Rapport — Correction des vulnérabilités (TP Chapitre 3)

Auteur : Mohamed Taher Ghazel

Projet : Application web PHP/MySQL

Repo Github : https://github.com/MedTaherGhazel/login.git

1. Objectif

L'objectif de ce travail est d'identifier et de corriger les vulnérabilités livrées intentionnellement dans l'application PHP/MySQL fournie, et d'implémenter les mécanismes vus en cours pour prévenir :

- l'injection SQL (SQLi),
- le Cross-Site Scripting (XSS) — stocké et réfléchi,
- le Cross-Site Request Forgery (CSRF),

ainsi que d'améliorer la gestion des mots de passe et des sessions. Le code produit doit appliquer des bonnes pratiques côté serveur et côté client.

2. Vulnérabilités initialement trouvées

Après examen des fichiers fournis, j'ai relevé les vulnérabilités suivantes :

3.1 Injection SQL

- Fichiers concernés : `add.php` (ancienne version), `show.php`

- Exemple : concaténation directe des entrées utilisateur dans les requêtes SQL :

```php
$sql = "insert into login (username, password) values ('".$username."','".$password."');";
$sql = "select * from login where username ='".$usr."'";
```

3.2 2. **XSS (reflected & stored)**

Fichier concerné : `show.php`

Exemple : affichage sans échappement de `$_POST['usr']` et injection directe dans le HTML :

```php
echo ($_POST['usr']);
echo "<h1><center> ID ... '".$usr."' : ".$row['ID']." </center></h1>";
```

3.3 CSRF

Tous les formulaires (login, ajout d'utilisateur) ne contenaient pas de token CSRF. Un site externe pouvait soumettre un POST malveillant au nom d'un utilisateur authentifié.

3.4 Mots de passe en clair

Les mots de passe étaient insérés tels quels dans la base de données, ce qui est inacceptable.

3.5 Fuite d’information

Affichage direct d’erreurs SQL/DB à l’utilisateur (ex : ` echo "Error: " . $sql . "<br>" . $conn->error;`), exposant la structure interne.

---

3. Correctifs appliqués

Pour chaque vulnérabilité ci-dessus, j’ai appliqué des corrections concrètes. Extraits avant / après :

3.6 Prévention de l’injection SQL

Avant :

```php
$sql = "select * from login where username ='".$usr."'";
$result = mysqli_query($con, $sql);
```

Après (prepared statement) :

```php
$stmt = $con->prepare("SELECT ID, username, password FROM login WHERE username = ?");
$stmt->bind_param('s', $username);
$stmt->execute();
$result = $stmt->get_result();
```

```

Raisons : Les prepared statements séparent le code SQL des données et empêchent l'exécution de payloads d'injection.

## 3.7 Protection contre le XSS

Avant :

```php
echo ($_POST['usr']);
```

Après (échappement) :

```php
function e($s) { return htmlspecialchars($s, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8'); }
echo e($_SESSION['username']);
```

Raisons : `htmlspecialchars()` empêche l'interprétation HTML/JS des chaînes utilisateur, bloquant XSS réfléchi et stocké lors de l'affichage.

3.8 Protection CSRF

Ajout dans les formulaires (exemple) :

```html
<input type="hidden" name="csrf_token" value="<?php echo
htmlspecialchars($_SESSION['csrf_token']); ?>">
```
```

Validation côté serveur :

```php
if (!isset($_POST['csrf_token']) || !hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'])) {
   http_response_code(400);
   die("Invalid CSRF token.");
}
```

Raisons : Un token par session - imprévisible - empêche qu'un site tiers soumette des requêtes authentifiées.

3.9 Hashage des mots de passe

Avant (stockage en clair) :

```php
INSERT INTO login (username, password) VALUES ('user','password');
```

Après (hashage) :

```php
$hash = password_hash($password, PASSWORD_DEFAULT);
$stmt = $con->prepare("INSERT INTO login (username, password) VALUES (?, ?)");
$stmt->bind_param('ss', $username, $hash);
```

```

Et vérification :

```php
if (password_verify($password, $storedHash)) { /* auth OK */ }
```

Raisons : `password_hash` + `password_verify` assurent une résistance aux compromissions de DB.

3.10 Meilleures pratiques générales

- session_regenerate_id(true)** après authentification pour éviter fixation de session.
- set_charset('utf8mb4')** pour la connexion (prévenir problèmes d'encodage et XSS via encodages bizarres).
- Ne pas afficher d'erreurs SQL** côté client ; logguer côté serveur (error_log).
- Validation côté serveur** : longueur, présence des champs.

4. Tests réalisés (procédure et résultats)

Pour vérifier chaque correctif, j'ai exécuté les tests suivants et observé les résultats :

3.11 Test SQLi (login bypass)

- Payload : username = `' OR '1'='1` dans le formulaire de login.
- Avant : la requête concaténée pourrait retourner la première ligne et permettre un contournement.
- Après : la requête préparée traite l'entrée comme donnée : authentification échoue si mot de passe invalide. **Résultat : succès** (injection bloquée).
```

3.12 Test XSS stocké/réfléchi

- Payload : enregistrer username = ` <script>alert('XSS')</script>` via le formulaire d'ajout.
- Avant : affichage exécutait l'alerte lors de la visite de ` show.php ` .
- Après : la sortie est échappée ; la page affiche ` <script>... ` et aucun script ne s'exécute.
- Résultat : succès(XSS bloqué).

3.13 Test CSRF

- Payload : page externe qui fait un POST automatique vers ` add.php ` (sans token).
- Avant : compte créé sans interaction explicite.
- Après : requête rejetée (HTTP 400, message « Invalid CSRF token. »). **Résultat : succès** (CSRF bloqué).

3.14 Test mots de passe

- Ajout d'un utilisateur via le formulaire ; vérification que la DB contient un hash (` $2y$... `). Tentative de connexion avec mot de passe correct réussit (via ` password_verify `).
- Résultat : succès

5. Extraits utiles à insérer dans le code (pour le rapport)

3.15 Exemple d'échappement :

```php
echo htmlspecialchars($value, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');
```

```

### 3.16 Exemple de prepared statement :

```php
$stmt = $con->prepare("INSERT INTO login (username, password) VALUES (?, ?)");
$stmt->bind_param('ss', $username, $hash);
```

3.17 Exemple de génération/validation CSRF :

```php
// génération
$_SESSION['csrf_token'] = bin2hex(random_bytes(32));

// validation
hash_equals($_SESSION['csrf_token'], $_POST['csrf_token']);
```

6. Recommandations finales

- Toujours utiliser des prepared statements pour toute requête contenant des données utilisateurs.
- Utiliser `password_hash()` et `password_verify()` pour la gestion des mots de passe.
- Échapper toute sortie vers le HTML avec `htmlspecialchars()` (ou utiliser un moteur de templates qui échappe automatiquement).
- Protéger tous les formulaires state-changing par des tokens CSRF.
- Limiter les informations d'erreur envoyées aux clients ; conserver les logs côté serveur.
- Activer HTTPS, définir des politiques de cookies (`HttpOnly`, `Secure`, `SameSite`) et appliquer des règles de verrouillage de compte après plusieurs tentatives de connexion échouées.
```

- Scanner régulièrement l’application avec des outils automatisés (ex : ZAP, sqlmap en environnement de test).