

Créer un système d'inscription & connexion en PHP / MySQL

Version NON SÉCURISÉE

- Cette partie est purement pédagogique et montre concrètement pourquoi la version non sécurisée est dangereuse.
 - Ce qui nous conduit ensuite à construire la version sécurisée comme "solution logique et inévitable".
-

Exploitation des failles dans la version NON sécurisée

Avant d'améliorer la sécurité, voici ce que *n'importe qui* pourrait faire sur un système construit avec cette version.

Cela montre clairement pourquoi on doit passer à la version sécurisée.

1. Injection SQL — Connexion sans connaître le mot de passe

La requête non sécurisée :

```
SELECT * FROM register WHERE mail='$mail' AND mdp='$mdp'
```

Exploit : Se connecter sans connaître le mot de passe

Dans le champ **email** :

```
' OR 1=1 #
```

Mot de passe : ne pas saisir de mot de passe.

Pourquoi ça marche ?

La requête devient :

```
SELECT * FROM register  
WHERE mail='' OR '1'='1' # ' AND mdp=''
```

'1'='1' est toujours vrai --> donc l'utilisateur est considéré comme connecté.

Le # commente la suite, rendant mdp inutile.

N'importe quel compte peut être ouvert.

2. Injection SQL — Prendre le compte Admin

Cet exploit **ne fonctionne que si un compte "admin" existe** et que la clause `mail='admin'` correspond à celui-ci.

L'attaquant est connecté comme admin.

Dans le champ **email** :

```
admin' #
```

Dans le champ **mot de passe** : peu importe.

Cela transforme la requête en :

```
SELECT * FROM register
WHERE mail='admin'
```

Le `#` désactive tout ce qui suit.

- Le mot de passe est totalement ignoré.
- L'attaquant devient **admin**.

3. Injection SQL — Voir toutes les données de la table

« L'injection UNION SELECT ne fonctionne que si le nombre de colonnes et leurs types correspondent à la requête d'origine. »

Dans un champ (email par exemple) :

```
' UNION SELECT 1, 'HACK', 'HACK', 'HACK', 'HACK' #
```

Cette injection :

- fusionne votre requête avec une autre
- affiche les données d'une autre requête
- peut révéler les colonnes, la structure, etc.

Très dangereux sur les pages qui affichent des résultats.

4. Injection SQL — Créer un compte sans passer par le formulaire (page A_login.php)

Dans le champ **email** :

```
Nom1'; INSERT INTO register(nom, prenom, mail, mdp) VALUES ('Hacker','Pirate','pirate@carnus.fr','1234');
```

La requête exécutée devient :

```
SELECT * FROM register WHERE mail='Nom1';  
INSERT INTO register(nom, prenom, mail, mdp)  
VALUES ('Hacker','Pirate','pirate@carnus.fr','1234');
```

L'attaquant **crée directement un compte** dans la base.

Si le serveur autorise plusieurs requêtes (multi_query), c'est dramatique.

Par défaut, `mysqli_query()` bloque les requêtes multiples. Cette faille est donc spécifique à `mysqli_multi_query()` — l'emploi de cette fonction est particulièrement dangereux ici.

5. Injection SQL — Supprimer toute la table

(Exemple volontairement extrême)

Champ email :

```
test'; DROP TABLE register; --
```

La requête devient :

```
SELECT * FROM register WHERE mail='test';  
DROP TABLE register; -- ' AND mdp='xyz'
```

- La table entière est détruite.
- Perte totale des données.

Heureusement, `multi_query()` est souvent désactivé. Mais certains hébergements peu sécurisés l'autorisent.

- `--` (ou `#`) commence un commentaire en SQL et ignore tout ce qui suit.
- En mode multi-query, `DROP TABLE` serait exécuté avant que PHP ne s'en rende compte.

Même si cette attaque paraît extrême, elle montre le pouvoir destructeur d'une injection SQL exécutée sans restriction.

6. Mot de passe en clair — Vol immédiat

Dans la version non sécurisée, les mots de passe sont enregistrés **en clair**.

Conséquence :

- Un simple accès à phpMyAdmin suffit à les lire
- Une fuite de la base = accès immédiat à tous les comptes
- Beaucoup d'utilisateurs réutilisent leurs mots de passe ailleurs

La compromission est totale.

7. Vérification mail + mdp --> faille logique

Dans le code non sécurisé :

```
SELECT * FROM register WHERE mail='$mail' AND mdp='$mdp'
```

- Le compte est considéré comme "existant" **si mail + mdp existent**
- Si l'utilisateur change de mot de passe --> le test ne trouve plus le compte
- L'attaquant pourrait contourner la détection du compte existant
- Cela génère des incohérences et peut être exploité

Rappel technique :

- ' interrompt une chaîne SQL.
- # ou -- transforme le reste de la ligne en commentaire.
- UNION, INSERT et DROP permettent d'exécuter d'autres requêtes.
- Avec multi_query(), tout cela peut s'enchaîner dans une seule commande.

Conclusion : Comment corriger ces failles ?

Ces exemples montrent **pourquoi** on passe ensuite à la version sécurisée :

- Mot de passe hashé → password_hash()
- password_verify()
- Protection SQL → Requetes préparées
- Vérification email-only
- Messages d'erreur neutres
- Sécurisation des sessions regeneration ID + contrôle utilisateur

Ce n'est pas du luxe, c'est indispensable.

Maintenant que nous avons identifié les failles de la version simple, nous allons construire étape par étape la **version sécurisée**, en corrigeant chaque problème un par un.