Injection SQL

Une injection SQL est une attaque consistant à insérer ou manipuler des requêtes SQL mallveillantes dans une application afin d'accéder à des BDD pour modifier ou supprimer des données de manière non autorisée. Elle tire parti de failles dans les applications web qui ne valident pas correctement les entrées utilisateurs.

Il existe plusieurs types d'injections SQL :

1. **Injection SQL classique** : C'est l'attaque la plus courante, consistant à injecter du code malveillant dans une requête SQL, généralement via un chemin de saisie utilisateur.

2. **Injection SQL aveugle (Blind SQL Injection)** : Lorsqu'il est impossible de voir directement les résultats de l'attaque (pas de messages d'erreur), l'attaquant peut déduire des informations sur la BDD en analysant la réponse du serveur.

   Injection SQL aveugle conditionnelle : L'attaquant soumet des requêtes SQL qui provoquent une réponse différente selon qu'une condition soit vrai ou fausse.

   Injection SQL aveugle par estimation : L'attaquant pose des questions pour obtenir des informations sur la structure de la BDD auxquelles il peut répondre oui ou non.

3. **Injection SQL par UNION** : Utilise la clause UNION pour combiner les résultats de la requêtes légitime et ceux de la requête malveillante. Cela permet à l'attaquant d'extraire des données de tables non accessibles via la requête initiale.

4. **Injection SQL basé sur l'URL** : L'attaque exploite des paramètres passés dans l'URL d'une application web.

Une injection SQL fonctionne de manière simple, lorsqu'un utilisateur soumet des données par exemple dans un formulaire, l'application utilise ces données dans une requête SQL, cela permet à un attaquant de modifier la requête.

Voici un exemple classique d'une connexion à un compte :

SELECT * FROM users WHERE name_user = 'admin' AND password = 'admin1234' ;

Voici comment l'attaquant peut modifier cette requête :

SELECT * FROM users WHERE name_user = ' ' OR 1=1 ;

La condition 1=1 est toujours vrai donc ici l'attaquant peut avoir accès à un compte de manière non-autorisé.

Exemples d'injections SQL :

' UNION SELECT table_name, null FROM information_schema.tables –

Cela permet de récupérer les noms de toutes les tables de la BDD.

' UNION SELECT column_name, null FROM information_schema.columns WHERE table_name = 'utilisateurs' –

Ici on récupère les colonnes de la table utilisateurs.

Pour se protéger au mieux des injections SQL voici quelques bonnes pratiques :

1. **Utiliser des requêtes paramétrées :** Il s'agit d'utiliser des bibliothèques qui permettent de séparer le code SQL des données envoyées par l'utilisateur. Cela empêche l'injection SQL**.**
   Exemple avec PHP PDO (PHP Data Object) :
   $stmt = $pdo->prepare('SELECT * FROM utilisateurs WHERE nom_utilisateur = :nom AND mot_de_passe = :mdp') ;
   $stmt ->execute(['nom' => $nom_utilisateur, 'mdp' => $mot_de_passe]) ;

$stmt = $pdo->prepare(...)

• **$pdo** : C'est l'objet de connexion à la base de données, créé auparavant avec quelque chose comme : $pdo = new PDO('mysql:host=localhost;dbname=my_base', 'utilisateur', 'motdepasse');

• **prepare(...)** : Cette méthode prépare une requête SQL à exécuter ultérieurement. Elle ne l'exécute **pas immédiatement**, ce qui permet

• de sécuriser la requête (prévention des injections SQL),

• de l'optimiser si elle est réutilisée plusieurs fois,

• de remplacer des **paramètres nommés,** ici (:nom, :mdp) par des **valeurs sécurisées** au moment de l'exécution.

• **Requête SQL** :

SELECT * FROM utilisateurs WHERE nom_utilisateur = :nom AND mot_de_passe = :mdp

Elle récupère toutes les colonnes de la table utilisateurs pour un utilisateur donné, si le nom et le mot de passe correspondent.

$stmt->execute([...])

• Cette ligne **exécute** la requête préparée en **remplaçant les paramètres** :nom et :mdp par les valeurs données dans le tableau associatif.

• PDO s'occupe de **protéger automatiquement ces valeurs** :

• Il **échappe les caractères spéciaux**,

• Il **les convertit au bon type SQL** (entier, chaîne, etc.),

• Il **empêche toute tentative d'injection SQL**, même si $nom_utilisateur contient des caractères malveillants comme ' OR 1=1 --.

Avantages de cette méthode :

• **Sécurité** : Les valeurs sont insérées **sans concaténation** dans la requête SQL.

• **Lisibilité** : Le code reste propre et facile à lire.

• **Réutilisabilité** : Tu peux réutiliser $stmt pour d'autres valeurs (ex. dans une boucle).

• **Indépendance du SGBD** : PDO fonctionne avec MySQL, PostgreSQL, SQLite, etc.

Exemple concret :

```
$nom_utilisateur = $_POST['username'];

$mot_de_passe = $_POST['password'];

$stmt = $pdo->prepare('SELECT * FROM utilisateurs WHERE nom_utilisateur = :nom AND
mot_de_passe = :mdp');

$stmt->execute(['nom' => $nom_utilisateur, 'mdp' => $mot_de_passe]);

$user = $stmt->fetch(); // Récupère le résultat sous forme de tableau associatif

if ($user) {

   echo "Connexion réussie !";

} else {

   echo "Identifiants incorrects.";

}
```

2. **Utiliser des ORM (Object-Relational Mapping) :** Beaucoup d'ORMs modernes évitent directement l'injection SQL en automatisant l'utilisation de requêtes paramétré.
3. **Activer des messages d'erreurs généraux :** Les messages d'erreurs détaillés peuvent donner des informations sensibles sur la base de données.
4. **Mise à jour régulière des systèmes et logiciels :** Maintenir à jour les systèmes frameworks, et bibliothèques pour bénéficier des dernières corrections de sécurité.
5. **Limiter les permissions des comptes de base de données :** Ne jamais donner trop de privilèges aux comptes utilisés par l'application. Il faut limiter au stricte nécéssaire.

Conclusion : L'injection SQL est une vulnérabilité sérieuse qui peut compromettre la sécurité d'une application web. Il est essentiel d'adopter les bonnes pratiques comme les requêtes préparées, la validation des entrées et la gestion des permissions pour se protéger efficacement.