Compte-rendu — Présentation du

SQL Injection (SQLi)

Rédigé par : Kitoko David

16 octobre 2025

Résumé

Ce document présente de manière synthétique : définition, impacts, détection, exemples d'exploitation, variantes (blind, second-order), méthodes d'examen de la base, et recommandations pour prévenir les vulnérabilités SQL injection.

Table des matières

1 Définition

Le *SQL injection* (abrégé SQLi) est une vulnérabilité de sécurité web permettant à un attaquant d'altérer les requêtes SQL qu'une application envoie à sa base de données. En injectant du code SQL via des entrées contrôlées (URL, formulaires, JSON, cookies...), l'attaquant peut accéder, modifier ou supprimer des données auxquelles il ne devrait pas avoir accès, et dans certains cas compromettre le serveur ou l'infrastructure en aval.

2 Impact d'une attaque réussie

Les conséquences possibles incluent :

— Exfiltration de données sensibles (mots de passe, numéros de carte, informations personnelles).

— Contournement d'authentification (connexion en tant qu'un autre utilisateur).
— Modification ou suppression persistante de données.
— Implantation de portes dérobées ou compromission prolongée.
— Déni de service via requêtes lourdes.

Ces incidents peuvent mener à des pertes financières, un préjudice réputationnel et des sanctions réglementaires.

3 Détection des vulnérabilités SQLi

Méthodes manuelles courantes :

— Injecter une apostrophe simple `'` et observer les erreurs.
— Tester des conditions booléennes (`OR 1=1`, `OR 1=2`) et comparer les réponses.
— Utiliser des payloads *time-based* pour mesurer les délais de réponse.
— Employer des payloads *OAST* (out-of-band) pour provoquer une interaction réseau observable.

Outils : scanners (par exemple Burp Scanner) permettent d'automatiser la découverte de nombreux cas.

4 Emplacements fréquents d'injection

Une injection peut se produire dans différentes parties d'une requête SQL :

— Clause `WHERE` d'une `SELECT` (cas le plus courant).
— Valeurs d'un `INSERT` ou d'un `UPDATE`.
— Noms de table ou de colonne (ex. dans un `SELECT` construit dynamiquement).
— Clause `ORDER BY`.
— Toute entrée utilisateur passant dans une requête SQL.

5 Exemples et techniques d'exploitation

5.1 Récupérer des données cachées

URL vulnérable :

```
https://insecure-website.com/products?category=Gifts
```

Requête SQL attendue :

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

Injection illustrative :

```
https://insecure-website.com/products?category=Gifts'--
```

La requête devient :

```
SELECT * FROM products WHERE category = 'Gifts'--' AND released = 1
```

Le reste est commenté ; la contrainte `released = 1` est ignorée.

5.2 Contournement d'authentification (subverting logic)

Si l'application exécute :

```
SELECT * FROM users WHERE username = 'wiener' AND password = 'bluecheese
    '
```

En soumettant le nom d'utilisateur `administrator'--` et un mot de passe vide :

```
SELECT * FROM users WHERE username = 'administrator'--' AND password =
    ''
```

La vérification du mot de passe est commentée et l'attaquant peut se connecter.

5.3 Attaque UNION (exfiltration depuis d'autres tables)

Si la requête retourne `name, description`, on peut ajouter :

```
' UNION SELECT username, password FROM users--
```

Cela concatène les résultats d'une autre table dans la réponse.

5.4 Blind SQLi

Lorsque l'application ne renvoie pas les résultats ni d'erreur :

- **Boolean blind** : injecter une condition vraie/fausse et observer un changement de comportement.
- **Time-based** : injecter `IF(condition, SLEEP(5), 0)` et mesurer le délai pour inférer la vérité de la condition.
- **OAST** : provoquer une requête DNS/HTTP vers un domaine contrôlé pour exfiltrer des données hors bande.

5.5 Second-order (stocké)

Les données injectées sont d'abord stockées (apparemment sans danger), puis réutilisées plus tard dans une requête non sécurisée, provoquant l'injection à un autre moment (attaque dite *second-order*).

6 Examiner la base et techniques avancées

Après identification d'une injection, il est courant d'essayer d'obtenir :

- La version du SGBD (par ex. `SELECT * FROM v$version` sur Oracle).
- La liste des tables et colonnes (`information_schema.tables`, etc.).
- Des particularités du SGBD (syntaxe, commentaires, requêtes empilées).

7 Mesures de prévention (bonnes pratiques)

1. **Requêtes paramétrées (prepared statements)** : ne jamais concaténer directement des entrées utilisateur dans une requête.

```
// Mauvais (vuln rable)
String query = "SELECT * FROM products WHERE category = '" + input
    + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

```
// Bon (param tr )
PreparedStatement ps = conn.prepareStatement(
  "SELECT * FROM products WHERE category = ?");
ps.setString(1, input);
ResultSet rs2 = ps.executeQuery();
```

Listing 1 – Exemple Java vulnérable et corrigé

2. **Whitelist** : pour les cas où des valeurs (ex. colonnes, ORDER BY) doivent être dynamiques, n'accepter que des valeurs prédéfinies.
3. **Échapper correctement** les caractères (solution de dernier recours et dépendante du SGBD).
4. **Principe du moindre privilège** : le compte DB de l'application doit avoir seulement les droits nécessaires.
5. **Gestion stricte des erreurs** : ne pas exposer les messages d'erreur SQL aux utilisateurs finaux.
6. **WAF / filtrage** : pare-feu applicatif pour réduire les attaques automatisées (ne remplace pas des correctifs dans le code).
7. **Revue de code et tests** : analyses statiques/dynamiques et pentests réguliers.
8. **Ne pas faire confiance aux données stockées** : ne pas considérer comme sûres les données venant de la base si elles proviennent d'utilisateurs.