrer le mot de passe pour qu’il ne soit pas envoyé en clair sur le réseau. Pour cela, nous utilisons l’algorithme de hachage ***SHA-256*** qui est considéré comme bon. Lorsqu’un compte est créé, le mot de passe du nouvel utilisateur est haché avec *SHA-256* et stocké dans la base de données SQLite. Idem lorsqu’un changement de mot de passe a lieu. Le nouveau mot de passe est toujours haché avec SHA-256 et est remplacé dans la base de données.

Quand un utilisateur tente de se connecter, le mot de passe qu’il entre est haché avec SHA-256 et est ensuite comparé avec le mot de passe stocké dans SQLite. Si les deux mots de passe sont identiques, l’utilisateur est authentifié et une session PHP est créée. Si les mots de passe ne correspondent pas ou si le username n’existe pas, une erreur est retournée et l’utilisateur est renvoyé à la page de login (login.php).

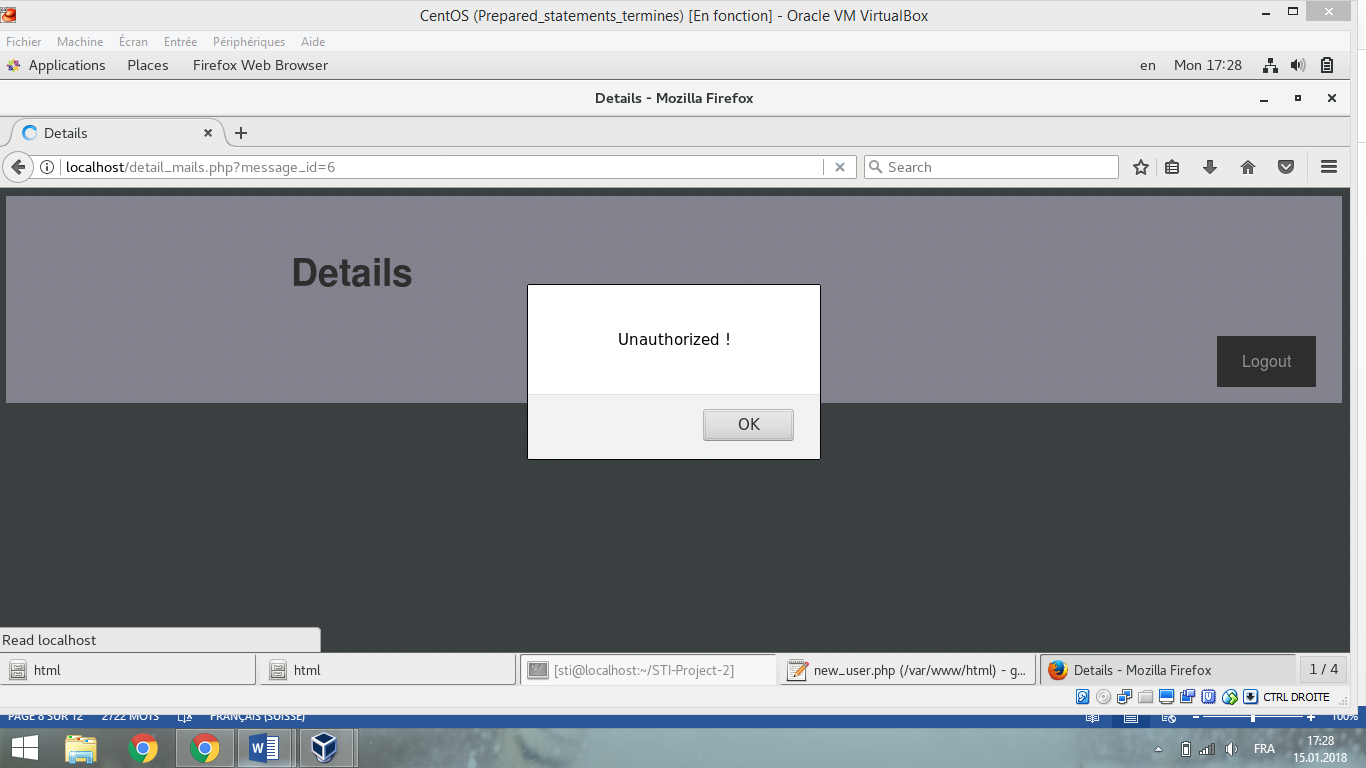
Il n’y a pas de sel lors de la génération du mot de passe. Un attaquant peut utiliser des Rainbow Tables pour tenter de découvrir à quel mot de passe correspond le hash qu’il a réussi à sniffer sur le réseau.

## Contre-mesure 6

Pour garantir la confidentialité des conversations de nos utilisateurs, il ne faut pas que l’on puisse accéder à des messages qui ne nous concernent pas. Pour ce faire, on récupère l’ID de l’utilisateur et on vérifie que c’est bien lui le destinataire du message (*receiver\_id = :id\_current\_user*).

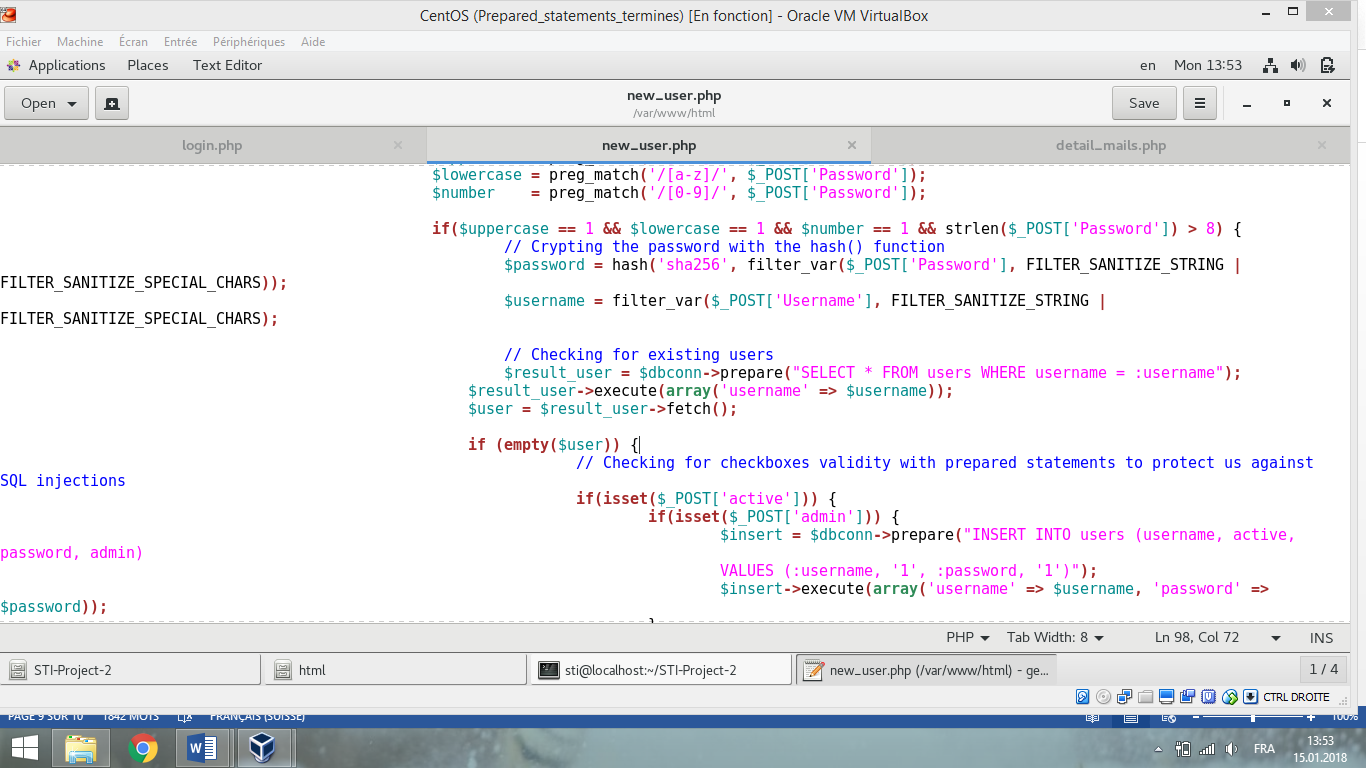


Si un utilisateur tente de modifier la valeur de *message\_id* avec un ID d’un message dont il n’est pas le destinataire, il reçoit en retour un message d’erreur lui indiquant qu’il n’a pas le droit d’accéder à ce message et il est ensuite redirigé sur sa boîte de réception (list\_received\_mails.php).



## Contre-mesure 7

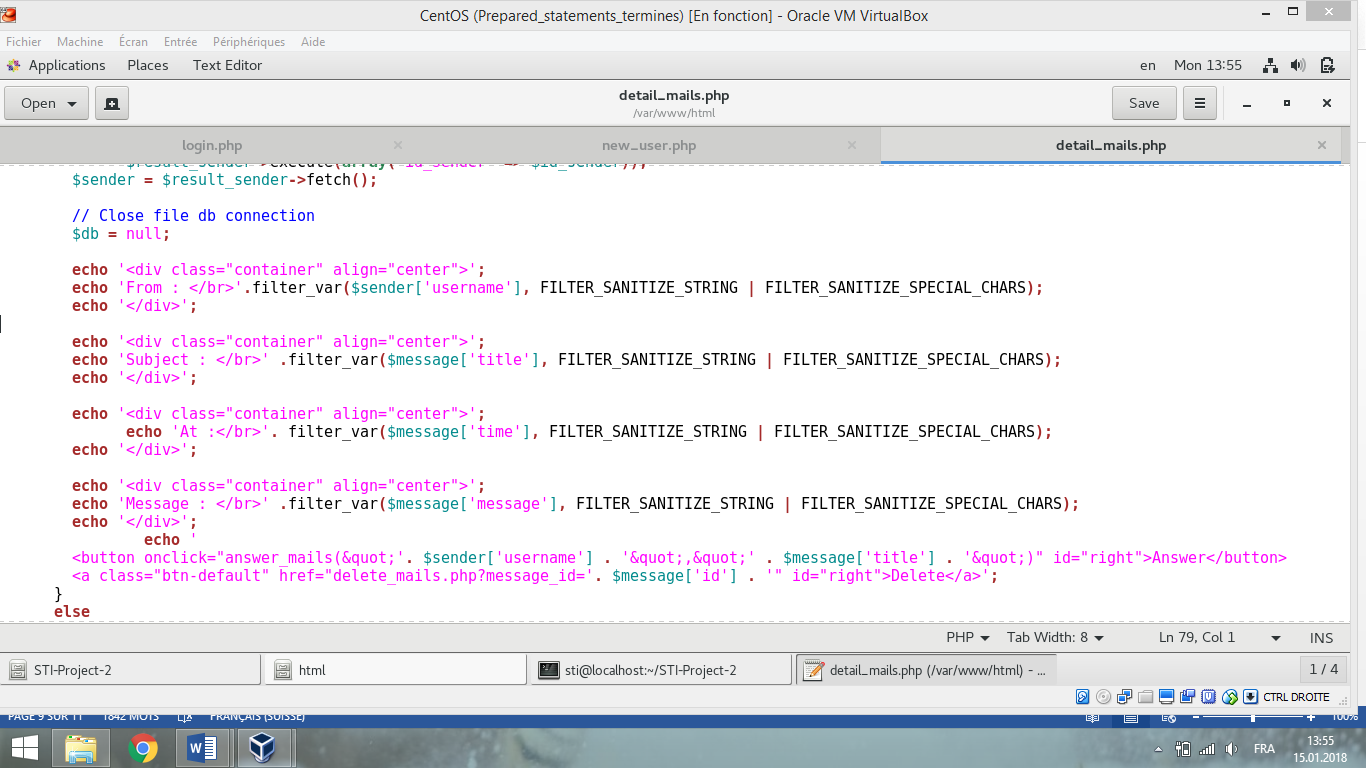
Il faut empêcher la création de comptes avec le même username. Le username doit être unique !



## Contre-mesure 8

Il se pourrait que pour une raison X, un script ait pu être stocké dans notre base de données SQLite. Il faut donc, tout comme pour la contre-mesure 4, utiliser la fonction **filter\_var(maVariable, FILTER\_SANITIZE\_STRING | FILTER\_SANITIZE\_SPECIAL\_CHARS)** pour enlever tous les tags HTML dans une chaîne de caractères.

Pour une meilleure sécurité, il est toujours mieux de filter les scripts dans les deux sens, c’est-à-dire lors de l’écriture dans la base de données et lors de la lecture d’une valeur de la database.



# Conclusion

Avant de commencer la deuxième phase du projet, l’application de messagerie comportait énormément de failles de sécurité. Nous avons tenté d’en sécuriser la plus grande partie mais il en reste néanmoins quelques-unes qui n’ont pas pu être traitées.

Nous considérons que notre application de messagerie est maintenant beaucoup plus sûre et sécurisée qu’auparavant mais elle comporte encore quelques faiblesses que nous n’avons pas réussies à corriger ou que nous n’avons pas eu le temps de traiter.

1. Pas de sel avec le hash : nous avons rencontré des problèmes avec la fonction *hash()* pour ajouter un sel aléatoire lors du hachage du mot de passe. Nous n’avons donc pas réussi à mettre en place un système de sel pour le hachage des mots de passe.
2. Les ID ne sont pas générés aléatoirement : les ID des utilisateurs et des messages sont incrémentés de 1 à chaque nouvel utilisateur ou chaque nouveau message. Pour une meilleure protection, il faudrait pouvoir générer ces ID de manière totalement aléatoire.
3. Pas de protection contre le brute force : nous n’avons pas eu le temps d’implémenter un mécanisme qui désactiverait momentanément le compte d’un utilisateur avec lequel on aurait échoué plusieurs fois consécutives à se loguer. Cela permettrait de se protéger contre les attaques de brute force mais augmenterait en contrepartie le risque de dénis de service sur les comptes des utilisateurs.
4. Pas de protection contre la vulnérabilité CSRF (Cross-Site Request Forgery) : nous n’avons pas traité la vulnérabilité CSRF mais nous nous sommes renseignés sur comment protéger notre application contre cela pour pouvoir le faire dans le futur. Il faudrait générer un nouveau token à chaque requête effectuée dans une session active.
5. Utilisation du protocole HTTPS à la place d’HTTP : il faudrait que nous mettions en place les certificats nécessaires pour que notre application puisse fonctionner avec le protocole HTTPS qui est plus sécurisé qu’HTTP que nous utilisons actuellement.
6. Définir un timeout après un certain temps d’inactivité d’une session sur l’application : quand un utilisateur est inactif un certain temps sur l’application, il serait déconnecté pour éviter que quelqu’un aille sur sa session. Mais en contrepartie, il devrait se reconnecter presque à chaque fois qu’il déciderait d’aller consulter ses mails.