Injections SQL

Définition

L'injection SQL est une **attaque qui vise à exploiter une vulnérabilité dans une application web** pour manipuler directement les requêtes SQL envoyés à une Base de Données. L'attaquant peut injecter du code SQL malveillant dans des champs de saisie ou des URL, ce qui peut entraîner des conséquences graves pour la sécurité de la base de données..

Exemples concrets

Ces exemples ne montrent pas d'injection

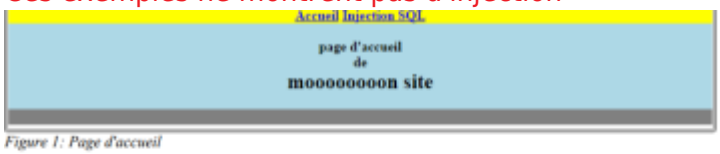

Figure 1: Page d'accueil

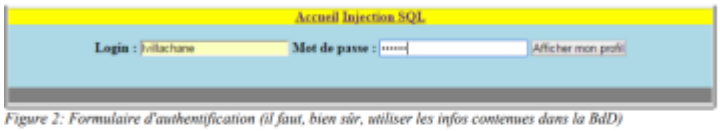

Figure 2: Formulaire d'authentification (il faut, bien sûr, utiliser les infos contenues dans la BdD)

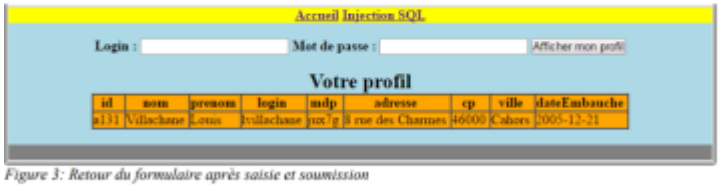

Figure 3: Retour du formulaire après saisie et soumission

Figure 4: Erreur sur page inexistante (URL modifiée à la main)

Se protéger : validation des données entrantes

Voir Validation des données entrantes

Se protéger : principes des requêtes préparées

Généralités

La plupart des SGBD modernes utilisent un mécanisme dit “Requêtes sécurisées” pour sécuriser les injections SQL. Ce mécanisme a pour but d'améliorer la sécurité et l'efficacité des requêtes SQL.

* La requête est analysée puis optimisée avant d'être exécutée, sans inclure les données réelles (les paramètres) au moment de sa préparation.

* Une requête préparée peut être réutilisable plusieurs fois sans être recompilée à chaque fois.

* Cela garantit que le comportement de la requête reste constant.

* Les requêtes préparées s'exécutent plus rapidement lorsqu'elles sont utilisées plusieurs fois.

Les requêtes préparées offrent plusieurs avantages par rapport aux requêtes classiques. Elles permettent de Sécuriser les requêtes en évitant l'insertion de code malveillant et en améliorant les performances des requêtes répétées, car la requête n'a besoin d'être analysée qu'une seule fois, même si elle est exécutée plusieurs fois avec des données différentes. Cela renforce la sécurité en empêchant l'exécution de code SQL injecté et améliore les performances des requêtes répétées. De plus, elles “figent” la structure de la requête, ne laissant aucune place à la modification malveillante de cette dernière, réduisant considérablement la surface d'attaque pour les injections SQL.

Paramètres anonymes

Définition:: Ce sont des marqueurs utilisés dans les requêtes préparées pour définir à l'avance les valeurs saisies pour éviter toute tentative d'injection SQL. Ils ont représentés par des points d'interrogation **(?)** dans la requête SQL.

Lorsque la requête est exécutée, un tableau contenant les valeurs des paramètres est fourni, et ces valeurs sont liées aux marqueurs anonymes dans l'ordre de leur apparition dans la requête. Cela permet de sécuriser les requêtes sans risquer d'injecter du code malveillant dans la Base de Données.

Paramètres nommés

Définition: Une alternative aux paramètres anonymes dans les requêtes préparées. Au lieu d'utiliser des points d'interrogation pour marquer l'emplacement des valeurs, chaque paramètre est identifié par un nom précédé d'un deux points (:). Cela permet de spécifier les paramètres indépendamment de leur position dans la requête et d'améliorer la lisibilité, surtout pour les requêtes complexes. Un tableau associatif est ensuite utilisé pour lier chaque paramètre nommé à une valeur.

Mise en œuvre PHP

Paramètres anonyme

```php
$sql = "INSERT INTO uneTable (uneColonne, uneAutreColonne) VALUES (?, ?)";
$pdo_stmt->execute([150, 'rouge']);
```

Description: Les valeurs des paramètres sont représentées par des points d'interrogation ***?*** dans la requête.

Exécution: Lors de l'exécution, tableau contenant les valeurs est fourni à la méthode execute(). L'ordre des valeurs dans ce tableau doit strictement correspondre à l'ordre des marqueurs ? dans la requête. Cette méthode est simple, mais elle exige de respecter scrupuleusement l'ordre des paramètres.

Paramètres Nommés

```
$sql = "INSERT INTO uneTable (uneColonne, uneAutreColonne)
        VALUES (:uneValeur, :uneAutreValeur)";
$pdo_stmt->execute(array(':uneValeur' => 150, ':uneAutreValeur' =>
'rouge'));
```

Description: Dans cette approche, chaque paramètre est identifié par un nom explicite précédé de : (exemple : :uneValeur). Cela permet de rendre la requête plus compréhensible et plus flexible.

Exécution: Lors de l'exécution, vous passez un tableau associatif, où chaque clé est le nom du paramètre et chaque valeur est celle à utiliser. Nous n'avons pas besoin de suivre l'ordre des paramètres dans la requête. C'est plus lisible et flexible, surtout quand la requête est complexe.

Pourquoi utiliser ces méthodes ?

1. Sécurité : Empêche les injections SQL en séparant le code SQL des données utilisateur.
2. Performance : Une requête préparée est analysée une seule fois, même si elle est exécutée plusieurs fois avec des valeurs différentes.
3. Lisibilité : Les paramètres nommés rendent le code plus clair et plus facile à comprendre.

Mise en œuvre Java

Paramètres anonyme

```
String sql = "INSERT INTO uneTable (uneColonne, uneAutreColonne) VALUES (?,
?)";
PreparedStatement preparedStatement = connexion.prepareStatement(sql);
preparedStatement.setInt(1, 150);
preparedStatement.setString(2, "rouge");
preparedStatement.executeUpdate();
```

Explication :

Les ? représentent des paramètres anonymes, qui seront remplacés par des valeurs réelles avant l'exécution. On utilise setInt() et setString() pour associer les valeurs aux marqueurs, en respectant leur position. Cette méthode empêche les injections SQL car elle empêche l'exécution de code malveillant inséré par un utilisateur.

Paramètres Nommés

```
String sql = "INSERT INTO uneTable (uneColonne, uneAutreColonne) VALUES
(:valeur1, :valeur2)";
MapSqlParameterSource params = new MapSqlParameterSource();
params.addValue("valeur1", 150);
params.addValue("valeur2", "rouge");
```

```
namedParameterJdbcTemplate.update(sql, params);
```

Explication :

Les paramètres sont définis par des noms explicites (:valeur1, :valeur2), ce qui rend le code plus lisible et compréhensible. On utilise une Map (MapSqlParameterSource) pour associer chaque paramètre à sa valeur, ce qui évite l'injection SQL et rend le code plus robuste. Avec cette approche, l'ordre des paramètres n'a pas d'importance, ce qui facilite la maintenance.

Binding

```
String sql = "SELECT * FROM utilisateurs WHERE email = ?";
PreparedStatement preparedStatement = connexion.prepareStatement(sql);
preparedStatement.setString(1, emailSaisiParUtilisateur);
ResultSet resultSet = preparedStatement.executeQuery();
```

Pourquoi utiliser le binding ?

Empêche les attaques SQL en traitant les données utilisateur comme des valeurs et non comme du code SQL. Renforce la sécurité en garantissant que même si un utilisateur saisit " ' OR 1=1 – ", la requête ne sera pas modifiée. Améliore la performance car la requête préparée est précompilée et optimisée par le SGBD.

Pourquoi ces techniques sont essentielles ? Sans requêtes préparées, une application est vulnérable aux injections SQL, ce qui peut permettre à un attaquant d'accéder, modifier ou supprimer les données de la base.

En utilisant PreparedStatement (Java) ou NamedParameterJdbcTemplate (Spring), tu sépares le code SQL des données utilisateur, bloquant toute tentative d'injection SQL.

Si tu veux des précisions ou une autre approche, dis-moi !