THEME : INJECTION SQL (BLIND)

Réalisé par :

Fady Axel BAMBA
Maixente Yakeliomi KOHIO
Pierre Marie KONATE
Cheick Abdel Hadime Hakim SAWADOGO
Mouhamed SALANE

Introduction

Définition de l'injection SQL

L'injection SQL (SQL Injection) est une vulnérabilité de sécurité web qui permet à un attaquant d'insérer des requêtes SQL malveillantes dans les champs d'entrée d'une application web. Cette attaque vise à manipuler la base de données sous-jacente pour accéder, modifier ou supprimer des informations sensibles.

Présentation de l'injection SQL aveugle (Blind SQL Injection)

L'injection SQL aveugle est une variante de l'injection SQL classique où l'attaquant ne reçoit pas directement de retour d'erreur ou de réponse visible de la base de données. Au lieu d'afficher les résultats des requêtes, l'application répond de manière binaire (vrai ou faux) ou avec un délai dans l'exécution de la requête. Cette technique est souvent utilisée lorsque les messages d'erreur SQL sont désactivés ou filtrés.

Différences entre injection SQL classique et injection SQL aveugle

Critère	Injection SQL Classique	Injection SQL Aveugle
Réponse de la base de données	L'attaquant reçoit des messages d'erreurs et des résultats SQL exploitables	Aucune réponse directe de la base de données
Facilite d'exploitation	Facile à détecter et exploiter2	Plus difficile à exploiter, nécessite des techniques spécifiques
Méthodes utilisées	Erreurs SQL, affichage des données	Réponse booléennes, délais d'exécution

I) Fonctionnement de l'Injection SQL Aveugle

Principe général de l'attaque

L'injection SQL aveugle repose sur l'exploitation de requêtes SQL mal sécurisées dans une application web. Un attaquant envoie des entrées malveillantes pour vérifier indirectement si l'application est vulnérable, en observant des changements dans le comportement de l'application.

Raisons pour lesquelles une application est vulnérable

• Mauvaise gestion des entrées utilisateur (absence de filtrage et de validation)

• Utilisation de requêtes SQL dynamiques concaténées avec des entrées utilisateur

• Absence de requêtes préparées (Prepared Statements)

• Affichage générique des erreurs (aucun retour de message SQL explicite)

Types d'injections SQL aveugles

1. **Boolean-Based Blind SQL Injection**

L'attaquant injecte une requête SQL qui renvoie une réponse conditionnelle (vrai ou faux). En observant la réponse de l'application (changement d'affichage, redirection, etc.), il peut en déduire des informations sur la structure de la base de données.

Exemple : Si une application utilise la requête suivante :

```
SELECT * FROM utilisateurs WHERE id = '$id';
```

L'attaquant peut tester :

```
id=1' AND 1=1 --
```

Si la page se charge normalement, cela signifie que la requête est valide. Ensuite, il peut tester :

```
id=1' AND 1=2 --
```

Si la page affiche un comportement différent (ex. : message d'erreur, absence de réponse), alors l'application est vulnérable.

2. **Time-Based Blind SQL Injection**

Dans cette variante, l'attaquant injecte une commande SQL qui force la base de données à attendre un certain temps avant de répondre. S'il observe un retard dans la réponse, il peut en déduire que l'injection fonctionne.

Exemple :

id=1' AND SLEEP(5) --

Si la page met 5 secondes à répondre, cela confirme la vulnérabilité.

II)Exploitation de la Vulnérabilité

Comment un attaquant identifie une vulnérabilité

L'attaquant commence par tester différentes entrées pour observer le comportement de l'application :

• Envoi de caractères spéciaux (', ", --, #) pour repérer des erreurs SQL masquées

• Injection de requêtes conditionnelles (AND 1=1, AND 1=2) pour observer des variations de réponse

• Utilisation de fonctions SQL comme SLEEP(), IF(), LENGTH(), etc., pour extraire des informations

Exemples de requêtes SQL malveillantes

Détection d'une table existante :

id=1' AND (SELECT COUNT(*) FROM information_schema.tables WHERE table_name='utilisateurs') > 0 --

Si la table utilisateurs existe, la page répond différemment.

Récupération du nom d'une colonne :

id=1' AND (SELECT LENGTH(column_name) FROM information_schema.columns WHERE table_name='utilisateurs' LIMIT 1) > 5 --

Cela permet de deviner la longueur des noms de colonnes.

Techniques d'exfiltration de données

Les attaquants peuvent extraire des informations en injectant des requêtes SQL répétées avec des tests booléens ou des délais.

Exemple : Récupération d'un caractère du nom d'utilisateur

id=1' AND ASCII(SUBSTRING((SELECT username FROM utilisateurs LIMIT 1),1,1))=97 --

Si la page répond normalement, cela signifie que le premier caractère du username est a (ASCII 97).

III) Conséquences et Risques

Accès non autorisé aux bases de données

Un attaquant peut obtenir un accès illimité aux données stockées, même sans authentification.

Vol d'informations sensibles

• Accès aux identifiants et mots de passe (ex. : hachés mais parfois mal protégés)

• Lecture de données personnelles (emails, adresses, numéros de téléphone)

Possibilité de prise de contrôle totale d'un système

Dans certains cas, un attaquant peut exécuter des commandes système via SQL, permettant :

• L'exfiltration massive de données

• L'exécution de code malveillant sur le serveur

• L'altération ou la suppression de données

IV) Etude des différents niveaux de sécurité DVWA

NIVEAU LOW

- **Vulnérabilités** :
    - L'entrée id est récupérée via $_GET et injectée directement dans la requête SQL sans validation.
    - Utilisation de la concaténation de chaînes pour construire la requête (WHERE user_id = '$id'), permettant une injection SQL simple.

- **Risque** : Toutes les injections peuvent être performées sans problème

Low SQL Injection (Blind) Source

```php
<?php

if( isset( $_GET[ 'Submit' ] ) ) {
    // Get input
    $id = $_GET[ 'id' ];
    $exists = false;

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            // Check database
            $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
            try {
                $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ); // Removed 'or die' to suppress mysql errors
            } catch (Exception $e) {
                print "There was an error.";
                exit;
            }
```

NIVEAU MEDIUM

Medium SQL Injection (Blind) Source

```php
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $id = $_POST[ 'id' ];
    $exists = false;

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            $id = ((isset($GLOBALS["___mysqli_ston"]) && is_object($GLOBALS["___mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["___mysqli_ston"],  $id ) : ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string

            // Check database
            $query  = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
            try {
                $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ); // Removed 'or die' to suppress mysql errors
            } catch (Exception $e) {
                print "There was an error.";
                exit;
            }
```

- **Améliorations/Restrictions** :
    - L'entrée id est récupérée via $_POST, ce qui complique légèrement l'accès direct.

- `mysql_real_escape_string()` : Fonction obsolète pour échapper les caractères spéciaux, inefficace si les guillemets manquent.

- **Vulnérabilités** :

- **Absence de guillemets** : Permet à un attaquant de contourner l'échappement en injectant directement des opérateurs SQL (ex: `OR 1=1`).
- **Liste déroulante + POST** : Rend l'attaque moins évidente, mais ne résout pas la vulnérabilité sous-jacente.

NIVEAU HIGH

High SQL Injection (Blind) Source

```php
<?php

if( isset( $_COOKIE[ 'id' ] ) ) {
    // Get input
    $id = $_COOKIE[ 'id' ];
    $exists = false;

    switch ($_DVWA['SQLI_DB']) {
        case MYSQL:
            // Check database
            $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1;";
            try {
                $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query ); // Removed 'or die' to suppress mysql errors
            } catch (Exception $e) {
                $result = false;
            }

            $exists = false;
            if ($result !== false) {
```

- **Améliorations/Restrictions :**
    - **L'id est récupéré depuis un cookie ($_COOKIE), ce qui oblige l'attaquant à manipuler les cookies.**
    - **Utilisation de mysqli_query, mais sans requêtes préparées.**
- **Vulnérabilités :**
    - **Les cookies peuvent être modifiés côté client (via des outils comme Burp Suite).**

NIVEAU IMPOSSIBLE

Impossible SQL Injection (Blind) Source

```php
<?php

if( isset( $_GET[ 'Submit' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );
    $exists = false;

    // Get input
    $id = $_GET[ 'id' ];

    // Was a number entered?
    if(is_numeric( $id )) {
        $id = intval ($id);
        switch ($_DVWA['SQLI_DB']) {
            case MYSQL:
                // Check the database
                $data = $db->prepare( 'SELECT first_name, last_name FROM users WHERE user_id = (:id) LIMIT 1;' );
                $data->bindParam( ':id', $id, PDO::PARAM_INT );
                $data->execute();

                $exists = $data->rowCount();
                break;
            case SQLITE:
                global $sqlite_db_connection;

                $stmt = $sqlite_db_connection->prepare('SELECT COUNT(first_name) AS numrows FROM users WHERE user_id = :id LIMIT 1;' );
                $stmt->bindValue(':id',$id,SQLITE3_INTEGER);
                $result = $stmt->execute();
                $result->finalize();
                if ($result !== false) {
                    // There is no way to get the number of rows returned
                    // This checks the number of columns (not rows) just
                    // as a precaution, but it won't stop someone dumping
                    // multiple rows and viewing them one at a time.
```

- **Mesures de sécurité :**

- **Validation stricte** : Conversion de l'id en entier avec intval(), rejetant toute entrée non numérique.
- **Requêtes préparées** : Utilisation de PDO avec bindParam pour séparer les données de la structure SQL.
- **Protection Anti-CSRF** : Vérification d'un token pour bloquer les requêtes frauduleuses.
- **Gestion des erreurs** : Aucun message d'erreur exposé, limitant les fuites d'informations.

- **Résultat** : Rend l'injection SQL théoriquement impossible grâce au typage strict et aux requêtes paramétrées.

V) Méthodes de Protection et de Prévention

1.Utilisation des requêtes préparées (Prepared Statements)

Les requêtes préparées empêchent l'injection SQL en séparant les données des commandes SQL.

Exemple en PHP avec PDO :

```
$stmt = $pdo->prepare("SELECT * FROM utilisateurs WHERE id = :id"); $stmt->execute(['id' => $id]);
```

Ici, la valeur de $id est traitée comme une donnée et non comme une commande SQL.

2. **Validation et filtrage des entrées utilisateurs**

• Vérifier que les entrées utilisateur correspondent aux formats attendus (ex. : uniquement des nombres pour un id)
• Supprimer ou échapper les caractères spéciaux dangereux (', ", --, etc.)

3. **Mise en place d'un Web Application Firewall (WAF)**

Un WAF peut détecter et bloquer des requêtes malveillantes en analysant les patterns d'attaque.

4. **Restriction des permissions sur les bases de données**

- Attribuer le **moindre privilège nécessaire** aux comptes utilisateurs SQL

- Désactiver les fonctionnalités dangereuses si elles ne sont pas nécessaires (ex. : LOAD_FILE(), xp_cmdshell)

Conclusion

L'injection SQL aveugle est une attaque redoutable permettant à un attaquant d'exploiter des failles même en l'absence de messages d'erreur visibles. Une bonne protection repose sur l'utilisation de requêtes préparées, un filtrage rigoureux des entrées et des contrôles de sécurité comme les WAF et la restriction des permissions en base de données.